

S E C S / H S M S C o m m u n i c a t i o n P a c k a g e
(T D S)

(Trust Design simple Secs/hsms communication library)

Programmer's Manual

Version 10.060 : 2010.06.10
Version 11.083 : 2011.08.27
Version 11.091 : 2011.09.08
Version 11.103 : 2011.10.15
Version 11.110 : 2011.11.01
Version 11.111 : 2011.11.15
Version 11.120 : 2011.12.01
Version 12.010 : 2012.01.01
Version 12.030 : 2012.03.01
Version 12.070 : 2012.07.07
Version 12.121 : 2012.12.12
Version 14.040 : 2014.04.25
Version 14.060 : 2014.06.01
Version 15.050 : 2015.05.08
Version 15.080 : 2015.08.01
Version 15.110 : 2015.11.13
Version 16.011 : 2016.01.15
Version 16.012 : 2016.02.25
Version 16.040 : 2016.04.05
Version 16.060 : 2016.06.10
Version 17.100 : 2017.10.06
Version 18.010 : 2018.01.10
Version 18.020 : 2018.02.10
Version 18.041 : 2018.04.23
Version 18.050 : 2018.05.28
Version 18.070 : 2018.09.25
Version 19.070 : 2019.07.05

Trust Design Limited Liability Company

Chino City Nagano Prefecture Japan

E-mail: info@trust-design.co.jp
URL: <http://www.trust-design.co.jp>

T a b l e o f C o n t e n t s

1. Function overview

2. Data structure

2.1 Data structure

2.2 SECS/HSMS communication module internal data configuration

3. SECS/HSMS Communication Library API Specification

3.1 SECS/HSMS Message communication function (usually API)

3.2 SECS/HSMS Message communication function (abbreviated API)

3.3 SECS Message construction, analysis function

3.4 SECS message construction and analysis function using SML format message structure definition

3.5 Other features

4. Tool

4.1 Communication control process

4.2 Communication data queue table reference tool

4.3 Error list display

A. SML format message structure definition file format

B. SECS/HSMS Communication library (C# version) API specification

B.1 SECS/HSMS Message communication function (usually API)

B.2 SECS/HSMS Message communication function (abbreviated API)

B.3 SECS Message construction, analysis function

B.4 SECS message construction and analysis function using SML format message structure definition

C. SECS/HSMS Communication library (Visual Basic version) API specification

C.1 SECS/HSMS Message communication function (usually API)

C.2 SECS/HSMS Message communication function (abbreviated API)

C.3 SECS Message construction, analysis function

C.4 SECS message construction and analysis function using SML format message structure definition

D. SECS/HSMS Communication library (Java version) API specification

D.1 SECS/HSMS Message communication function (usually API)

D.2 SECS/HSMS Message communication function (abbreviated API)

D.3 SECS Message construction, analysis function

D.4 SECS message construction and analysis function using SML format message structure definition

E. SECS/HSMS library (Function limited Java (not using JNA) version) API specification

E.0 SECS/HSMS Communication parameter configuration file (.ini file) format

E.1 SECS/HSMS Message communication function (usually API)

E.2 SECS/HSMS Message communication function (abbreviated API)

E.3 SECS Message construction, analysis function

1. Function overview

This library performs communication based on the SECS standard that SEMI can determine.
This library conforms to following standards.

+ E4	: SECS-I	Message transfer	(RS232C connection)
+ E5	: SECS-II	SECS Message contents	
+ E37	: HSMS	High-Speed SECS message service (HSMS) generic service	(TCP/IP)
+ E37.1	: HSMS-SS	High-speed SECS message service single session mode	(HSMS-SS)
+ E37.2	: HSMS-GS	High-speed SECS message service general session	(HSMS-GS)

This library has following features.

(1) Supports following as an operating platform.

(a) Windows	(2000, XP, 7, 8, 10)
(b) Linux	(2.4.21 or later)
(c) FreeBSD	(10.1 or later)
(d) Solaris (Intel)	(11.4 or later)
(e) MacOS X	(10.6.3 or later)
(f) HP-UX	(11.0 or later)

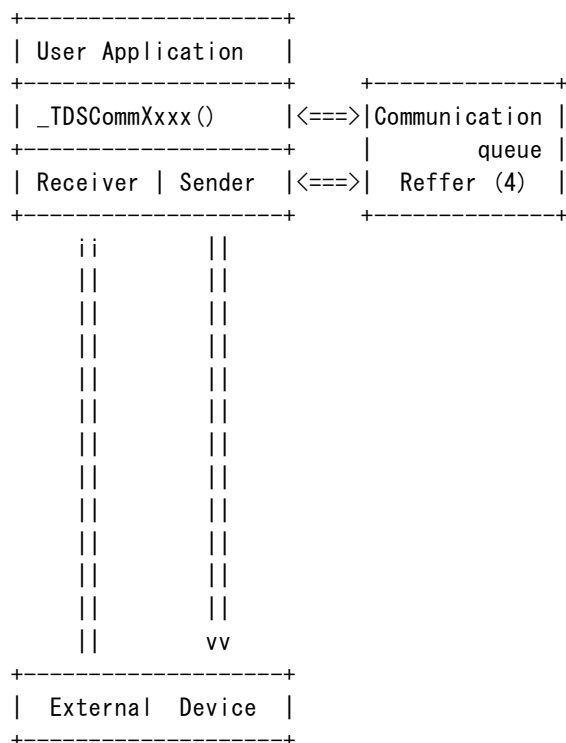
(2) Supports following as a transmission medium.

- (a) RS232C Serial connection (SECS-1)
- (b) TCP/IP Network connection (HSMS-SS, HSMS-GS)

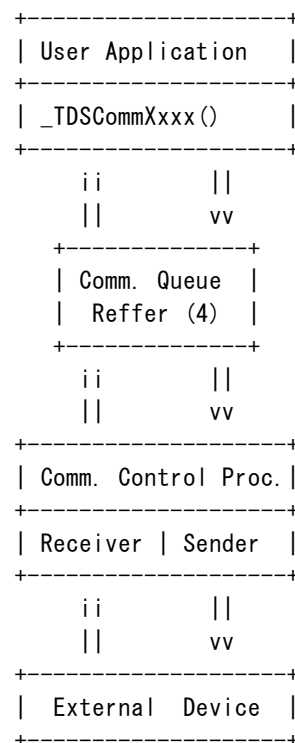
(3) The following operation modes of communication control unit are supported.

- (a) Act as a thread in user Application
- (b) Act as a communication control process

Process related diagram case (a)



Process related diagram case (b)



(4) Supported queue structure of communication data connecting user AP and library unit

- (a) Local memory table
- (b) Shared memory table
- (c) Disk file

No	Feature	(a)	(b)	(c)
1	Supported platform (Windows)	0	0	0
	Supported platform (Linux)	0	0	0
	Supported platform (FreeBSD)	0	0	0
	Supported platform (Solaris 11)	0	0	0
	Supported platform (MacOS X)	0	0	0
	Supported platform (HP-UX)	0	0	0
2	Processing in user AP	0	X	0
	Processing by communication control process	X	0	0
3	Table access speed	High	High	Middle
4	Send/receive communication message larger than table record size	0	X	0
5	Reference from outside of table data (such as tdsd)	X	0	0

(5) The following two types of SECS communication message structure analysis and construction means are provided.

- (a) Analyze and construct all communication message items by calling API.
- (b) Analyze and construct by referring to communication message structure definition file described in SML format. Referring to the message by message name, and referring to and setting necessary message items by item name.

(Note 1) Processing speed is higher in (a).

(6) This library uses following resources.

Resources used can be forcibly released by "tdsm -r". (UNIX only. Refer to 4.2)

- (a) Resource lock semaphore (UNIX), mutex (Windows)
 - + Communication trace Access control
 - + Execution trace Access control
 - + Send queue Access control
 - + Receive queue Access control
 - + Sending result queue Access control
- (b) Shared memory for queue data storage (only when shared memory)
 - + Send queue For storage
 - + Receive queue For storage
 - + Sending result queue For storage

(Note 2)

This package uses following ports of UDP/IP for license management.
 Also use following class D address as UDP/Multicast address. Please set not to block these by firewall etc. of your computer.

- 36274/udp
- 239.254.200.74

However, including Internet connection environment, it can be used even when network connection can not be made and NIC does not exist, there are no functional restrictions on usage as compared with internet connection environment.

2. Data structure

(1) Data structure

(2) SEC/HSMS communication module internal data configuration

2.1 Data structure

- (1) SECS/HSMS Communication parameter configuration file (.ini file)
- (2) Communication message queue data file
- (3) Communication trace file

(1) SECS/HSMS Communication parameter configuration file (.ini file)

(a) File attribute

Create at any position, with any name.

(b) Overall structure

```
[Section]           // Section name
Token = Parameter   // Comment
# Comment line
```

(Note 0) The file name is free but must not contain ','.

(Note 1) Multiple connection conditions can be set with different section names in one file.
The section name is specified at the Open call of this library.

(Note 2) Section names [DEFAULT] and [SECSCOMMON] are scanned regardless of section names.
Therefore, it is possible to specify common specifications in the [DEFAULT] or [SECSCOMMON] section and specify only individual specifications in each section.
Each section is read in the following order regardless of the position where the section is defined, and the values read later are prioritized.

Reading order	Priority	Section name
1.	3.	[DEFAULT]
2.	2.	[SECSCOMMON]
3.	1.	[Specified section]

However, for DEVID, the specified value in each section read in the above reading order will be added to all sections in common. Therefore, do not specify DEVID in the [DEFAULT] and [SECSCOMMON] sections if you want to use different settings in each section of the specification. In particular, when performing multiple connections (multiple _TDSCommOpen()) in a system using TDS (because the first deviceID is used for the TDS internal queue identification ID and trace file name), for those connections, (At least the first device ID) must be set so as not to overlap. Therefore, in this case, do not set DEVID to [DEFAULT] or [SECSCOMMON], and must be specified in [SECTION] used by each individual _TDSCommOpen() calls.

(Note 3) Specify section names and token names with alphanumeric characters, and are not case sensitive.

(Note 4) This file is scanned at specified (INTER3) time intervals also after Open processing of this library. Tokens whose parameter values can be changed during execution are shown in (d).

(Note 5) When an integer value is given as a parameter, if the first character is x or 0x, it can be specified in hexadecimal. If the parameter string contains a space or comma (,), enclose the entire parameter with ".

(Note 6) Bits and items described as <reserved> in this section, and undescribed bits must be =0.

(Note 7) This library ignores descriptions of the sections in this file other than [DEFAULT], [SECSCOMMON], and specified section. Even in the specified section, tokens other than the tokens described below are ignored. Therefore, it is possible to describe user-specific parameters in this file.

< Reference >

The result of acquiring the contents described in this file using `_TDSCommOpen ()` is output as a communication trace. By confirming that the output is intended, the correctness of the setting can be confirmed.

< Example >

[DEFAULT]

```

INTERO      = 100
PORT        = 22300
DEVID       = x20
DEVID       = x21

QUEDIR      = /tmp
QUESIZE     = 1024
QUEREC      = 255

TRCDIR      = /tmp
TRCDELTIME  = 081500
TRCDELDAY   = 000007
TRCTTYPE    = 3

```

[HOST]

```

SECSMODE    = 0x01

```

[EQUIP]

```

SECSMODE    = 0x71
HOST        = "192.168.2.165"

```

[MYSECT]

```
// User-specific section
```

```

MYTOKEN1    = "MYPARAM1"           // User-specific parameters
MYTOKEN2    = "MYPARAM2"

```

(c) Token and Parameter

SECSMODE = 0xffffffff // SECS communication parameter

```
// UTS          JIHG FEDC BA98 7654 3210
// +-----+-----+-----+-----+-----+-----+
// |||          |||| |||| |||| |||| ||+--- Communication (0:SECS      1:HSMS      )
// |||          |||| |||| |||| |||| ||  Type          (2:SECS on TCP 3:<reserved> )
// |||          |||| |||| |||| |||| ||+---- HSMS Passive IP-Address force open
// |||          |||| |||| |||| |||| |          (0:No      1:Yes      )
// |||          |||| |||| |||| |||| |          Set to 1 if "[ERROR] INITIALIZE (-98 ...)" is
// |||          |||| |||| |||| |||| |          recorded in the communication trace.
// |||          |||| |||| |||| |||| ||+---- HSMS Address type (0:IPv4      1:IPv6      )
// |||          |||| |||| |||| ||||
// |||          |||| |||| |||| ||||+--- Device type          (0:Host      1:Equipment )
// |||          |||| |||| |||| ||||+--- SECS Connection type (0:Slave      1:Master      )
// |||          |||| |||| |||| ||||+--- HSMS Connection type (0:Passive    1:Active      )
// |||          |||| |||| |||| ||||+--- HSMS Resolve host name (0:OFF      1:ON      )
// |||          |||| |||| ||||
// |||          |||| |||| ||||+--- HSMS Passive side accept priority
// |||          |||| |||| |||| 0: Immediately disconnect the next Accept, until the
// |||          |||| |||| |||| connection that has been accepted is disconnected.
// |||          |||| |||| |||| 1: Last accepted connection is effective and disconnect
// |||          |||| |||| |||| the previous connection.
// |||          |||| |||| |||| (Note 1) Normally, in HSMS set (SECSMODE&0x0100)==0.
// |||          |||| |||| |||| In case of necessary to avoid an abnormal condition
// |||          |||| |||| |||| on the Active side, set to !=0.
// |||          |||| |||| ||||
// |||          |||| |||| ||||+--- The meaning of the 15th bit of HSMS sessionID
// |||          |||| |||| |||| 0: SessionID uses 0th bit to 15th bit
// |||          |||| |||| |||| 1: As with the SECS-1 deviceID, the 15th bit is a
// |||          |||| |||| |||| Reverse bit.
// |||          |||| |||| |||| (Note 2) Normally, in HSMS set (SECSMODE&0x0200)==0.
// |||          |||| |||| |||| The sessionID (deviceID) is in the range of 0x0000 to
// |||          |||| |||| |||| 0x7fff in HSMS-SS, and 0x0000 to 0xffff in HSMS-GS.
// |||          |||| |||| |||| In special cases where you want to treat the HSMS
// |||          |||| |||| |||| sessionID the same as SECS-1 deviceID, set !=0.
// |||          |||| |||| ||||
// |||          |||| |||| ||||+---- Handling of sessionID (deviceID) in HSMS Select,
// |||          |||| |||| |||| Deselect, Separate Request.
// |||          |||| |||| |||| 0: Always set to 0xffff.
// |||          |||| |||| |||| 1: As with Data transmission, use the specified
// |||          |||| |||| |||| SessionID (DeviceID).
// |||          |||| |||| |||| (Note 3) Normally, in HSMS-SS set (SECSMODE&0x0400)==0
// |||          |||| |||| |||| and in HSMS-GS set to !=0.
// |||          |||| |||| ||||
// |||          |||| |||| ||||+---- Handling of W-bit in HSMS Select, Deselect, LinkTest
// |||          |||| |||| |||| Request
// |||          |||| |||| |||| 0: Set to 0
// |||          |||| |||| |||| 1: Set to 1
// |||          |||| |||| |||| (Note 4) Normally, in HSMS set (SECSMODE&0x0800)==0.
// |||          |||| |||| |||| Also for special conditions where W-Bit must be
// |||          |||| |||| |||| enabled for necessary control messages, set !=0.
// |||          |||| |||| ||||
// VVV          VVVV VVVV
```

< Continue to next page >

< Continue from previous page >

```

//  VVV          VVVV VVVV
//  UTS          JIHG FEDC BA98 7654 3210
//  +-----+-----+-----+-----+-----+-----+-----+-----+
//  |||          ||| |||
//  |||          ||| |||+-- Disconnection procedure for multiple connection request in
//  |||          ||| ||| HSMS passive operation.
//  |||          ||| ||| 0: Disconnect 2nd accepted connection immediately.
//  |||          ||| ||| 1: After accepting the 2nd connection, in response to the
//  |||          ||| ||| subsequent Select Request, return Status=3 and wait for
//  |||          ||| ||| Separate Request (or T0) to disconnect.
//  |||          ||| ||| (Note 5) Although it is often assumed that =0 in the normal
//  |||          ||| ||| HSMS system, it may be necessary to set !=0 in a system
//  |||          ||| ||| that allows Reject.
//  |||          ||| |||
//  |||          ||| |||+--- Operation when the number of SessionID with Selected state
//  |||          ||| ||| is >0 after HSMS Separate Request send/receive processing.
//  |||          ||| ||| 0: Continue TCP/IP connection
//  |||          ||| ||| 1: Disconnect TCP/IP connection
//  |||          ||| ||| (Notice 6) If the number of Selected State SessionID
//  |||          ||| ||| becomes 0, the TCP/IP connection is always disconnected.
//  |||          ||| ||| As a normal setting, both HSMS-SS and HSMS-GS are set to
//  |||          ||| ||| =0. In HSMS-GS, set =1 if Separate Request is used for
//  |||          ||| ||| closing SessionID of all Select State at one time.
//  |||          ||| |||
//  |||          ||| |||+---- Operation when the number of SessionID with Selected state
//  |||          ||| ||| becomes 0 after sending and receiving HSMS Deselect Request.
//  |||          ||| ||| 0: Continue TCP/IP connection
//  |||          ||| ||| 1: Disconnect TCP/IP connection
//  |||          ||| ||| (Note 7) If the number of Selected State SessionID is >0,
//  |||          ||| ||| the TCP/IP connection state is always maintained.
//  |||          ||| ||| In the normal setting, in the case of HSMS-SS (as describ-
//  |||          ||| ||| ed in SEMI E37.1), Deselect may not be issued, so
//  |||          ||| ||| the setting value of this bit may be either 0 or 1. In
//  |||          ||| ||| HSMS-GS, set to 1.
//  |||          ||| ||| In HSMS-GS, Set to =0 in case of close of TCP/IP
//  |||          ||| ||| connection depends on Separate Request. However, even in
//  |||          ||| ||| this case, if the Selected State Session count is 0 and
//  |||          ||| ||| passes T7T0 time, the state shifts to the Close State.
//  |||          ||| |||
//  |||          ||| |||+----- Handling of HSMS Select Request
//  |||          ||| ||| 0: Perform Select processing TCP/IP connection, not for
//  |||          ||| ||| each SessionID
//  |||          ||| ||| 1: Perform Select processing for each SessionID
//  |||          ||| ||| (Note 8) In general, set =0 in HSMS-SS, and set =1 in
//  |||          ||| ||| HSMS-GS.
//  |||          ||| ||| In the case of =1, all sessionID to be processed must be
//  |||          ||| ||| specified in DEVID, and set (DEVMODE&0x0001)==0. Also
//  |||          ||| ||| And (SECSMODE&0x0400) is always treated as !=0.
//  VVV          VVVV In the case of =0, normally set (SECDMODE&0x0400)==0.

```

< Continue to next page >

< Continue from previous page >

```

// VVV          VVVV
// UTS          JIHG FEDC BA98 7654 3210
// +-----+-----+-----+-----+-----+-----+-----+-----+
// |||          |||
// |||          |||+--- When Active connection is made automatically at the time of HSMS
// |||          ||| operation, wait time from disconnection detection to execution of
// |||          ||| next connect operation.
// |||          ||| 0: Immediate
// |||          ||| 1: After T5 time
// |||          |||
// |||          ||+--- Whether multiple connections can be made on the same TCP/IP Port#
// |||          || during HSMS Passive connection.
// |||          || 0: No : The second and subsequent connections to the same TCP/IP
// |||          || Port# are processed according to the (SECSMODE&0x1100)
// |||          || value, and multiple connections are not allowed at the
// |||          || same time.
// |||          || 1: Yes: Accept multiple connections (Connect and subsequent
// |||          || Select requests) to the same TCP/IP Port#.
// |||          || (Note 9) In general, it seems unlikely that multiple connections
// |||          || to the same TCP/IP Port# in HSMS Passive connection will be
// |||          || allowed. It is assumed that this bit is turned ON in host
// |||          || operation etc. where connection from the same device that
// |||          || operates under the same condition must be permitted.
// |||          || If this bit is ON and multiple connections on the same TCP/IP
// |||          || Port# are permitted during HSMS Passive connection, all the
// |||          || following conditions must be cleared.
// |||          || + You must _TDSCommOpen() call to make a passive connection
// |||          || for every other device using that TCP/IP Port#.
// |||          || (Even if TCP/IP Port# is the same, it can not be single
// |||          || _TDSCommOpen() call.
// |||          || + It does not determine which of the multiple _TDSCommOpen()
// |||          || is connected to which the other side(device) is not actually
// |||          || connected. The call order of _TDSCommOpen() does not
// |||          || necessarily match the connection order from the other side.
// |||          || Therefore, it is necessary to specify the connection partner
// |||          || by DEVID or message itself by message exchange after
// |||          || connection.
// |||          || + The number of calls of _TDSCommOpen() for passive connection
// |||          || in one process must be 64 or less (at present).
// |||          || + If you use _TDSCommClose() on the first _TDSCommOpen() for a
// |||          || passive connection that uses the same TCP/IP Port#, all
// |||          || passive connections for that TCP/IP Port# will no longer be
// |||          || possible. That is, _TDSCommClose() corresponding to the first
// |||          || _TDSCommOpen() for the same TCP/IP Port# must be performed
// |||          || after all the processing for that Port# is not necessary.
// |||          || + Connection requests exceeding the number for which
// |||          || _TDSCommOpen() has been executed are disconnected regardless
// |||          || of the (SECSMODE&0x0100) specification. The disconnection
// |||          || method follows the specification of (SECSMODE&0x1000).
// |||          ||
// |||          || < Continue to next page >
// VVV          VV

```

< Continue to next page >

< Continue from previous page >

```
// VVV          VV
// UTS          JIHG FEDC BA98 7654 3210
// +-----+
// ||          ||
// ||          ||      < Continue from previous page >
// ||          ||
// ||          ||      + Communication can not be performed using the SECS/HSMS
// ||          ||      communication control process (tdsc). That is, the mode of
// ||          ||      _TDSCCommOpen() must be (mode&0x03)==0x02.
// ||          ||      + When using a disk file as the SECS/HSMS communication message
// ||          ||      send/receive queue ((QUEMODE&0x80)==0), set the first
// ||          ||      DeviceID specified by DEVID used by each _TDSCCommOpen() to
// ||          ||      different values. (Refer to the description of DEVID for
// ||          ||      details.)
// ||          ||
// ||          ||      ++----- Handling of TCP/IP KEEPALIVE in HSMS connection
// ||          ||      0: By setting of OS      2: KEEPALIVE function OFF
// ||          ||      1: <reserved>           3: KEEPALIVE function ON
// ||          ||
// ||          ||
// ||          ||
// ||          ||      ++-- In the SECS-1 connection, whether send request can be made during BLOCK
// ||          ||      receiving during MULTI BLOCK transfer.
// ||          ||      0: No : After receiving all BLOCKs, start send processing.
// ||          ||      1: Yes: Sending process is started even during MULTI BLOCK receiving
// ||          ||
// ||          ||      +---- In the SECS-1 connection, handling of the BLOCK number when receiving the first
// ||          ||      BLOCK at the time of MULTI BLOCK transfer.
// ||          ||      0: Only =1 is valid.
// ||          ||      1: Receives subsequent BLOCKs with the received BLOCK number as the first BLOCK
// ||          ||      number.
// ||          ||
// ||          ||      +---- In the SECS-1 connection, processing when a BLOCK number other than consecutive
// ||          ||      expected BLOCK numbers is received during MULTI BLOCK transfer.
// ||          ||      0: End immediate receiving processing as error BLOCK (-E_ILLBLOCK)
// ||          ||      1: Reply NAK, prompt sender to resend
```

< Continue to next page >

< Continue from previous page >

DEVMODE = 0xffffffff // Device control mode

```

//          G FEDC BA98 7654 3210
//  +---+---+---+---+---+---+
//          | | | | | | | | | | +--- Device ID check
//          | | | | | | | | | | 0: Yes: Send and receive only the device ID
//          | | | | | | | | | |      specified by DEVID, and check at the time
//          | | | | | | | | | |      of sending and receiving
//          | | | | | | | | | | 1: No : When sending, send the specified deviceID,
//          | | | | | | | | | |      and when receiving, report the received
//          | | | | | | | | | |      device ID as it is
//          | | | | | | | | | | +--- Processing when receiving a secondary message that
//          | | | | | | | | | |      is not in receiving wait state
//          | | | | | | | | | | 0: Handle as error
//          | | | | | | | | | | 1: Receive processing as it is
//          | | | | | | | | | | +--- Block number for SECS-1 SingleBlock
//          | | | | | | | | | | 0: Set to 0      1: Set to 1
//          | | | | | | | | | | +--- <reserved>
//          | | | | | | | | | | +----- Transaction management
//          | | | | | | | | | |      (User Data sending and receiving)
//          | | | | | | | | | | 0: Yes          1: No
//          | | | | | | | | | | +----- Transaction management
//          | | | | | | | | | |      (Control information sending and receiving)
//          | | | | | | | | | | 0: Yes          1: No
//          | | | | | | | | | | +----- End transaction in S9FX receiving
//          | | | | | | | | | | 0: Finish        1: Not finish
//          | | | | | | | | | | +----- In the case of HSMS, Transaction ends upon receipt
//          | | | | | | | | | |      of Reject request
//          | | | | | | | | | | 0: Finish        1: Not finish
//          | | | | | | | | | | +----- T3TimeOut Report (S9F9) Auto send
//          | | | | | | | | | | 0: Yes          1: No
//          | | | | | | | | | | +----- Undef. Device ID report (S9F1) Auto send
//          | | | | | | | | | | 0: Yes          1: No
//          | | | | | | | | | | +----- Data size over report (S9F11) Auto send
//          | | | | | | | | | | 0: Yes          1: No
//          | | | | | | | | | | +----- When a secondary message is not received, etc.
//          | | | | | | | | | |      Abnormal data report (S9F7) Auto send
//          | | | | | | | | | | 0: Yes          1: No
//          | | | | | | | | | | +----- Reject request at HSMS Auto send
//          | | | | | | | | | | 0: Yes          1: No
//          | | | | | | | | | | +----- Automatic disconnection during communication when
//          | | | | | | | | | |      HSMS T6 Timeout occurs
//          | | | | | | | | | | 0: Yes          1: No
//          | | | | | | | | | | +--- <reserved>

```

< Continue to next page >

< Continue from previous page >

```

XDEV      = 9999          // Maximum number of connected devices
DEVID     = "999,0xff"    // Connected device ID
// The maximum number of XDEV can be specified, and multiple lines can be separated by
// ',' on one line. Or, specify in multiple lines. Every DEVID is valid across
// [DEFAULT], [SECSCOMMON] and designated sections.
// This can not be omitted if (DEVMODE&0x01)==0 mentioned above.
// (Note 1) Normally 0x0000-0x7fff is set for SECS-1 connection.
// Set 0x0000 to 0xffff for HSMS connection. When using 0x8000-0xffff, must set
// (SECSMODE&0x0200)==0.
// (Note 2) Multiple connections (any number of APs) in one system (in case of multiple
// _TDSCommOpen()) is performed, set the first connected deviceID so as not to be
// duplicated for each connection.
// If multiple connections must use the same DeviceID, assign a number that is not
// used in any connection as the first DeviceID, and specify a number to be used as
// the second and subsequent DeviceIDs. In this case, _TDSCommSend() must always
// specify the DeviceID of the send data.
// The connection deviceID at the beginning can not be duplicated because it is used
// as a key to identify the ID of the queue used internally by TDS and the trace
// file. Therefore, .ini is changed for each _TDSCommOpen() so that multiple DEVIDs
// specified by _TDSCommOpen() do not duplicate. Or in .ini [DEFAULT], DEVID is not
// defined, and it is necessary to define DEVID of [SECTION] to be used for each
// _TDSCommOpen() as a different value (at least the first connection DeviceID).
// However, if Local Memory is used as the internal queue ((QUEMODE&0xc0)==0x80),
// this is not the case as long as trace file names are not duplicated by other
// methods (for example, changing the storage directory). (In this case, same first
// DeviceID can be used in multiple connections.)

XMSGSIZE  = 999999999     // Maximum SECS message byte length
// It is not possible to send and receive SECS messages whose length exceeds this size.
// Therefore, specify more than the maximum length of the message to be sent and
// received. However, since the memory is secured on the system in relation to the
// designated size, it is not preferable to designate the meaninglessly large size.
// (Note) Depending on the specification, this library may not operate because memory
// can not be allocated.
// If you use SML format message definition file, you need to keep in mind the lower
// MDMXPOOL value as well.
// Also note the size of the message storage area when calling the SECS message
// construction functions _TDSMssgInit() and _TDSMDMssgInit().

XTRANX    = 999999        // Maximum number of transactions to process simultaneously
// Significantly if (DEVMODE&0x30)!=0x30 as described above
// Related to QUEREC described later.
// Specify a large value, with some leeway, than the maximum number of open
// transactions expected.

```

< Continue to next page >

< Continue from previous page >

```

SRCID0    = 99999      // SourceID given to user AP sending message      ( 0 - 32767)
SRCID1    = 99999      // SourceID given to HSMS control message        ( 0 - 32767)
// (Note) If SRCID0 and SRCID1 have the same value, Transaction ID of the return value
// when sending of the primary message is requested by CommSend() may be different
// from the ID actually assigned at the time of sending.
// Refer to CommSend() for details.

XIDMIN    = 99999      // Minimum value of transaction ID to be assigned      ( 1 - 65535)
XIDMAX    = 99999      // Maximum value of transaction ID to be assigned      ( 1 - 65535)
// The upper 16 bits of user AP sending message and system byte added to HSMS control
// message are SRCID0 value and SRCID1 value, respectively, and lower 16 bits are
// XIDMIN to XIDMAX values.
// (Note) It is also possible to set a value larger than 65535 as XIDMIN and XIDMAX,
// and use it as the transaction ID used up to the upper 16 bits of the system byte.
// In this case, care should be taken to ensure that the generated system byte value
// does not overlap between the user AP sending message and HSMS control message as
// SRCID0 and SRCID1 in relation to transaction ID.

INTER0    = 99900      // Communication control unit processing interval      (unit : ms)
// Communication idle time (mainly receiving process)
// <Important> Be sure to refer to the following [Information]

INTER1    = 99900      // Overall control unit processing interval            (unit : ms)
// Monitoring cycle of control processing (transaction management
// etc.)

INTER2    = 990        // Serial communication receiving interval              (unit : ms)
// Reception processing ENQ waiting time (SECS-1)

INTER3    = 999        // File related processing interval                    (unit : s)
// Re-read .ini file, old file deletion process monitoring cycle
// Communication trace processing Connection retry interval to
// external monitoring AP

```

```

// [Information]
// The value of INTER0 largely affects the interval at which TDS receives the SECS Message
// from the other end.
// In receiving process that is executed periodically (depending on INTER0), TDS wait an
// interval of INTER0 until the next receiving operation is executed if there is no Message
// receiving. When Message transmission from the other end is fast, Message sent
// continuously is received continuously, but once Message receiving process is
// interrupted, the next receiving processing will be after INTER0(ms). Therefore, if
// Message transmission from the other end continues at a relatively high speed, it is
// necessary to set the INTER0 value to a relatively small value (about 10 to 100 ?).
// However, reducing this value also increases overhead, so it is necessary to set
// appropriately considering the performance of the entire operating system.

```

< Continue to next page >

< Continue from previous page >

```

SDEVICE    = "XXXXXXX" // Serial connection device name
                // This token can not be omitted if (SECSMODE&0x01)==0
                // Otherwise unused
SSPEED     = 99999      // Serial connection bit rate
                // = 300 / 600 / 1200 / 2400 / 4800 / 9600 (/ 19200 / 38400 Etc)
SCSIZE     = 9          // Serial connection character bit size
                // = 7 / 8                                     (Normaly, can not set except =8)
SPARITY    = 9          // Serial connection parity type
                // = 0: None                                     (Normaly, can not set except =0)
                //   1: ODD Parity
                //   2: EVEN Parity
SSTOP      = 9          // Serial connection #of stop bit
                // = 1 / 2                                     (Normaly, can not set except =1)

HOST       = "XXXXXXX" // HSMS TCP/IP Destination host name or IP-Address
                // This token can not be omitted if (SECSMODE&0x41)==0x41
                // Otherwise unused

PORT       = 99999      // HSMS TCP/IP Connection port number (1 - 65535)
                // This token can not be omitted if (SECSMODE&0x01)==0x01
                // Otherwise unused.
                // When using the same PORT for calls of multiple _TDSCCommOpen() in
                // the case of HSMS Passive connection ((SECSMODE&0x41)==0x01),
                // It must be enabled by setteing multiple connections on the same
                // TCP/IP Port# at HSMS Passive connection as set
                // (SECSMODE&0x020000)!=0.
                // See the section (SECSMODE&0x020000) for details.

LINKINT    = 9999      // HSMS Link test execution interval (Unit : s)
                // = 0: Do not run link test
                // > 0: Link test execution interval

XRETRY     = 99        // SECS Block transfer maximum number of retries

T1         = 999       // T1 Timeout (Inter character) <Unuse> (Unit :ms)
T2         = 999       // T2 Timeout (Block transfer) (Unit : s)
T3         = 999       // T3 Timeout (Message reply) (Unit : s)
T4         = 999       // T4 Timeout (Inter block) (Unit : s)
T5         = 999       // T5 Timeout (Connect, Shutdown) (Unit : s)
T6         = 999       // T6 Timeout (Control transaction) (Unit : s)
T7         = 999       // T7 Timeout (NOT SELECT) (Unit : s)
T8         = 999       // T8 Timeout (Inter network packet) (Unit : s)

TORECV     = 99999999 // Time to disconnect due to no message received (Unit : s)

```

< Continue to next page >

< Continue from previous page >

```

QUEDIR      = "XXXXXXX" // Send/receive queue file storage directory
                // In the case of relative path name, it is relative to the
                // position where the specified .ini file exists.
                // Not used in the case of (QUEMODE&0x80)!=0x00)

QUESIZE     = 9999      // SECS/HSMS Communication message send/receive queue file byte
                // length of 1 record. If the communication message is larger than
                // this size, the send/receive memory or file is automatically
                // created, deleted, and sent/received as designated as an extended
                // data area. Therefore, it does not have to be XMSGSIZE or more.
                // In general, it is desirable to set according to the size of
                // frequently sent and received messages.
                // However, when specifying a shared memory as a queue data
                // attribute, since the extended data area can not be used, it is
                // necessary to specify at least the maximum size of all
                // communication messages.

QUEREC      = 9999      // Number of records in SECS/HSMS communication message
                // send/receive queue area
                // Related to the XTRANX mentioned above. Specify a value larger
                // than the number of transactions that may be in the sent/received
                // queue (preferably twice or more is desirable).

QUEMODE     = 0xff      // Processing mode of SECS/HSMS communication message send/receive
                // queue area
                // 7654 10
                // +----+----+
                // ||| | |++ Transmit overwrite (0:No 1:Yes )
                // ||| | |++ Reception processing mode
                // ||| | | = 0: Receive all messages
                // ||| | | 1: Received only after the last receiving time
                // |||+----- Use of extended data area (0:No 1:Yes )
                // ||+----- Automatic deletion after use of extended data area
                // || (0:No 1:Yes )
                // |+----- Queue data Memory attribute (0:Local 1:Shared)
                // +----- Queue data attribute (0:Disk 1:Memory)
                //
                // (Note) When specifying a shared memory as a queue data area, it
                // is not possible to specify the use of the extended data area.
                // Therefore, in this case, the maximum size that can be
                // sent and received is the value specified by QUESIZE.
                // When shared memory is specified, SECS/HSMS communication
                // control needs to be a separate process, not a thread.

```

< Continue to next page >

< Continue from previous page >

```

THSTACKUSR = 999999999 // Stack size of thread executing callback function
THSTACKSYS = 999999999 // Thread stack size used internally by this library
// If =0, use system default value.
// (Note) When running TDS using JNA in a Java execution environment, or running
// a large AP, etc., it is necessary to specify a sufficient amount of stack
// size (2MB, 4MB, 8MB, etc.).

TRCTHOST   = "XXXXXXX" // Communication trace processing Host name or IP-address on which
// the external AP operates
TRCTPORT   = 99999      // Communication trace processing TCP/IP Port# for connecting to
// external AP
// If host name or IP-Address is specified for TRCTHOST and TCP/IP Port# is
// specified for TRCTPORT, TCP connection is made to specified TCP/IP Port# of that
// host, and communication trace is sent out sequentially in text format.
// If you do not use this function, set this parameter so that this function does
// not work. That is, TRCTHOST="" or TRCTPORT=0.
// In the setting in which this function operates, if the communication trace
// processing external AP is not operated, overhead for trying connection at a
// predetermined interval occurs.
// When using the communication trace processing external AP function, as an
// external AP, it is desirable to have a TCP/IP Server function that operates on
// TRCTHOST, receive sequential text messages from the connected TCP/IP Port, and
// run external AP that process them in advance.
//
// The following communication trace messages are sent to the external AP via TCP/
// IP.
// "YYMMDD.hhmmss.999:99:99:XXXXX-----X¥n"
// ~~~~~ ~~~~~ ~~~ ~ ~ ~ ~~~~~
// |      |      |      |      |      +-- Communication trace message
// |      |      |      |      | +----- Content code (see _TDSCommOpen() (*6) (Note 3))
// |      |      |      +----- Communication trace output level
// |      |      +----- Mili second
// |      +----- Time
// +----- Date

TRCDIR     = "XXXXXXX" // Trace file storage directory
// In the case of relative path name, it is relative to the position where the
// specified .ini file exists.

```

< Continue to next page >

< Continue from previous page >

```

TRCTTYPE  = 0xffff    // Communication message output format to communication trace
// BA98 7654 3210
// +---+---+---+
// ||++ |+++ |||+-- List format output                      (0:No 1:Yes)
// || | | | |+--- Hexadecimal format output                (0:No 1:Yes)
// || | | | |+---- Item data display format                (0: 1line 1:Multiple lines)
// || | | | +----- Hexadecimal display                  (0:16Bytes/line 1:20Bytes/line )
// || | | +----- List output format
// || | |          (0:TDS 2:SML 4:NSG/TS300)
// || | |          (1:<reserved> 3:<reserved> 5-7:<reserved>)
// || | +----- Message Name and Item Name Display        (0:No 1:Yes)
// || |          (Valid only when a valid message definition is specified in
// || |          MDMSSG. Keep in mind that there is a processing speed overhead
// || |          in this specification, and when specifying this specification,
// || |          be sure to specify the correct message definition file in
// || |          MDMSSG.)
// || +----- Message definition file format
// ||          0:SML 1:<reserved> 2:NSG/TS300 3:<reserved>
// |+----- Number of leading spaces in list display      (0:2 1:0 )
// +----- Message name single line display                (0:No 1:Yes)
//          (Enabled when item name display (bit#7) is enabled)
//
// (Note) When message names and item names are displayed using a message
// definition file specified in MDMSSG, a corresponding overhead occurs.
// Therefore, be careful when the execution speed is high (the frequency of
// communication is high). In such a case and when a valid message definition
// file is not specified, be sure to set bits7 and 11 to OFF.

```

```

TRCTOUT    = 0xffff    // Communication trace output mode
// D      8 765 10
// +---+---+---+---+
// +-+---+ |++ +-+ Trace output destination
// |      || = 0: No output
// |      || 1: Output to STDOUT
// |      || 2: Output to specified trace file
// |      || 3: Output to STDOUT and predetermined trace file
// |      |--- Trace file daily switch specification at output
// |      | = 0: Not performed
// |      | 1: Do .. Create a directory of dates and store in it
// |      | 2: Do .. Date is attached to head of the file name
// |      | 3: Do .. Date is attached after the file name
// |      +--- File switching specification when exceeding the specified size
// |      = 0: Not performed
// |      1: Do
// +-+ Number of trace files to keep
// = 0      : Keep all files
// 1 - 63   : Delete files before a specified number from the time of
//            switching files
//
// (Note) This specification is valid for files on the same day. Therefore, the
// number of designated files is stored on a daily basis. Also, deletion of
// old files is only performed on files prior to a specified generation when
// changing file names, and it does not apply to all previous files. Setting
// TRCDELTIME and TRCDELDATE parameters is also effective for organizing old
// trace files.

```

< Continue to next page >

< Continue from previous page >

```

TRCTLEVEL  = 9          // Communication trace output level
                        // = 0: Do not output
                        // 1: Communication Header (List)
                        // 2: Communication Error
                        // 3: Communication Header (Hexa)
                        // 4: Communication Message (List)
                        // 5: Communication Message (Hexa)
                        // 6: Communication Control code
                        // (Note) Specifying a value of +5 also outputs a trace for
                        // LinkTest.

TRCTATTR    = 0xffff    // Communication trace output attribute (Usually not changeable)
                        // F      8      3210
                        // +---+---+---+---+
                        // |      |      |||+-- Output time          YYMMDD.hhmmss
                        // |      |      ||+--- Output time (ms)      .999
                        // |      |      ||   If the 1st bit=1, the 0th bit is
                        // |      |      ||   unconditionally treated as 1
                        // |      |      ||
                        // |      |      |+---- Process ID number      :99999
                        // |      |      +---- Trace output level      :99
                        // |      |
                        // |      +----- Key string of trace line    :XX
                        // |
                        // +----- =0: Make end of line only LF
                        //          1: Make end of line CR/LF

TRCTSIZE    = 99999999 // Communication trace file size
                        // If a positive value is specified, the file is switched when the
                        // trace file size exceeds the value.

TRCTXDUMP   = 999      // Communication trace Hexa dump aborted bytes
                        // = 0: Output all bytes
                        // > 0: Data at byte positions larger than the specified number of
                        // bytes is not output except for the last line

```

< Continue to next page >

< Continue from previous page >

```

TRCPOUT   = 9           // Process trace output mode                (Refer TRCTOUT)
TRCPLEVEL = 9           // Process trace output level
// = 0: Do not output
// 1: Execution control
// 2: Transaction processing, old file deletion
// 3: Lower layer processing function
// 4: Send and receive messages
// 5: Internal message table access
// 6: LinkTest message processing
// 15: Bottom layer communication function I/O state
// 23: Idle loop
// 25: Bottom layer communication function I/O state empty loop
TRCPATTR  = 0xffff      // Process trace output attribute          (Refer TRCTATTR)
TRCPSIZE  = 99999999    // Process trace file size                 (Refer TRCTSIZE)
TRCPPRINT = 9           // Process print output level (For debugging use only, Do not set)

TRCUOUT   = 9           // User I/F function trace output mode     (Refer TRCTOUT )
TRCULEVEL = 9           // User I/F function trace output level
// = 0: Do not output
// 1: Execution control
// 3: Send and receive messages
// 4: Send and receive messages (List)
// 23: Idle loop
TRCUATTR  = 0xffff      // User I/F function trace attribute        (Refer TRCTATTR)
TRCUSIZE  = 99999999    // User I/F function trace file size        (Refer TRCTSIZE)
TRCUPRINT = 9           // User I/F print output level (For debugging use only, Do not set)

TRCDELTIME = hhmmss     // Old trace deletion execution time
// Old trace deletion is executed at the timing when the specified
// time has passed after the function start of this library.
TRCDELDAY  = YYMMDD     // Old trace deletion target progress date
// = 0: Do not do old trace file deletion processing
// !=0: Specify the date to be deleted for old trace files in
//      YYMMDD format from today.
//      000115 and -000115 mean one and a half months ago.
//      When YYMMDD is specified as a negative value (ex. -000015),
//      old trace deletion is executed once at _TDSCommOpen().
//      This process is executed only once a day (including the
//      execution immediately after startup).

```

< Continue to next page >

< Continue from previous page >

```

LICENSENAME= "XXXXXXX" // Name of individual or group who received license for operation
LICENSECODE= "XXXXXXX" // Code name of the individual or organization who received license
LICENSESER = "XXXXXXX" // Serial# in the individual or group who received the license
LICENSEDATE= "YYYYMMDD" // Acquisition date of execution license          (YYYYMMDD format)
LICENSEKEY = "XXXXXXX" // Execution license key code
//
// [Warning]
// If the license key code is not set, if the wrong license key code is set, if the
// expired operation check license key code is set, the operation is stopped after
// operation for a fixed time. Also, even if a valid operation confirmation license
// key code (Default License) is set, the operation is stopped after several hours
// of operation.
// In the case of a license that has been officially licensed for execution but is
// not a license key for operation confirmation (Default License), different PCs
// must be written with different license key code information.
// It is not necessary to set different license key codes for multiple APs using
// TDS operating in the same PC.

// The following items do not need to be set if none of the following is performed
// using SML format message definition file.
// + Display message name and item name in communication trace
// + SECS message parsing and construction using message definitions

MDMSSG      = "XXXXXXX" // Message definition file path name in SML format.
// In the case of relative path name, it is relative to the position where the
// specified .ini file exists.
// It is not necessary to specify this if you do not want to display message names,
// item names, etc. during communication trace, and specify file path names with
// _TDSMDMssgInitialize().

MDMXITEM    = 9999      // Maximum number of total data items used in each message
// (Items of the same attribute used in different messages are
// counted as one.)
MDMXMSG     = 9999      // Maximum number of messages to define
MDMXITEM    = 99999     // Maximum number of total items used in each message + maximum
// number of items when expanding messages
MDMXPOOL    =9999999    // Message definition setting data storage area size
// Size of area for storing data to be set in data item
//
// (Note 1) These items determine the size of the internal table used when
// performing message analysis and construction processing using the message
// definition. Since the internal table is also used as a work area for message
// analysis and construction, specify the number of messages to be actually
// defined and the size with enough margin compared to the number of items.
// Also, be aware of the XMSGSIZE parameter above when dealing with large sized
// messages.
//
// (Note 2) The construction, analysis, and communication trace output of
// communication messages using message definitions require more load than when
// not used. If you need to improve the communication speed, you should not
// specify execution of processing using message definition.

```

(d) Default value of parameter and possible change during execution

	:	Change	:	Change while running : 0:OK X:NG	
	:	while	:		
Token	:	running	:	Parameter default value	

SECSMODE	:	X	:	0x0001	(HSMS / Host / (Slave) / Passive / Unresolve name)
DEVMODE	:	X	:	0x1800	(Do DEVID management, transaction management) (End transaction with S9Fx, Reject Req) (Send automatic response other than S9F7 and Reject Req) (Automatically disconnect when T6 Timeout occurs)
DEVID	:	X	:		(If (DEVMODE&0x01)==0x00, it can not be omitted)
XDEV	:	X	:	16	(Maximum number of devices)
XMSGSIZE	:	X	:	4096	(Maximum SECS message size)
XTRANX	:	X	:	1024	(Maximum number of concurrent transactions)
SRCID0	:	0	:	0x0000	(Source ID given to user AP sending message)
SRCID1	:	0	:	0x7fff	(Source ID given to HSMS control message)
XIDMIN	:	0	:	1	(Minimum value of transaction ID to be assigned)
XIDMAX	:	0	:	65535	(Maximum value of transaction ID to be assigned)
INTER0	:	0	:	300	(ms) (Communication monitoring interval)
INTER1	:	0	:	1000	(ms) (Transaction monitoring interval)
INTER2	:	0	:	100	(ms) (SECS communication port monitoring interval)
INTER3	:	0	:	60	(s) (Configuration file monitoring interval)
SDEVICE	:	X	:	""	(If (SECSMODE&0x01)==0x00 this can not be omitted)
SSPEED	:	X	:	9600	(Bit rate)
SCSIZE	:	X	:	8	(Character size Unchangeable)
SPARITY	:	X	:	0	(Parity none Unchangeable)
SSTOP	:	X	:	1	(1 Stop bit Unchangeable)
HOST	:	X	:	""	(If (SECSMODE&0x41)==0x41 this can not be omitted)
PORT	:	X	:	0	(If (SECSMODE&0x01)==0x01 this can not be omitted)
LINKINT	:	0	:	0	(Do not execute LinkTest)
XRETRY	:	0	:	3	(number of retries)
T1	:	0	:	100	(T1 TimeOut value .. ms)
T2	:	0	:	10	(T2 TimeOut value ... s)
T3	:	0	:	45	(T3 TimeOut value ... s)
T4	:	0	:	45	(T4 TimeOut value ... s)
T5	:	0	:	10	(T5 TimeOut value ... s)
T6	:	0	:	5	(T6 TimeOut value ... s)
T7	:	0	:	10	(T7 TimeOut value ... s)
T8	:	0	:	10	(T8 TimeOut value ... s)
TORECV	:	0	:	0	(Time to disconnect due to no message received ... s)
QUEDIR	:	X	:	""	(Runtime current directory)
QUESIZE	:	X	:	1024	(Size to fit most SECS messages)
QUEREC	:	X	:	1024	(Maximum number of concurrent transactions or more)
QUEMODE	:	X	:	0x00b0	(Local memory use, extended data area use)

< Continue to next page >

< Continue from previous page >

```

      : Change :          Change while running : 0:OK X:NG
      : while :
Token : running : Parameter default value
-----
THSTACKUSR : X : 0 (Windows : System default)
              1048576 (UNIX : (1024*1024) Bytes)
THSTACKSYS : X : 0 (Windows : System default)
              1048576 (UNIX : (1024*1024) Bytes)

TRCTHOST : 0 : "" (Communication trace External AP function not used)
TRCTPORT : 0 : 0 (Communication trace External AP function not used)

TRCDIR : X : "" (Runtime current directory)
TRCTTYPE : 0 : 0x0001 (Output only List format in TDS format)

TRCTOUT : X : 0x0002 (Output only to trace file)
TRCTLEVEL : 0 : 5 (Output other than communication control code, LinkTest)
TRCTATTR : 0 : 0x810f or 0x010f (All information except source code information)
              (Windows : 0x810f, Unix : 0x010f)
TRCTSIZE : X : 5000000 (Perform trace file switching by size at 5 MB or more)
TRCTXDUMP : 0 : 0 (Number of aborted bytes for hexa dump)

TRCPOUT : X : 0x0000 (Do not output)
TRCPLEVEL : 0 : 5 (Output other than lower-layer communication, empty loop)
TRCPATTR : 0 : 0x812f or 0x012f (All information except source code information)
TRCPSIZE : X : 5000000 (Perform trace file switching by size at 5 MB or more)
TRCPPRINT : 0 : 0 (Do not print debug)

TRCUOUT : X : 0x0000 (Do not output)
TRCULEVEL : 0 : 5 (Output other than lower-layer communication, empty loop)
TRCUATTR : 0 : 0x812f or 0x012f (All information except source code information)
TRCUSIZE : X : 5000000 (Perform trace file switching by size at 5 MB or more)
TRCUPRINT : 0 : 0 (Do not print debug)

TRCDELTIME : 0 : 000000 (00:00'00")
TRCDELDAY : 0 : 000000 (Do not delete)

LICENSENAME : X : "" (Name of individual or group)
LICENSECODE : X : "" (Code name of the individual or organization)
LICENSESER : X : "" (Serial# in the individual or group)
LICENSEDATE : X : "" (Acquisition date of execution license)
LICENSEKEY : X : "" (Execution license key code)

MDMSSG : *2 : "" (Message definition file path name in SML format)
MDMXITEM : *2 : 1024 (Maximum number of total data items)
MDMXMSSG : *2 : 512 (Maximum number of messages to define)
MDMXITEM : *2 : 10240 (Maximum number of total items)
MDMXPOOL : *2 : 65536 (Message definition setting data storage area size)

```

(Note1) If the trace output is not set to output at startup, it can not be changed to the output to be output halfway. Therefore, output is not performed at startup, but if there is a possibility of output during the process, specify output at OUT parameter at startup and set the level to 0.

(*2) MDMSSG to MDMXPOOL become effective when _TDSMDMMsgInitialize() is executed after the change during execution becomes effective (after .ini file has been read for the open communication control identifier). Also, these parameters can not be changed during execution for an open control identifier for which communication control has already been started. Therefore, it is not possible to change the communication trace during communication during execution.

(2) Communication message queue data file

(a) File attribute

Automatically create following name at location specified by QUEDIR in .ini file.

(This file does not exist when using a memory table as a communication message queue)

```
+ Receive queue      : qxffffr.sjq          qxffffr.sjq.9999
+ Send queue         : qxffffs.sjq          qxffffs.sjq.9999
+ Send result queue  : qxffffc.sjq          qxffffc.sjq.9999
                     ~~~~~
x      : Host side is 'h', device side is 'e'
ffff   : Device ID in hexadecimal notation
```

When using multiple device IDs, use the first one of the DEVID specifications.
The file on the right side is the extended data area file.

(b) overall structure

Determine the following from the following tokens in the .ini file:

```
+ QUEDIR : Queue file creation position
+ QUESIZE : Data record length
+ QUEREC : Number of data records
```

(3) Communication trace file

(a) File attribute

Automatically creates following name at location specified by TRCDIR in .ini file, and deletes the old file automatically if specified.

+ Communication trace	: txxxxft.YYMMDD.trc	txxxxxft.YYMMDD.9999.trc
+ Communication error trace	: txxxxfe.YYMMDD.trc	txxxxxfe.YYMMDD.9999.trc
+ Processing trace	: txxxxfp.YYMMDD.trc	txxxxxfp.YYMMDD.9999.trc
+ User I/F function trace	: txxxxfu.YYMMDD.trc	txxxxxfu.YYMMDD.9999.trc

x	: Host side is 'h', device side is 'e'
ffff	: Device ID in hexadecimal notation
YYMMDD	: Trace output date (may not have YYMMDD by specification)
9999	: Serial number from 0000 (may not have 9999 by specification)

When using multiple device IDs, use the first one of the DEVID specifications.
 If no device ID is specified, the device ID as the file name is '0000'.
 If you specify trace file size, it will be the file name on the right side.

(b) overall structure

Determine the following from the following tokens in the .ini file:

+ TRCDIR	: Trace file creation position
+ TRCTTYPE	: Communication trace output format
+ TRCTOUT	: Communication trace output destination
+ TRCTSIZE	: Communication trace file size
+ TRCTLEVEL	: Communication trace output level
+ TRCPOUT	: Processing trace output destination
+ TRCPSIZE	: Processing trace file size
+ TRCPLLEVEL	: Processing trace output level
+ TRCUOUT	: User I/F function trace output destination
+ TRCUSIZE	: User I/F function trace file size
+ TRCULEVEL	: User I/F function trace output level

(Note2) The attributes of the communication error trace are defined by parameters related to the communication trace.

2.2 SEC/HSMS communication module internal data configuration

- (1) SECS/HSMS communication packet configuration
- (2) < Missing number >
- (3) Parameter configuration for user callback function
- (4) Parameter configuration for user trace output function

(1) SECS/HSMS communication packet configuration

```

typedef struct{
    TDLU32 len;    // Byte length including header part of communication data after header
} TDSECSLength;

typedef struct{
    TDLU16 did;    // DeviceID + Reverse bit
                  // FE          0
                  // +-----+
                  // |+-----+
                  // |          +----- DeviceID      (0x0000 - 0x7fff)
                  // +----- Reverse bit      (0:Host --> Equ.)
                  //                                     (1:Equ. --> Host)
    TDLU08 scd;    // S Code + Wait bit
                  // 76          0
                  // +-----+
                  // |+-----+
                  // |          +----- S Code      (0 - 127)
                  // +----- Wait bit      (0:No wait)
                  //                                     (1:With wait)
    TDLU08 fcd;    // F Code      (0 - 255)
    TDLU08 ptp;    // P Type      : Valid for HSMS (Block# for SECS)
                  // = 0          : SECS Code
                  // 1 - 127      : <reserved for appendix>
                  // 128 - 255    : <reserved>
    TDLU08 stp;    // S Type      : Valid for HSMS (Block# for SECS)
                  // = 0          : Data Message
                  // 1           : Select Request
                  // 2           : Select Response
                  // 3           : Deselect Request
                  // 4           : Deselect Response
                  // 5           : LinkTest Request
                  // 6           : LinkTest Response
                  // 7           : Reject Request
                  // 8           : <reserved>
                  // 9           : Separate Request
                  // 10          : <reserved>
                  // 11 - 127     : <reserved for appendix>
                  // 128 - 255    : <reserved>
    TDLU16 sid;    // Source ID      (Upper System Byte)
                  // When this library gives this for the primary message, it usually gives
                  // according to the following rules. This value can be changed in .ini, and
                  // does not have to be as follows. See the descriptions of SECID0, SRCID1,
                  // XIDMIN and XIDMAX in the .ini file.
                  // = 0          : Usual Message
                  // 1 - 32766    : <reserved>
                  // 32767        : Control Message
                  // 32768 - 65535 : <reserved>
    TDLU16 xid;    // Transaction ID (Lower System Byte)
                  // = 0          : <reserved>
                  // 1 - 65535    : Transaction ID
} TDSECSHead;

```

(Note 1) When using this structure in each function of this library, items consisting of multiple bytes are automatically converted to Byte order in own system. Therefore, there is usually no need to worry about the difference in Byte order in the user AP. That is, it automatically converts Byte Order of did, sid, xid, and ptp+stp in case of SECS-1.

(2) < Omission >

(3) Parameter configuration for user callback function

```
typedef struct{
    int      vec;      // Type                      (1:Receive  2:Send result)
    int      req;      // Request code
                        // = 0: Data received
                        // 61: Connect error
                        // 80: Illegal block#
                        // 81: T1 Time out occure
                        // 82: T2 Time out occure
                        // 83: T3 Time out occure
                        // 84: T4 Time out occure
                        // 85: T5 Time out occure
                        // 86: T6 Time out occure
                        // 87: T7 Time out occure
                        // 88: T8 Time out occure
                        // 89: Retry over
                        // 90: No message time out occure
                        // 91: Connected
                        // 92: Selected
                        // 97: Rejected
                        // 98: Deselected
                        // 99: Separate and Shudtown request
    int      rtn;      // Return value
                        // >=0: OK : Data length stored in 'msg' for receiving
                        // < 0: NG or Status change
                        // Refer to return value of _TDSCommRecv()
    int      devid;    // Sendt/Received message SECS Device ID
    int      sf;       // Sendt/Received message SECS SF-Code
    unsigned int xid;   // Sendt/Received message SECS Transaction ID
    TDSECSHead thd;     // Sendt/Received message SECS Header
    char      *msg;     // Received      message SECS Body
} TDSCBData;
```

(Note 1) Refer to (1) (Note 1)

(4) Parameter configuration for user trace output function

```
typedef struct{
    int      proc;     // Trace event code                      (Refer to 3.1 (1) (*6))
    int      date;     // Trace output date                    (YYYYMMDD)
    int      time;     // Trace output time                    (hhmmss)
    int      msec;     // Trace output mili second            (999)
    char      *msg;     // Trace message
} TDSTRData;
```

3. SECS/HSMS Communication Library API Specification

- (1) SECS/HSMS Message communication function (usual API)
- (2) SECS/HSMS Message communication function (abbreviated API)
- (3) SECS Message construction, analysis function
- (4) SECS Message construction and analysis function using SML format message structure definition
- (5) Other features

(Note 1) The following headers and libraries are required to use this library.

- + Windows
 - TDS.h
 - TDS.lib
 - TDS.dll
- + UNIX
 - TDS.h
 - Linux : libTDS.so (LD_LIBRARY_PATH)
 - FreeBSD : libTDS.so (LD_LIBRARY_PATH and LD_32_LIBRARY_PATH)
 - MacOS X : libTDS.dylib (DYLD_LIBRARY_PATH)
 - Solaris : libTDS.so (LD_LIBRARY_PATH)
 - HP-UX : libTDS.sl (SHLIB_PATH)

(Note 2) This library supports 32-bit code and 64-bit code. Therefore, when performing make on a 64-bit OS, specify the following architecture and use the corresponding (32-bit or 64-bit) library according to the application's corresponding architecture.

- | | 32-bit | 64-bit |
|---------|-------------------|-----------------|
| - MS/VS | : "vcvarsall x86" | "vcvarsall x64" |
| - gcc | : -m32 | -m64 |
| - clang | : -m32 | -m64 |
| - HP-C | : +DD32 | +DD64 |

(Note 3) Note that it may be necessary to specify the stack size when using this library.

(Note 4) With regard to error number which is the return value of library, list of its meaning and corresponding macro constants can be known by 'tdse'. See Section 4.3 for starting 'tdse'.

3.1 SECS/HSMS Message communication function (usual API)

(1) SECS/HSMS communication : Open processing

Analyzes parameters of the specified section in the specified .ini file, acquires SECS/HSMS communication conditions, and starts SECS/HSMS communication process.

Following two operations can be specified for operation of the module that executes SECS/HSMS communication processing, and should be selected appropriately.

- + Act as another thread in user AP
- + Use SECS/HSMS communication control process (tdsc)

```

int  _TDSCCommOpen(          // Return value                      (*1)
int   mode,                  // in  : Processing mode      (*2)
char  *ini,                  // in  : .ini file path name  (*3)
char  *sect,                 // in  : .ini file section name to use (*4)
int   (*cbrecv)(void*, TDSCBData*), // in  : User callback function address (for receive) (*5)
void  *parrecv,              // in  : Parameters given to 'cbrecv' (*5)
int   (*cbsend)(void*, TDSCBData*), // in  : User callback function address (for send) (*5)
void  *parsend,             // in  : Parameters given to 'cbsend' (*5)
int   (*fnctrc)(void*, int, char*), // in  : User trace output function address (*6)
void  *partrc)              // in  : Parameters given to 'fnctrc' (*6)

```

(Note 0) If this Call is executed multiple times in one system (regardless of the number of processes used and the number of threads), the DeviceID (at least the first DeviceID) defined in the .ini file (as DEVID) must be different.

Since the start DeviceID is used as the key for identifying communication queue used internally by TDS and name of the trace file name, different connections require a different DEVID (first value). Therefore, when performing multiple calls, when using same .ini, do not specify DEVID in [DEFAULT] and [SECSCOMMON]. (As DEVID does not give priority to later designations, all designations are added. Therefore, if DEVID is specified in [DEFAULT] or [SECSCOMMON], the top DeviceID will be the same.)

Be sure to specify DEVID only in individual [SECTION].

(*1) Return value

```

> 0: OK          : Control identifier used for subsequent processing
== 0: <reserved>
< 0: NG          : Error code
    -EBUSY       : Processing control block has already been opened
    -ENOMEM      : Memory allocation error for internal processing
    -ENOSPC      : SECS/HSMS communication control table is full
    -ENOENT      : Specified .ini file does not exist
    -EINVAL      : Specification of SECSMODE is not subject to processing
                   No device ID to be processed has been specified
                   'mode' specification is abnormal
    -ENOSYS      : Communication processing process (tdsc) execution error
Other .. : Other error

```


(*2) mode : Processing mode

FE	10
+-----+	
	++-- 0: <reserved>
	1: Use already started tdsc (SECS communication control process)
	2: Act as another thread in user AP
	3: <reserved>
+-----	Lock processing during _TDSMsgXxxx() processing
	0:No 1:Yes (Currently unused)
+-----	0: Open SECS/HSMS communication process.
	1: Only the contents of the specified .ini file are fetched as internal processing parameters, but Open processing is not performed.
	If you only analyze the SECS message and do not perform communication processing, you should turn this bit ON.

(Note 1) When the communication message queue is built on shared memory, it can not be operated as another thread in the user AP.

Also, when constructing a communication message queue on Local Memory, it must be operated as another thread in the user AP.

(*3) ini : Path name of .ini file

Use "tds.ini" as the path name if ini==0 or *ini==""

(*4) sect : Target section name in .ini file

Use "TDS" as the section name if sect==0 or *sect==""

- (*5) cbrecv : User callback function address (for receive)
 cbSEND : User callback function address (for send)

Specify the function address of the following format. Specify 0 if you do not use a callback function.

The same function may be set for both receive and send. In that case, cbdata->vec can be used to determine which function called the call.

```
int CBProc(          // <reserved> Must return =0
  void             *cbparam, // in : Parameters specified by _TDSCommOpen()
  TDSCBData        *cbdata)  // in : Refer to 2.2 (3)
```

cbrecv : It starts when receiving operation from this library (message from another side, connect, select, etc.) or when connection state change, error (T3 timeout, etc.) occurs. When this callback function is specified, if _TDSCommRecv() is called at the user AP, that Call results in an error.

cbSEND : Report the processing result for send request to this library.
 Therefore, the Call of _TDSCommSend() does not usually cause an error (except for parameter errors etc.).

(Note 2) Each callback function is executed in a dedicated processing thread.
 Therefore, functions that can be used by callback functions and their lower functions in other threads must be thread safe.
 Also, each time 'cbparam' is called back, its address is passed to the specified callback function. Note that if the callback function always uses it, the contents of cbparam must be maintained (whether or not the content has changed) whenever the connection is maintained.

(*6) fnctrc : User trace output function address

Specify the function address of the following format. Specify 0 if you do not use the user trace output function. When this function is specified, when outputting communication trace, pass each output line to this function.

The presence or absence of Call of this function depends on TRCTLEVEL of the communication trace output parameter described in .ini, but not on TRCTOUT.

```
int fnctrc(          // <reserved> Must return =0
void      *partrc,  // in  : Parameters specified by _TDSCommOpen()
TDSTRData *trdata) // in  : Refer to 2.2 (4)
```

(Note 3) trdata.proc (trace message content code) indicates the content of the trace message with the following values.

- = 0: SECS/HSMS : Comment specified by user using _TDSCommUserComment() for output
- 1: SECS/HSMS : Receive SECS message
- 2: SECS/HSMS : Send SECS message
- 4: HSMS : TCP/IP Port Initialization (Passive port open processing)
- 5: HSMS : TCP/IP Port Accept (Passive connection)
- 6: HSMS : TCP/IP Port Connect (Active connected)
- 8: HSMS : TCP/IP Port Shutdown (Shutdowned)
- 9: HSMS : TCP/IP Port termination (Passive port close processing)
- 11: HSMS : TCP/IP Port Connect (Active connected)
- 12: HSMS : Became 'SELECT' state
- 13: SECS : Transmitted and received control code related to receive processing
- 14: SECS : Transmitted and received control code related to send processing
- 17: HSMS : Received 'REJECT'
- 18: HSMS : Received 'DESELECT'
- 19: HSMS : TCP/IP Port Disconnected
- 20: SECS : Receive a communication packet with an abnormal block number
- 21: SECS : T1 Timeout has occurred
- 22: SECS : T2 Timeout has occurred
- 23: SECS/HSMS : T3 Timeout has occurred
- 24: SECS : T4 Timeout has occurred
- 25: HSMS : T5 Timeout has occurred
- 26: HSMS : T6 Timeout has occurred (Include LinkTest timeout)
- 27: HSMS : T7 Timeout has occurred
- 28: HSMS : T8 Timeout has occurred
- 29: SECS : Retry over has occurred
- 30: HSMS : Disconnected because no message has been received for specified time or more
- 31: SECS/HSMS : Receiving device ID is not to be processed
- 32: SECS/HSMS : Received message exceeds maximum message length
- 33: SECS/HSMS : Received secondary message that is not in waiting for receiving
- 34: SECS : Checksum error
- 35: SECS/HSMS : Processing sequence is not normal
- 36: SECS/HSMS : S9F1 Sent automatically
- 37: SECS/HSMS : S9F7 Sent automatically
- 38: SECS/HSMS : S9F11 Sent automatically
- 39: SECS/HSMS : S9F9 Sent automatically
- 40: HSMS : Connect failed
- 51: SECS/HSMS : List format display of received SECS message
- 52: SECS/HSMS : List format display of send SECS message
- 53: SECS/HSMS : Hexadecimal format display of received SECS message
- 54: SECS/HSMS : Hexadecimal format display of send SECS message

(Note 4) Call of user function at the time of communication trace output by this setting is valid only when the SECS communication library is not executed by tdsc but executed by thread operation.

That is, it is valid only when $(mode \& 0x03) == 0x02$ at `_TDSCommOpen()`.

(Note 5) User trace output function is executed as part of a series of communication processing in SECS/HSMS communication processing thread. That is, during processing of the user trace output function, communication processing can not be performed. Thus, the user trace output function should terminate as soon as possible.

For example, consider a case where an AP (like using an MFC) sends a message regarding window display and waits for the completion of the message sending result. The communication processing unit of TDS calls the user trace output function, and the user trace output function outputs the trace message to the display window instance. Therefore, the control related to the window processing does not return to 'Main Loop' at that time, so it waits without being able to output a message. As a result, a phenomenon occurs in which the sending process is not completed, and completion of sending result is awaited. In this case, it is necessary to avoid deadlock by setting the sending result to be received by callback, threading user trace output, or the like.

(2) SECS/HSMS communication : Close processing

Ends open processing of SECS / HSMS communication processing.

```
int _TDSCommClose(      // Return value                      (*1)
int      fd,           // in  : Control identifier      (Obtained by _TDSCommOpen())
int      mode)         // in  : Processing mode        (Must be =0)
```

(*1) Return value

```
> 0: <reserved>
== 0: OK
< 0: NG .. Error code
    -EBADF   : The specified 'fd' has not been _TDSCommOpen() processed
    Other    : Other error
```

(3) SECS/HSMS communication : Receive processing

Message processed by SECS/HSMS communication module is acquired from receive queue.
It also receives reports on the following events that occur in the process of performing SECS/HSMS communication processing.

Connect / Shutdown / Various timeouts occur

```
int  _TDSCommRecv(          // Return value                                (*1)
int   fd,                  // in  : Control identifier          (Obtained by _TDSCommOpen())
int   mode,                // in  : Processing mode            (*2)
int   *devid,              // out : Received message Device ID
int   *sf,                 // out : Received message SF-Code   (*3)
int   *xid,                // out : Received message Transaction ID (*4)
void  *msg,                // out : Received message Body      (Data part)
int   len,                 // in  : Receiveable message length (Number of bytes)
TDSECSHead *hd)            // out : Received message Header    (Refer to 2.2 (1) (Note 1))
```

(Note 1) If any out parameter is specified as 0, it is not stored.

(*1) Return value

>= 0: OK .. Received message (data part) effective data length (number of bytes)

< 0: NG .. Error code

```
-EBADF      : The specified 'fd' has not been _TDSCommOpen() processed
-EBUSY      : Using incoming callback function
-ENOMEM     : Memory allocation error for receiving
-ENOEXEC    : Checksum error
-EPROTO     : Protocol error

-ENODEV     : (S9F1) Undefined device
-E2BIG      : (S9F11) Data size over
-EFAULT     : (S9F7) Receive secondary messages not waiting to be received

-E_CONNECT  : Connected (Indicates that connected state has been entered)
-E_SELECT   : Selected (Indicates that selected state has been entered)
-E_DESELECT : Deselected (Indicates that deselected state has been entered)
-E_NOTCONNECT : Shutdowned (Indicates that shutdowned state has been entered)

-E_CONN_ERR : Connect failed in HSMS communication
-E_REJECT   : Reject request received in receive operation in HSMS communication
-E_ILLBLOCK : Received invalid block number in receive operation in SECS-1
-E_RETRYOVER : Retry count over occurred in receive operation in SECS-1

-E_T1TIMEDOUT : T1 Timeout Occure
-E_T2TIMEDOUT : T2 Timeout Occure
-E_T3TIMEDOUT : T3 Timeout Occure (devid, sf, xid are for T3 Timeout target mess.)
-E_T4TIMEDOUT : T4 Timeout Occure
-E_T5TIMEDOUT : T5 Timeout Occure
-E_T6TIMEDOUT : T6 Timeout Occure (Include LinkTest timeout)
-E_T7TIMEDOUT : T7 Timeout Occure
-E_T8TIMEDOUT : T8 Timeout Occure
-E_NOMSSGTO  : Message not received continues for more than specified time

-E_NODATA    : There is no received message

Other       : Other error
```

< Continue to next page >

< Continue from previous page >

(Note 2) -ENODEV, -EFAULT, -E2BIG store acquired data in 'msg', 'hd'. For -E_REJECT, store reason code in lower 8 bits of sf and transaction ID of Reject target message in 'xid'.

(Note 3) -E_NOTCONNECT may occur multiple times depending on the timing of TCP/IP connection disconnection and occurrence of communication error.

(*2) mode : Processing mode

1

+-----+

+--- Receive data queue processing mode

= 0: Process all messages without receiving following check

1: Valid only when receiving message from receiving data queue
is the last time receive processing was performed

(*3) sf : SF-Code

FE 8 7 0

+-----+

|+-----+

| | +----- F Code (0 - 255)

| +----- S Code (0 - 127)

+----- Wait bit (0:No Wait 1:With Wait)

(*4) xid : Transaction ID

G F 0

<-----+

<---+ +-----+

| +----- Transaction ID (1 - 65535)

+----- Source ID (Usually: 0:User data !=0:Control data)

(4) SECS/HSMS communication : Send processing

Send a message transmission request to SECS/HSMS communication module.

Also makes requests for "connect", "disconnect" and etc. related to SECS/HSMS communication.

```
int  _TDSCommSend(          // Return value                                (*1)
int    fd,                  // in  : Control identifier          (Obtained by _TDSCommOpen())
int    mode,                // in  : Processing mode              (*2)
int    devid,               // in  : Send message Device ID      (*3)
int    sf,                  // in  : Send message SF-Code        (*4)
int    xid,                 // in  : Send message Transaction ID  (*5)
void    msg,                // in  : Send message Body           Data part)
int    len,                 // in  : Send message length         (Number of bytes)
TDSECSHead *hd)             // out : Send message Header         (Valid only when (mode&0xf00)==0)
                                When=0, it does not store.      (Refer to 2.2 (1)(note 1))
```

(*1) Return value

>= 0: OK .. When sending a packet, Transaction ID of the send message
See description of 'xid' in _TDSCommRecv()

(Note 1) Suppose that same value is specified for SRCID0 and SRCID1 in .ini
file. When using sending completion call back function, and (mode&0x02)
==0, it may be different from the Transaction ID given at time of actual
sending.

< 0: NG .. Error code

```
-EBADF      : The specified 'fd' has not been _TDSCommOpen() processed
-EINVAL     : Illegal parameter
-EILSEQ     : It is not time to send specified message
              (Send SECS message without selecting etc.)
-ENOMEM     : Memory allocation error for send
-ENODEV     : Specified 'devid' is not Device ID to process
-ENOEXEC    : Requested transmission for SessionID not selected in transmission
              in HSMS-GS mode.
-ETIMEDOUT  : Transmission time out (time out occurred in sending result
              receiving)
-ENOSPC     : Transaction management table is full
              Send queue write area is full with unprocessed messages

-E_CONN_ERR : Connect failed in HSMS communication
-E_RETRYOVER : Retry count over occurred in sending operation in SECS-1
              communication

-E_T2TIMEDOUT : T2 Timeout occurred
-E_T4TIMEDOUT : T4 Timeout occurred

Other       : Other error
```


(*2) mode : Processing mode

BA98 1

+-----+-----+-----+-----+

|||| +--- In same message judgment at the time of sending result
acquisition ...

|||| = 0: Wait for result of sending some message

|||| 1: Strictly wait for the result for sent messages

||||

|||| (Note 1) Consideration of using a callback function specified in
_TDSCCommOpen() for processing related to received message, etc.

|||| when operating in different thread from sending processing.

|||| Sending of primary message and sending of secondary message

|||| regarding received primary message can occur simultaneously.

|||| In that case, when sending result is obtained each time

|||| (without using callback function), order of obtaining sending

|||| result corresponding to sending message becomes unequal.

|||| Therefore, there are cases where it is not possible to obtain
correct sending result corresponding to sending message.

|||| If this bit is set to 1 in an environment where such
condition occurs, it may not be acquired correctly and it may
become await. In such case, it is necessary to set this bit to
0 or exclusive control so that different threads do not
process CommSend() simultaneously.

|||| --> In Ver.16.030 or later, exclusive control is performed
internally, so exclusive control by AP in this case is
unnecessary.

||||

|||| (Note 2) When=0 is specified, Transaction ID of the return value
may be different from that actually assigned.

|||| Refer to (Note 1) of the '(*)1) return value'.

||||

++++----- Message type

= 0 : SECS/HSMS Message

1 : Connect request

2 : Select request

3 - 6 : <reserved>

7 : Reject request

8 : Deselect request

9 : Separate and Shutdown request

10 - 15 : <reserved>

(*3) devid : Device ID

= -1: Use first Device ID defined in .ini

= 0: If 0x0000 is not included in the Device ID defined in ini, use first Device ID
defined. If it is included, use Devid=0x0000 as it is.

> 0: Use specified Device ID as it is.

(*4) sf : SF-Code

Refer to description of 'sf' in _TDSCCommRecv()

In case of Reject request, set the reason code in F-Code part (lower 8 bits)

(*5) xid : Transaction ID

Refer to description of 'xid' in _TDSCCommRecv()

This parameter is significant in following cases

+ When sending secondary message : Specify same xid as received primary message.

+ When sending Reject message : Specify same xid as reject target message.

(5) SECS/HSMS communication : Illegal message receiving result sending process

Sends SxF0 and S9Fx messages, which are sent when an unauthorized message is received.
For the message header used when sending S9Fx messages, specify the message header received with `_TDSCommRecv()` as it is.

```
int _TDSCommSendError( // Return value (Refer to (4) (*1))
int     fd,           // in  : Control identifier (Obtained by _TDSCommOpen())
int     mode,         // in  : Processing mode (*1)
int     devid,        // in  : Sending message Device ID (Refer to (4) (*3))
int     sf,           // in  : Sending message SF-Code (Refer to (4) (*4))
TDSECSHead *rhd,      // in  : Target bad header of S9Fx (Only significant for S9Fx)
TDSECSHead *shd,      // out : Sending message Header (Refer to 2.2 (1))
void     *sdt)         // out : Sending message Body
                        // Make the area of sufficient size (16 bytes or more).
```

(*1) mode : Processing mode

1

--+-----+-----+-----+

+--- In same message judgment at the time of transmission result acquisition ...

= 0: Wait for the result of sending some message

1: Strictly wait for the result for sent messages

(Note 1) 'shd' and 'sdt' are not stored when =0 is specified.

(6) Connection status check

Get current connection status.

```

int _TDSCommStatus(      // Return value
                        // < 0: <To be determined>
                        // = 0: Disconnected
                        // 1: Connecting, Not Selected state
                        // 2: Connecting, During Select processing
                        // 3: Connecting, Selected state
                        // 4: Connecting, During Diselect processing
                        // > 4: <To be determined>
                        // (Note 0) That is, ==3 is SECS connection status, !=3 is SECS
                        // disconnection status
int      fd,             // in : Control identifier          (Obtained by _TDSCommOpen())
int      mode)           // in : Processing mode              (Must be =0)

```

(Note 1) Current connection status is confirmed by receiving status with _TDSCommRecv().
Therefore, if you do not use receive Callback function, use this function in combination with the Call of _TDSCommRecv().

(7) Checking connection status for each session ID (HSMS-GS)

Get current connection status for each session ID in operation in HSMS-GS mode.

```

int _TDSCommSelectStatus(// Return value
                        // < 0: NG
                        //      -ENODEV : Designated Session ID is not for processing
                        // = 0: Disconnected, Not Selected state
                        // 1: Connecting, Not Selected state      <reserved>
                        // 2: Connecting, During Select processing <reserved>
                        // 3: Connecting, Selected state
                        // 4: Connecting, During Diselect processing <reserved>
                        // > 4: <To be determined>
                        // (Note 0) That is, ==3 is SECS connection status, !=3 is SECS
                        // disconnection status
int      fd,             // in : Control identifier          (Obtained by _TDSCommOpen())
int      mode,           // in : Processing mode              (Must be =0)
int      devid)          // in : Session ID (Device ID)

```

(Note 1) Refer to (Note 1) in _TDSCommStatus()

(8) Additional output of user comment messages to communication trace file

Output any comment message to communication trace file output by this library.

```
int _TDSCommUserComment( // Return value                <reserved>
int      fd,             // in  : Control identifier          (Obtained by _TDSCommOpen())
int      mode,           // in  : Processing mode              (Must be =0)
char     *comm)          // in  : Comment message to output
```

(Note 1) If this function is called continuously for a short time, the communication control queue will be full, and the possibility of -28 error occurring when calling _TDSCommXxxx() increases. In that case, it is necessary to adjust the Call interval of this function or to increase QUEREC parameter of .ini.

3.2 SECS/HSMS Message communication function (abbreviated API)

In the simple API, Active side of HSMS does not need to perform Connect and Select processing. Connect and Select processing in this case is performed automatically. The AP can monitor connection status and know current status by `_TDSUDrvStatus()`. In addition, it is possible to narrow down the events acquired by `_TDSUDrvRecv()` to only necessary ones.

(1) SECS/HSMS communication : Open processing

Analyzes parameters of the specified section in the specified .ini file, acquires SECS/HSMS communication conditions, and starts SECS/HSMS communication process.

Refer to the description in 3.1 (1).

```
int _TDSUDrvOpen(          // Return value                (3.1 (1) (*1))
int    mode,              // in  : Processing mode          (3.1 (1) (*2))
char   *ini,              // in  : .ini file path name      (3.1 (1) (*3))
char   *sect,             // in  : .ini file section name to use (3.1 (1) (*4))
int     mask)             // in  : Mask of acquisition target event with _TDSUDrvRecv() (*1)
```

(*2) mask : Mask of acquisition target event with `_TDSUDrvRecv()`

The corresponding event is received by `_TDSUDrvRecv()` by setting bit shown below to =1. If mask=0, it is treated as mask=0x0049.

V	P0	N	HG	F	DC	BA98	7654	3210	
									SECS/HSMS Send/Receive message
									SECS T1 Timeout occur
									SECS T2 Timeout occur
									SECS/HSMS T3 Timeout occur
									SECS T4 Timeout occur
									HSMS T5 Timeout occur
									HSMS T6 Timeout occur
									(Include LinkTest T0)
									HSMS T7 Timeout occur
									HSMS T8 Timeout occur
									SECS Message transmission retry over
									Timeout occurs due to not receiving for
									specified time (TORECV) or more
									Receive undefined device ID message
									Received message data length too large
									SECS Message block number is abnormal
									HSMS Connect failure
									HSMS It became Connected state
									HSMS It became Select state
									HSMS Received Reject message
									HSMS It became Deselect state
									HSMS It became not connected state
									Another error has occurred

(2) SECS/HSMS communication : Close processing

Ends open processing of SECS / HSMS communication processing.

```
int _TDSUDrvClose(      // Return value                      (3.1 (2) (*1))
int    fd,              // in  : Control identifier      (Obtained by _TDSUDrvOpen())
                        //          Control identifier
int    mode)            // in  : Processing mode        (Must be =0)
```

(Note 1) In the case of HSMS Active connection, if Passive side has already stopped, it may take more than T5T0 time to finish Close processing.

(3) SECS/HSMS communication : Receive processing

Message processed by SECS/HSMS communication module is acquired from receive queue.
Also, various events specified by mask at _TDSUDrvOpen() that occurs in the process of executing SECS/HSMS communication processing are acquired.

```
int _TDSUDrvRecv(      // Return value                      (3.1 (3) (*1))
int    fd,              // in  : Control identifier      (Obtained by _TDSUDrvOpen())
int    mode,            // in  : Processing mode        (Must be =0)
int    *devid,          // out : Received message Device ID
int    *sf,              // out : Received message SF-Code      (3.1 (3) (*3))
int    *xid,            // out : Received message Transaction ID (3.1 (3) (*4))
void    *msg,           // out : Received message Body        (Data part)
int    len,             // in  : Receiveable message length  (Number of bytes)
TDSECSHead *hd)         // out : Received message Header      (Refer to 2.2 (1)(Note 1))
```

(Note 1) If any out parameter is specified as 0, it is not stored.

(4) SECS/HSMS communication : Send processing

Send a message transmission request to SECS/HSMS communication module.

```
int _TDSUDrvSend(      // Return value                      (3.1 (4) (*1))
int    fd,              // in  : Control identifier      (Obtained by _TDSUDrvOpen())
int    mode,            // in  : Processing mode        (3.1 (4) (*2))
int    devid,           // in  : Send message Device ID    (3.1 (4) (*3))
int    sf,              // in  : Send message SF-Code      (3.1 (4) (*4))
int    xid,             // in  : Send message Transaction ID (3.1 (4) (*5))
void    msg,            // in  : Send message Body        (Data part)
int    len,             // in  : Send message length      (Number of bytes)
TDSECSHead *hd)         // out : Send message Header
                        //          When=0, it does not store. (Refer to 2.2 (1)(note 1))
```

(5) SECS/HSMS communication : Illegal message receiving result sending process

Sends SxF0 and S9Fx messages, which are sent when an unauthorized message is received.
For the message header used when sending S9Fx messages, specify the message header received with `_TDSUDrvRecv()` as it is.

```
int _TDSUDrvSendError( // Return value (3.1 (4) (*1))
int     fd,           // in  : Control identifier (Obtained by _TDSUDrvOpen())
int     mode,         // in  : Processing mode (3.1 (5) (*1))
int     devid,        // in  : Sending message Device ID
int     sf,           // in  : Sending message SF-Code (3.1 (4) (*4))
TDSECSHead *rhd,      // in  : Target bad header of S9Fx (Only significant for S9Fx)
TDSECSHead *shd,      // out : Sending message Header (Refer to 2.2 (1) (Note 1))
void     *sdt)        // out : Sending message Body
// Make the area of sufficient size (16 bytes or more).
```

(Note 1) 'shd' and 'sdt' are not stored when =0 is specified.

(6) Connection status check

Get current connection status.

```
int _TDSUDrvStatus( // Return value (Refer to 3.1 (6) Return value)
int     fd,         // in  : Control identifier (Obtained by _TDSUDrvOpen())
int     mode)       // in  : Processing mode (Must be =0)
```

(Note 1) Current connection status is confirmed by receiving status with `_TDSUDrvRecv()`.
Therefore, use this function in combination with the Call of `_TDSUDrvRecv()`.

(7) Checking connection status for each session ID (HSMS-GS)

Get current connection status for each session ID in operation in HSMS-GS mode.

```
int _TDSUDrvSelectStatus( // Return value (Refer to 3.1 (6) Return value)
int     fd,              // in  : Control identifier (Obtained by _TDSUDrvOpen())
int     mode,            // in  : Processing mode (Must be =0)
int     devid)           // in  : Session ID (Device ID)
```

(Note 1) Refer to (Note 1) in `_TDSUDrvStatus()`

(8) Additional output of user comment messages to communication trace file

Output any comment message to communication trace file output by this library.

```
int _TDSUDrvUserComment( // Return value <reserved>
int     fd,              // in  : Control identifier (Obtained by _TDSUDrvOpen())
int     mode,            // in  : Processing mode (Must be =0)
char     *comm)          // in  : Comment message to output
```

3.3 SECS Message construction, analysis function

<Caution>

It is not possible to simultaneously execute series of construction and analysis processes in different threads using the same "Message identifier". (The same control identifier is not thread safe.) Be careful when using this function in multiple threads.

Also, when processing data item names using the SECS message definition file, "SECS communication control identifier" specified with MssgInit() and MssgFind() can not be used simultaneously by different threads.

(1) SECS message construction

Construct a SECS message in the specified area.

First, initialize the storage area of message to be built with _TDSMssgInit(), and add message items to be built sequentially from top of message with _TDSMssgBuild() (or _TDSMssgBuildL()). Finally, create SECS message in specified area by ending message construction with _TDSMssgEnd().

(Note 1) Maximum number of layers in list structure is 32.

(a) Initialize

```
int _TDSMssgInit(          // Return value                      (*1)
int    mode,              // in  : Processing mode
                        //      B
                        // +---+---+---+---+
                        //      +-- Type of parameter 'fdc'
                        //      =0:SECS communication control identifier
                        //      1:Maximum SECS message byte size to process
void    *msg,             // out : SECS Message storage area
int    len,              // in  : SECS Message storage area byte size
int    fdc)              // in  : SECS communication control identifier (Already opened)
                        //      or Maximum SECS message byte size to process.
// If you do not perform SECS communication and only perform message-related
// processing, it is better to set mode|=0x8000 at _CommOpen().
// (Note) The information currently required is only the maximum SECS message byte
// size to be processed, so if you do not use _CommOpen(), you can specify
// (mode&0x0800)!=0 and specify the SECS message byte size in fdc.
// In this case, SECS Message can be constructed without .ini file
```

(*1) Return value

```
> 0: OK      : Message identifier used for subsequent processing
== 0: <reserved>
< 0: NG      : Error code
      -ENOMEM : Internal processing memory allocation error
      -ENOSPC : Internal processing table full
      Other   : Other error
```

(b) End processing

```
int _TDSMssgEnd(          // Return value                      (*1)
int    md,               // in  : Message identifier          (Obtained by _TDSMssgInit())
int    mode,             // in  : Processing mode              (Must be =0)
void    *msg)            // i/o : SECS Message storage area
```

(*1) Return value

```
>= 0: OK      : Byte length of constructed SECS message
< 0: NG      : Error code
      -EBADF   : The specified 'md' has not been _TDSMssgInit() processed
      Other    : Other error
```


(c-1) Add data item (usual format)

```

int  _TDSmsgBuild(      // Return value                                (*1)
int      md,            // in  : Message identifier      (Obtained by _TDSmsgInit())
int      mode,          // in  : Processing mode
                        //      bit#14: Character string conversion processing in case of
                        //              (form&0xff)==022
                        //              =0: Do nothing
                        //              1: Convert own string code to the specified code.
void     *msg,          // i/o : SECS Message storage area
int      form,          // in  : Additional item Format code                                (*2)
int      nop,          // in  : Additional item number of parameter
void     *item)         // in  : Additional item parameter storage area

```

(*1) Return value

```

> 0: <reserved>
== 0: OK
< 0: NG      : Error code
    -EBADF   : The specified 'md' has not been _TDSmsgInit() processed
    -EINVAL  : Specified additional item format is illegal
    -ENOSPC  : Insufficient message area
    Other    : Other error

```

(*2) form : Additional item format code (specified in octal)

```

= 000: List                (TDS_LST)
  010: 1Byte Binary        (TDS_BIN)
  011: 1Byte Logical value (TDS_LOG)
  020: 1Byte Ascii Text string (TDS_ASC)
  021: 1Byte JIS-8 Text string (TDS_JIS)
  022: Multi-byte Text string (TDS_KAN)
  030: 8Bytes Signed Integer (TDS_S8)
  031: 1Byte Signed Integer (TDS_S1)
  032: 2Bytes Signed Integer (TDS_S2)
  034: 4Bytes Signed Integer (TDS_S4)
  040: 8Bytes IEEE754 Floating point (TDS_F8)
  044: 4Bytes IEEE754 Floating point (TDS_F4)
  050: 8Bytes Unsigned Integer (TDS_U8)
  051: 1Byte Unsigned Integer (TDS_U1)
  052: 2Bytes Unsigned Integer (TDS_U2)
  054: 4Bytes Unsigned Integer (TDS_U4)

```

(Note 1) Be aware of when specifying half-width katakana in form=021 (JIS-8 text string), depending on processing system, the representation format of half-width katakana is not limited to 1 byte (2 bytes for EUC-JP, 3 bytes for UTF-8).

(Note 2) For form=022 (Multi-byte Text string), specify the following if you do not use the processing standard kanji code.

```

UTF-8      : form = 0x00020000 + 022
Shift-JIS  : form = 0x00080000 + 022
EUC-JP     : form = 0x00090000 + 022

```

The processing standard kanji code is as follows

Win32	: Shift-JIS	Linux	: UTF-8
SunOS	: UTF-8	FreeBSD	: UTF-8
HP-UX	: Shift-JIS	MacOS X	: UTF-8

(c-2) Add data item (string form)

```

int _TDSMssgBuildL(    // Return value                                (Refer to (c-1))
int     md,           // in  : Message identifier      (Obtained by _TDSMssgInit())
int     mode,         // in  : Processing mode
                        // E          654
                        // +---+---+---+---+
                        // |          +++
                        // |          +-- List input format
                        // |          0:TDS Format          2 :SML Format
                        // |          1:<reserved>          3-7:<reserved>
                        // +-- <reserved>
void     *msg,         // i/o : SECS Message storage area
char     *str)         // in  : SECS data items in the given string format

```

(Note 1) It does not correspond to all data values of the SECS data item that are output over multiple lines with (mode&0x04)!=0 in _TDSMssgNextL().
Also, it does not support "... " notation due to excessive data.

(2) SECS Message analysis

Parse the SECS message stored in the specified area.
 First, `_TDSMssgFind()` performs initialization processing on the storage area of message to be analyzed, and `_TDSMssgNext()` (or `_TDSMssgNextL()`) sequentially acquires message items from beginning of the message. When acquisition is complete (or not), message analysis is terminated with `_TDSMssgExit()`.

(Note 1) Maximum number of layers in list structure is 32

(a) Initialize

```

int _TDSMssgFind(          // Return value                      (*1)
int      mode,             // in  : Processing mode
                                // If you continue to use _TDSMssgNextL() and perform item name
                                // display using message definition file, set the following bits.
                                // F DC B
                                // +---+---+---+---+
                                // | || +-- Type of parameter 'fdc'
                                // | ||      =0:SECS communication control identifier
                                // | ||      1:Maximum SECS message byte size to process
                                // | ++-- Origin node of analysis target message
                                // |      (Only significant when using message definitions)
                                // |      =0:Both side      2:My side message
                                // |      1:<reserved>      3:Oposite side message
                                // +----- Whether to use message definition
                                //          =0:Not use      1:Use
                                //          (Note) Set =0 except when _TDSMssgNextL() is used to
                                //          specify item name display.
void      *msg,             // in  : SECS Message storage area
int      len,              // in  : SECS message byte size
int      fdc,              // in  : SECS communication control identifier (Refer to the
                                // description of fdc in 3.3 (1) (a))
                                // or Maximum SECS message byte size to process.
TDSECSHead *hd,            // in  : SECS Message Header
                                // (Not required if you don't use message definition. Set =0)
char      *mname)          // out : SECS Message name                      (Do not store if =0)

```

(*1) Return value

```

> 0: OK      : Message identifier used for subsequent processing
== 0: <reserved>
< 0: NG      : Error code
      -ENOMEM : Internal processing memory allocation error
      -ENOSPC : Internal processing table full
      Other   : Other error

```

(Note 1) hd, mname

Only significant if $(mode \& 0x8000) \neq 0$. Otherwise set =0.

(Note 2) -----+
 | When acquiring message name (mname) with this function or acquiring item name of message |
 | with `_TDSMssgNextL()` by referring to message definition file, you must give fdc to this |
 | function. The same fdc can not be specified when processing by this function in different |
 | threads simultaneously. |
 +-----+

(b) End processing

```

int  _TDSMssgExit(      // Return value                                     (*1)
int    md,              // in  : Message identifier         (Obtained by _TDSMssgFind())
int    mode,            // in  : Processing mode           (Must be =0)
void    *msg)           // in  : SECS Message storage area

```

(*1) Return value

```

> 0: <reserved>
== 0: OK
< 0: NG          : Error code
    -EBADF       : The specified 'md' has not been _TDSMssgFind() processed
    Other        : Other error

```

(c-1) Data item acquisition (usually format)

```

int  _TDSMssgNext(      // Return value                                     (*1)
int    md,              // in  : Message identifier         (Obtained by _TDSMssgFind())
int    mode,            // in  : Processing mode
                                //      bit#14: Character string conversion processing in case of
                                //      (form&0xff)==022
                                //      =0: Do nothing
                                //      1: Convert the specified code to own string code.
void    *msg,           // i   : SECS Message storage area
int    *form,           // out : Item Format code           (Refer to form in _TDSMssgBuild())
int    *parl,           // out : Byte size occupied by one item (*2)
int    *noi,            // out : Number of items parameters (*2)
void    *item,          // out : Item parameter value storage area (*2)
int    xlen)           // in  : Byte size of item parameter value storage area

```

(*1) Return value

```

> 0: <reserved>
== 0: OK
< 0: NG          : Error code
    -EBADF       : The specified 'md' has not been _TDSMssgFind() processed
    -EINVAL      : Data structure error (Item code error)
    -EILSEQ      : Data structure error (Length Byte =0)
    -E2BIG       : Data structure error (Item length exceeds overall length)
    -ENOSPC      : Insufficient message item storage area
    -E_NODATA    : Message analysis end
    Other        : Other error

```

```

(*2) parl  : Byte size occupied by one item
    noi    : Number of items
    item   : Item storage area

```

'noi' pieces of data of the form indicated by 'form' are stored consecutively in 'item'. Therefore, 'parl'*'noi' bytes are valid data in item.

(c-2) Data item acquisition (string format)

```

int  _TDSMssgNextL(      // Return value                (Refer to (c-1) and lower (Note 1))
int    md,              // in  : Message identifier          (Obtained by _TDSMssgFind())
int    mode,            // in  : Processing mode
                        // EDC      7654  2
                        // +----+----+----+----+
                        // |||      |+++ |
                        // |||      | |  +-- Item data display format (other than TEXT)
                        // |||      | |      0:Display only one line
                        // |||      | |      1:Display all data in multiple lines
                        // |||      | +-- List output format
                        // |||      |      0:TDS Format          2 :SML Format
                        // |||      |      1:<reserved>         3-7:<reserved>
                        // |||      +---- Item name display          (0:No 1:Yes)
                        // |||      Valid only when valid message definition is
                        // |||      specified
                        // |++-- Origin node of analysis target message
                        // |      (Only significant when using message definitions)
                        // |      =0:Both side          2:My side message
                        // |      1:<reserved>          3:Oposite side message
                        // +---- <reserved>
void    *msg,            // i/o : SECS Message storage area
int     *form,           // out : Item format code            (Refer 'form' in _TDSMssgBuild())
int     *noi,            // out : Number of items
char    *str,            // out : SECS data item of predetermined string format (256bytes or
                        //      more required)
int     xlen)            // in  : Byte size of 'str'

```

(Note 1) If SECS data item format stored in str is same format as SML format communication trace, set bits 4 to 6 of mode to 2.

In this case, as return value, value=>0 is returned, and the value indicates number of list layers. The closing parenthesis ">" indicating the end of hierarchy needs to be displayed on the AP side based on return value.

(Refer to the sample code of each language included in "Sample0/" of this package.)

- + Note that in the case of "L,0", there is no opportunity to increase the hierarchy. Close parenthesis ">" corresponding to "<L,0" is processed when (form==000 && noi==0) is acquired.
- + For display method of closing parenthesis ">", refer to the code DispSECSMssg() in sample program "Sample0/SubFunction.h".

(Note 2) To specify valid message definition file and expand data item name, set 7th bit of mode =1. The format of message definition file is as specified by bits 8 and 9 of TRCTTYPE in .ini file.

(Note 3) When processing by this function in different threads simultaneously, md and fdc

~~~~~  
specified at \_TDSMssgFind() can not be specified the same.  
~~~~~

3.4 SECS message construction and analysis function using SML format message structure definition

(0) Message structure definition information acquisition and release

Read message structure definition file and acquire definition information in message control block in order to perform message construction and analysis shown in (1) and (2).

The format of parseable message structure definition file is as follows.

+ SML Format

Although it is possible to repeatedly execute (1) and (2) using the message control block initialized by initialization function, it is not possible to simultaneously execute (1) and (2). When executing simultaneously, different message control blocks need to be used.

A table area is allocated by malloc() in message control block initialized by initialization function. Therefore, when processing is completed, _TDSMDMssgTerminate() must be called to release the table area.

<Caution 1>

This function does not check validity of contents of message structure definition file. Therefore, be aware that if you specify message structure definition file for which abnormal settings exist (especially with regard to list structure), serious problems (abnormal termination etc.) may occur in the subsequent operations.

<Caution 2>

Processing using message structure definition file (construction of communication messages, analysis, communication trace output) requires more resources than similar processing when not used. Therefore, I recommend if you need faster communication response speed, use the function that constructs, analyzes and performs SECS communication message without using message structure definition file shown in Section 3.3 instead of the function shown in this section.

<Caution 3>

Refer to Appendix A for the format of message structure definition file.

(a) Message structure definition information acquisition and initialization

<Caution>

It is not possible to simultaneously execute series of construction and analysis processes in different threads using same "Message identifier" and the same "SECS process control identifier". (The same Message identifier and SECS control identifier are not thread safe.) If you use this function in multiple threads, be careful when using the following initialization function (_TDSMDMssgInitialize() with (mode&0x8000)==0). Also, if (mode&0x8000)!=0, you must call End() after calling Init(), and Exit() after calling Find().

```
int _TDSMDMssgInitialize(// Return value (*1)
int      mode,          // in : Processing mode
                        // E 10
                        // +---+---+---+---+
                        // |          +--- 0:SML Format 2:<reserved>
                        // |          1:<reserved> 3:<reserved>
                        // |          (Note 1) This specification is invalid
                        // |          when path == 0
                        // +--- When executing the functions shown in (1) and (2), do you
                        //      lock the thread ? (0:No 1:Yes)
int      fd,            // in : SECS communication control identifier (Already opened)
                        //      Refer to the description of 'fd' in 3.3 (1)(a)
                        //      (Always specify control identifier, not maximum message
                        //      size. It may be the 'fd' by (mode&0x8000)!=0 of
                        //      _TDSCommOpen())
char     *path)         // in : Message structure definition file path name
                        //      (Note 2) If path==0 or path[0]!='\0', use the name
                        //      specified in MDMSSG of .ini file. In that case, format
                        //      of definition file depends on the (TRCTTYPE&0x0300)
                        //      value.
```

(*1) Return value

```
> 0: OK      : Message identifier used for subsequent processing
== 0: <reserved>
< 0: NG      : Error code
   Other     : Other error
   -941      : Insufficient space in data item definition table
   -942      : Insufficient space in message definition table
   -943      : Insufficient space in table for storing items for each message
   -944      : Insufficient data storage area to set/check items per message
```

(Note 3) In case of above error, increase following parameters set in .ini file.

```
-941 : MDMXITEM : Maximum number of total data item
-942 : MDMXMSSG : Maximum number of messages to defin
-943 : MDMXMITEM : Maximum number of total items
-944 : MDMXPOOL : Message definition setting data storage area size
```

(b) Message structure definition information release

```
void _TDSMDMssgTerminate(
int      md,          // in : Message identifier (Obtained by _TDSMDMssgInitialize())
int      mode)        // in : Processing mode
                        // E
                        // +---+---+---+---+
                        // +--- Force release of lock resource when (mode&0x4000)!=0 is
                        //      specified in Initialize() (0:No 1:Yes)
```

(1) SECS message construction

Construct SECS message with specified name in specified area according to contents of message structure definition file.

First, initialize message storage area constructed with `_TDSMDMssgInit()`. Next, in `_TDSMDMssgBuild()`, set parameter value of each data item included in message. Finally, create SECS message in specified area by ending message construction with `_TDSMDMssgEnd()`.

The setting of the number of lists for an indefinite number of list items must be performed prior to setting of general item values to determine number of all list items. Also, determination of the number of list items must be made from higher-level list items.

The specification order of parameter values for specified data item by `_TDSMDMssgBuild()` does not necessarily have to be appearance order of data item (except for an indefinite number of list items).

(Note 1) If $(mode \& 0x4000) \neq 0$ in `_TDSMDMssInitialize()`, deadlock may occur if `MssgInit()` and `MssgEnd()` do not always correspond. That is, if `MssgInit()` succeeds, `MssgEnd()` must be called. If you do not call `MssgInit()`, you must not call `MssgEnd()`.

(a) Initialize

```
int  _TDSMDMssgInit(      // Return value                      (Refer to (0) (a))
int    md,                // in  : Message identifier  (Obtained by _TDSMDMssgInitialize())
int    mode,              // in  : Processing mode                      (*1)
void    *msg,             // out : SECS Message storage area
int     len,              // in  : SECS Message storage area byte size
char    *mname)           // in  : Message name in message structure definition file
```

(*1) mode : Processing mode

```
E
+-----+-----+-----+-----+
+---- Disable lock control
      = 0: According to specification at _TDSMDMssgInitialize()
      = 1: Do not perform lock control.
```

(b) End processing

```
int  _TDSMDMssgEnd(      // Return value                      (*1)
int    md,                // in  : Message identifier  (Obtained by _TDSMDMssgInitialize())
int    mode,              // in  : Processing mode                      (Refer to (a) (*1))
                        //      (Note) (mode&0x4000) must be same as the corresponding
                        //      _TDSMDMssgInit().
void    *msg)             // i/o : SECS Message storage area
```

(*1) Return value

```
>= 0: OK      : Byte length of constructed SECS message
< 0: NG       : Error code                      (Refer to 3.3(1) (c-1) (*2))
```


(c) Data item parameter value setting

```

int _TDSMDMssgBuild(    // Return value                                (Refer to (0) (a))
int     md,             // in  : Message identifier      (Obtained by _TDSMDMssgInitialize())
int     mode,           // in  : Processing mode          (Must be =0)
void    *msg,           // i/o : SECS Message storage area
char    *iname,         // in  : Additional item name
int     nop,            // in  : Additional item number of parameters
void    *item)          // in  : Addttional item parameter strage area

```

(Note 1) Among the data items in message specified in _TDSMDMssgInit(), the parameter value of data item whose parameter value is not specified by Call of this function uses "default value" specified in message structure definition file. If 'Default' is not specified, set ' ' (blank) for string items and =0 for numeric items.

(Note 2) When specifying number of list items Specify in 'nop'. However, if you specify 'item' as pointer to int, and specify number of list items in '*item', use that.

(Note 3) If the number of parameters of specified data item is undefined (99..99 specified), the number specified by 'nop' is the number of parameters. In fixed case, it is the number specified in message structure definition file.

(Note 4) If the specified 'nop' is out of the number range specified in message structure definition file, set the parameter value as follows.

- + If more than range : Stop at the specified maximum number.
- + If less than range : Padding up to the specified minimum, ' ' (blank) for text, 0 for numerics.

(Note 5) If the data item is character string (format code is 020, 021, 022) If 'nop'=0 is specified, the value calculated from length of character string specified in item (strlen()) is used.

(Note 6) If there is an indefinite number of list items in the message, you must first determine its value. In addition, when there is a plurality of indefinite number of list items, it is necessary to determine in definition order in same hierarchy from list items of upper hierarchy.

(Note 7) When there are indefinite number of list items in message and the number of list structures is specified, the data item names below list items are expressed in the format of XXXX:99. Where XXXX is data item name and 99 is number of iterations starting from 1. When an indefinite number of list items extend over plurality of hierarchies, display indicating the number of repetitions is added each time the hierarchy is followed. That is, the second hierarchy has name such as XXX:99:99, and the third hierarchy has name such as XXX:99:99:99. However, in case of an indefinite number list of [0|x] format, item name is not changed (the number of repetitions is not assigned).

(Example 1) When data item name is SVID, the name in first repetition is SVID:1, the name in first repetition is SVID:2, and the name in 17th repetition is SVID:17.

(Example 2) If you set NOI1=3, NOI2:1=1, NOI2:2=0, NOI2:3=2 in message definition on left side of following, it will expand sequentially to the structure on right side. (">" In the list is omitted due to space limitations.)

	NOI1=3	NOI2:1=1, NOI2:2=0, NOI2:3=2
<L[N]NOI1	---> <L[3]NOI1	---> <L[3]NOI1
<L[2]	<L[2]	<L[2]
<A[10]ABC>	<A[10]ABC:1>	<A[10]ABC:1>
<L[N]NOI2	<L[N]NOI2:1	<L[1]NOI2:1
<B[1]XYZ>	<B[1]XYZ:1>	<B[1]XYZ:1:1>
	<L[2]	<L[2]
	<A[10]ABC:2>	<A[10]ABC:2>
	<L[N]NOI2:2	<L[0]NOI2:2
	<B[1]XYZ:2>	
	<L[2]	<L[2]
	<A[10]ABC:3>	<A[10]ABC:3>
	<L[N]NOI2:3	<L[2]NOI2:3
	<B[1]XYZ:3>	<B[1]XYZ:3:1>
		<B[1]XYZ:3:2>

(Example 3) If you set NOI1=2, NOI2:1=0, NOI2:2=2, NOI3:2=2 in message definition on left side of following, it will expand sequentially to the structure on right side. (">" In the list is omitted due to space limitations.)

	NOI1=2	NOI2:1=0, NOI2:2=2	NOI3:2=2
<L[N]NOI1	---> <L[2]NOI1	---> <L[2]NOI1	--> <L[2]NOI1
<L[0 2]NOI2	<L[0 2]NOI2:1	<L[0]NOI2:1	<L[0]NOI2:1
<A[10]ABC>	<A[10]ABC:1>		
<L[N]NOI3	<L[N]NOI3:1		
<B[1]XYZ>	<B[1]XYZ:1>		
	<L[0 2]NOI2:2	<L[2]NOI2:2	<L[2]NOI2:2
	<A[10]ABC:2>	<A[10]ABC:2>	<A[10]ABC:2>
	<L[N]NOI3:2	<L[N]NOI3:2	<L[2]NOI3:2
	<B[1]XYZ:2>	<B[1]XYZ:2>	<B[1]XYZ:2:1>
			<B[1]XYZ:2:2>

(Reference) -----+
 | The state of expansion of item names when the number of unfixed quantity lists is |
 | fixed can be confirmed by using "Tust Design Simple SECS Simulator (Preliminary |
 | version)" (tdlSSim) of our product. When expansion operation of message structure is |
 | performed and the number of items in indefinite number list is decided, the item names |
 | are expanded and displayed. |
 | For details of the product, download the product from our company HP and refer to the |
 | attached document. |
 +-----+

(2) SECS message analysis

Analyzes SECS message stored in specified area, determines which message in message structure definition file corresponds to, and outputs the corresponding message name. Also, parameter value of designated data item name is output.

In the processing procedure, first, `_TDSMDMssgFind()` initializes storage area of message to be analyzed, and obtains message name. Next, in `_TDSMDMssgNext()`, acquire parameter value of specified item name. Finally, complete message analysis with `_TDSMDMssgExit()`. Parsable messages can include an arbitrary number of list items.

(Note 1) If $(\text{mode} \& 0x4000) \neq 0$ in `_TDSMDMssgInitialize()`, deadlock may occur if `MssgFind()` and `MssgExit()` do not always correspond. That is, when `MssgFind()` succeeds, `MssgExit()` must be called. If you do not call `MssgFind()`, you must not call `MssgExit()`.

(a) Initialize

```
int  _TDSMDMssgFind(      // Return value                                (*1)
int    md,                // in  : Message identifier      (Obtained by _TDSMDMssgInitialize())
int    mode,              // in  : Processing mode                                (*2)
void    *msg,             // in  : SECS Message storage area
int    len,              // in  : SECS message byte size
int    sf,               // in  : SF-Code to be acquired
                        //      The specification method of SF-Code is same as '(*1)
                        //      Return value' (bit#15=0, #14-8:S-Code, #7-0:F-Code).
                        //      = 0: Do not limit the acquisition target SF-Code
char    *mname)          // out : Message name in message structure definition file
```

(*1) Return value

```
> 0: OK
    FE      8 7      0
    +-----+-----+
    | +---+ +---+ +---+
    | |           +----- F-Code of corresponding message
    | +----- S-Code of corresponding message
    +----- W-Bit Value
== 0: <reserved>
< 0: NG                : Error code
    Other then bellow : Other error
    = -ENOENT          : There is no corresponding message
    -941                : Insufficient space in data item definition table
    -942                : Insufficient space in message definition table
    -943                : Insufficient space in table for storing items for each message
    -944                : Insufficient data storage area to set/check items per message
```

(Note 1) For -941 to -944, refer to (1)(a)

(*2) mode : Processing mode

```
EDC
+-----+-----+-----+
| +--- Acquisition message limited
|   = 0: Not limited      2: Acquisition message is my side
|   1: <reserved>        3: Acquisition message is oposite side
+---- Disable lock control
    = 0: According to specification at _TDSMDMssgInitialize()
    1: Do not perform lock control
```

(b) End processing

```

int  _TDSMDMssgExit(      // Return value                                     (*1)
int      md,              // in  : Message identifier   (Obtained by _TDSMDMssgInitialize())
int      mode,            // in  : Processing mode      (Refer to (a) (*1))
                                //      (Note) The value of (mode&0x4000) must be same as
                                //      corresponding _TDSMDMssgFind().
void      *msg)           // in  : SECS Message storage area

```

(*1) Return value

```

> 0: <reserved>
== 0: OK
< 0: <reserved>

```

(c) Get parameter value of specified item

```

int  _TDSMDMssgNext(      // Return value                                     (*1)
int      md,              // in  : Message identifier   (Obtained by _TDSMDMssgInitialize())
int      mode,            // in  : Processing mode      (Must be =0)
void      *msg,           // in  : SECS Message storage area
char      *iname,         // in  : Item name of acquisition target
int      nop,             // in  : Number of items parameters that can be acquired
void      *item)          // out : Item parameter storage area                 (*2)
                                //      Make area of 4096 bytes or more

```

(*1) Return value

```

> 0: OK                  : Number of items parameters acquired
== 0: <reserved>
< 0: NG                  : Error code
    -ENOENT               : Specified item does not exist
    Other                 : Other error

```

(*2) item : Item parameter storage area

Store in specified item type. For list items, store number of lists as int type. When the number of parameters is more than one, they are stored continuously. If nop=0 is specified, it is considered that same number as number of items stored in the message is specified. For string items, append '¥0' to end of stored string.

(Note 1) When there are an indefinite number of list items in the message, the name of item expanded from list is expressed in the form of XXXX:99. Where XXXX is data item name and 99 is number of iterations starting from 1.

(Example) When data item name is SVID, the name in first repetition is SVID:1, name in first repetition is SVID:2, and name in 17th repetition is SVID:17.

(3) Create reply SECS message for received message

Identify the received message by referring to message structure definition file from received SECS message header and message body. Automatically identify the reply secondary message corresponding to the message and create reply message using default parameters.

If there are multiple reply secondary messages that can be sent back, search is started from the defined position of received primary message, and the secondary message of corresponding SF-Code found first is used as the reply message. If it reaches end of definition file but can not find it, it returns to the beginning of file and continues the search.

In the following case, it is possible to automatically create a normal reply message

- + Received message W-bit is ON
- + Received message is primary message
- + Identify the definition of received message
- + Identify the definition of secondary message that can be sent back

If can not identify a secondary message that can be sent back, create the following message for that reason.

- + S9F3 : S-Code of received primary message is undefined
- + S9F5 : F-Code of received primary message is undefined
- + S9F7 : The SF-Code of received primary message is defined but message structure is incorrect
- + SxF0 : Received primary message is normal but replying secondary message is undefined
(x is received S-Code)

(a) Create reply message for received message

```

int  _TDSMDMmsgAutoRes(    // Return value                                (*1)
int      md,              // in  : Message identifier      (Obtained by _TDSMDMmsgInitialize())
int      mode,            // in  : Processing mode          (*2)
TDSECSHead *rhd,          // in  : Received SECS Message header
void      *msg,           // i/o : SECS Message storage area      (Size is 'xlen')
                                //      in  : Received primary message
                                //      out : Secondary message to reply
int      rlen,            // in  : Byte length of received SECS message
int      xlen,            // in  : Byte size of SECS message storage area (msg)
char      *rname,         // out : Received message name      (Do not store if =0)
char      *sname,         // out : Message name for reply      (Do not store if =0)
int      *sf)             // out : Reply SF-Code              (Do not store if =0)
                                //      bit#14 - 8: S-Code    bit# 7 - 0: F-Code

```

(*1) Return value

```

>= 0: OK          : Byte length of reply SECS message
< 0: NG           : Error code
Other             : Other error
    = -941 - -944 : Refer to _TDSMDMmsgInitialize()
    -930          : W-bit OFF
    -931          : Received message is not a primary message

```

(*2) mode : Processing mode

```

E C
+-----+-----+-----+
| +-+ Acquisition message search specification
|   = 0: Received message is oposite side message. Search only oposite side message.
|   = 1: Not limited
+----+ Disable lock control
    = 0: According to specification at _TDSMDMmsgInitialize()
    = 1: Do not perform lock control

```

3.5 Other features

(1) During SECS/HSMS communication processing, acquisition of error information

The following functions are for obtaining error information that has occurred in library regarding SECS/HSMS communication processing.

This function can not obtain information on last error that occurred unless a call is made immediately after an error occurs.

(a) Get last error code encountered

```
int  _TDSErrorValue(    // Last error code encountered
int   fd)              // in  : Control identifier
                        //      (Obtained by _TDSCommOpen() or _TDSUDrvOpen())
```

(b) Get last error position code encountered

```
int  _TDSErrorPosition( // Last error position code encountered
int   fd)              // in  : Control identifier
```

(c) Get last error information encountered

```
void _TDSErrorStatus(
int   fd,              // in  : Control identifier
int   *err,            // out : Last error code encountered
int   *pos)            // out : Last error position code encountered
```

(2) During SECS/HSMS message processing, acquisition of error information

The following functions are for obtaining error information that has occurred in library regarding SECS/HSMS message processing (construction, analysis).

This function can not obtain information on last error that occurred unless a call is made immediately after an error occurs.

(a) Get last error code encountered

```
int  _TDSMssgErrorValue( // Last error code encountered
int   md)              // in  : Control identifier
                        //      (Obtained by _TDSMssgInit() or _TDSMssgFind())
```

(b) Get last error position code encountered

```
int  _TDSMssgErrorPosition(// Last error position code encountered
int   md)              // in  : Control identifier
```

(c) Get last error information encountered

```
void _TDSMssgErrorStatus(
int   md,              // in  : Control identifier
int   *err,            // out : Last error code encountered
int   *pos)            // out : Last error position code encountered
```

4. Tool

- (1) Communication control process
- (2) Communication data queue table reference tool
- (3) Error list display

4.1 Communication control process

If (mode&0x03)==1 is specified when _TDSCommOpen() is specified to use SECS/HSMS communication control process (tdsc), this process executes communication control with other side. Refer to 1. (3) (b).

(1) Startup method

```
tdsc [options] ini [sect]
~~~~~
```

ini : File path name of SECS/HSMS communication parameter setting file (.ini file)

sect : Section name to be used in the configuration file

options... : Optional parameters

-v : Redundant output specification

-f : Run this process (tdsc) in the foreground.

-w : (Other than Windows) After the display, confirm the end
(Windows) Do not check the end after display

+W : On Windows, disconnect from the boot console.
(The window itself can be erased only when the startup console is occupied by this process. Therefore, the window itself is erased only when it is launched by short cut to the program itself.)

4.2 Communication data queue table reference tool

When file or shared memory is used as communication data queue, the contents of message for which communication processing has been performed can be externally referenced. This tool realizes its function.

Also, this subsystem uses some system resources. This tool can forcibly release the resource. (Refer to -r option below)

(1) Startup method

```
tdsm [options..] ini [type]
```

```
~~~~~
```

ini : File path name of SECS/HSMS communication parameter setting file (.ini file)

type : Display target of SECS/HSMS communication data queue contents

type = qu : Whole (data part is header information only)

qh : Header record

qw : Write pointer record

qr : Read pointer record

qd : Data record

options... : Optional parameters

-v : Redundant output specification

-f : Display the data in order from oldest data

-b : Display the data in order from newest data

-s sect : Section name to be used in the configuration file

-q type : Queue file type

type = r : SECS Receive message queue

s : SECS send message queue

c : SECS send result message queue

-c form : Display format of queue data

form = 1 : List format

2 : Hexa format

3 : List and Hexa format

-R start,stop : Specify target record of display data

-D start,stop : Restrict display data to specified date

-T start,stop : Restrict target of display data to specified time.

-H : Do not show queue file lines when viewing

-Z : Does not display when all data in display line is '0'.

-B : Do not display if all data in display line is ''.

-r : Release related resource ID and shared memory ID. (Significant for UNIX)

-w : (Other than Windows) After the display, confirm the end
(Windows) Do not check the end after display

4.3 Error list display

Display a list of error numbers that TDS generates on standard output.

(1) Startup method

```
tdse [options..]  
~~~~~
```

options... : Optional parameters

-e	:	Output by English
-w	:	(Other than Windows) After the display, confirm the end (Windows) Do not check the end after display

A. SML format message structure definition file format

(1) SML Format

(a) Overall structure

```

//                                     Comment line

S1F1_H      HW                        // Message definition line

S1F2_E      E                        // Message definition
<L[2]       // Fixed quantity list definition
  <A[6]MDLN "EQUIP">                // Fixed quantity item definition
  <A[6]REVISION "02.181">
>                                     // List end definition

S1F3_H      HW
<L[N]NOI1   // Undefined number list definition
  <L[2]
    <A[0..16]RPTID "RP001">        // Undefined number item definition
    <L[1.. 4]NOI2
      <U4[1..8]SVID "0,1,2">
  >
>
>

```

(b) Message definition line

```

name flag
~~~~~

```

name : Specifies name of this message in the form "SxFy_ZZZZ".
Specify S-Code to 'x', F-Code to 'y'. ZZZZ can specify any string.

(Note 1) Multiple messages can be defined by changing the name in same SF-Code.
When applying message definition to received message, the destination and message structure are compared to select the most suitable message.

(Note 2) Names are converted to upper case even if lower case is specified.

flag : Specify the combination of the following characters.
H : Host side outgoing message
E : Equipment side outgoing message
W : Primary message waiting to receive secondary message
D : Default secondary message

(Note 3) H and E can be omitted, and if neither is specified, they are treated as both outgoing messages.

(Note 4) D: When _TDSMDMssgAutoRes() searches for received primary message, if there is no corresponding primary message, the secondary message is determined using only SF code regardless of message structure. It is not necessary to define a secondary message specifying 'D'.

(c) List definition line, List end definition line

<L[n] or <L[N]name or <L[m..x]name or <L[0|x]name
 ~~~~~

n : Number of list items (Fixed number of items)

N : Keyword indicating that number of list items is undefined (Fixed number of items)

m : Minimum number of list items (Number of items undefined)

x : Maximum number of list items (Number of items undefined)

name : Name used to determine number of items in unfixed number list

(Note 1) In the case of an indefinite number list other than [0|x], the items that can be defined in the list must be "single".

(Note 2) When [0|x] is specified, it indicates that the number of items included in the list is 0 or x. In this case, the items included in the list do not have to be single, and x items can be specified.

In this case, the number of items can not be 1 to x-1. It is always 0 or x.

(Note 3) Names are not case sensitive.

(Note 4) When specifying an unfixed number list, must specify 'name' (field name).

>  
 ~

(Note 5) Any number of list definitions having an indefinite number of items can be performed in any hierarchy.

An item name must always be specified for an indefinite number of list items, and the number of lists must be determined when constructing a message. Refer to the note in "3.4

(1) (c) Data item parameter value setting" for expansion of item names when the number of unfixed-piece lists is fixed.

(Note 6) Items included in unfixed number list must be single (except for L[0|x] specification). The setting example is shown below. (Closed list ">" is omitted due to space limitations.)

| + OK specification | + NG specification | + OK specification |
|--------------------|--------------------|--------------------|
| 1. <L[N]NO11       | 1. <L[0..2]NO11    | 1. <L[0 2]NO11     |
| <L[2]              | <A[20]ID>          | <A[20]ID>          |
| <A[20]ID>          | <U4[1]DATA>        | <U4[1]DATA>        |
| <U4[1..2]DATA>     |                    |                    |
| 2. <L[0..5]NO1     | 2. <L[N]NO11       | 2. <L[0 2]NO11     |
| <A[0..40]TEXT>     | <A[20]TEXT>        | <A[20]TEXT>        |
|                    | <L[2]              | <L[2]              |
|                    | <A[20]ID>          | <A[20]ID>          |
|                    | <U4[1]DATA>        | <U4[1]DATA>        |

## (d) Item definition line

```
<X[n]name "val0[,val1[,...]]" ["cp00[,cp01[,...]]" [... ["cp30[,cp31[,...]]"]]]>
```

or

```
<X[m..x]name "val0[,val1[,...]]" ["cp00[,cp01[,...]]" [... ["cp30[,cp31[,...]]"]]]>
```

X : Item type

|                       |                                   |                            |
|-----------------------|-----------------------------------|----------------------------|
| B: 1Byte Binary       | I1: 1Byte Signed integer          | U1: 1Byte Unsigned integer |
| T: 1Byte Logical      | I2: 2Byte Signed integer          | U2: 2Byte Unsigned integer |
| A: Ascii string       | I4: 4Byte Signed integer          | U4: 4Byte Unsigned integer |
| J: 2Byte JIS-8 string | I8: 8Byte Signed integer          | U8: 8Byte Unsigned integer |
| K: Multi Byte string  | F4: 4Byte IEEE754 Floarting point |                            |
|                       | F8: 8Byte IEEE754 Floarting point |                            |

n : Number of item (Fixed number of items)

m : Minimum number of item (Number of items undefined)

x : Maximum number of item (Number of items undefined)

name : Item name (Not case sensitive. Be sure to specify.)

val0 .. : Default field value to use when sending

In the case of multiple items, whole is enclosed by '""' and the elements are separated by ','. In case of character items, it is specified as character string.

cp00 .. : Item value to be used for receiving check (up to 4 can be specified)

In the case of multiple items, specify in the same format as val.

(Note 1) For item type K (Multi Byte string), item match judgment can not be performed by specifying this parameter.

(Note 2) Unlike the list definition line, it is not possible to specify indefinite number of items in "N" with <X[N]name ...>.

(Note 3) When specifying '""' or '¥' in character string enclosed by '""' in character item specification, escape with '¥' preceding. That is, when specifying [123"456¥789] as character string, it describes as "123¥"456¥¥789".

(Note 4) When received message is compared with message definition, a check item such as cp00 determines whether the corresponding received data value matches value specified here. If it matches, it is determined that the received message matches this message definition. That is, a plurality of message definitions having same message structure can be defined for each settable value. If no check item is specified, it is not determined whether item values match.

## &lt;Setting Example&gt;

```
<A [0..6]NAME "NAME0" "NAME0" "NAME1" "NAME02" "NAME03">
```

```
<B [1]B1 "1" "1" "2">
```

```
<B [3]B3 "10, 20, 30" "10, 20, 30" "11, 21, 31" "12, 22, 32">
```

```
<U2[1]NWAFFER>
```

```
<B [1]CEID "122" "122" "123">
```

```
<A [6]MDLN "MACH01">
```

(Note 5) The following can be specified as a macro name

- + @TIME : Current time  
Specify combination of YYYY, YY, MM, DD, hh, mm, ss as default parameter value.
- + @FILEXXXXX : Acquisition of setting data from specified file  
Define @FILEXXXXX in the item list and specify type and number of data.  
The XXXXX part is an arbitrary name, and file path name is specified as default parameter value.

<Setting Example>

<A [ 16]@TIME "">

<A [ 16]@TIME "YYYYMMDDhhmmss00">

<U2[128]@FILEDATA0 "U2DataFile.dat">

<U1[1000..16777215]@FILEDATA1 "U1DataFile.dat">

## B. SECS/HSMS Communication library (C# version) API specification

- (1) SECS/HSMS Message communication function (usual API)
- (2) SECS/HSMS Message communication function (abbreviated API)
- (3) SECS Message construction, analysis function
- (4) SECS Message construction and analysis function using SML format message structure definition

(Note 1) Each function shown in this chapter has the following attributes.

- + namespace : TDCSL
- + class : TDS

(Note 2) Refer to Chapter 3 for processing of each function, parameter details, etc.

(Note 3) The following DLLs are required to use this library.

- + In case of Windows
  - TDCSS.dll, TDS.dll

## B.1 SECS/HSMS Message communication function (usually API)

In the following explanation, refer to corresponding API described in section 3.1 for return value, parameter contents, etc.

## (1) SECS/HSMS communication : Open processing

```
int  TDS._CommOpen(      // Return value
int    mode,             // in  : Processing mode
string *ini,             // in  : .ini file path name
string *sect)            // in  : .ini file section name to use
```

(Note 1) It is not able to set Callback function like C function.

If you need Callback function, you need to perform Callback processing on the AP side with the following code, for example.

```
+ Start thread for Callback function
Thread  th;
string  param;
param=fd.ToString(); // 'fd' is the control identifier obtained by TDS._CommOpen()
th=new Thread(new ParameterizedThreadStart(RecvProcThread));
th.Start(param);

+ Thread for Callback function
private static void RecvProcThread(object param)
{
    byte[]    msg=new byte[512],hd=new byte[12];
    uint      xid;
    int       fd,rtn,req,devid,sf;
    fd=int.Parse((string)param);
    for(;;){
        req=0;
        if((rtn=TDS._CommRecv(fd,0,out devid,out sf,out xid,ref msg,512,ref hd))==(-951)){
            Thread.sleep(100);
        }else{
            if((-1000)<rtn && rtn<(-959)) req=(-rtn)-900;
            CBRecvProc(req,rtn,devid,sf,xid,hd,msg);
        }
    }
}

+ Callback processing function
private static int
CBRecvProc(int req, int rtn, int devid, int sf, uint xid, byte[] thd, byte[] msg)
{
    // Callback processing
    // The parameters are same as each member variable of structure TDSCBData shown
    // in 2.2(3).
}
```

## (2) SECS/HSMS communication : Close processing

```
int  TDS._CommClose(      // Return value
int    fd,               // in  : Control identifier          (Obtained by TDS._CommOpen())
int    mode)             // in  : Processing mode          (Must be =0)
```



## (3) SECS/HSMS communication : Receive processing

```

int  TDS._CommRecv(      // Return value
int      fd,              // in  : Control identifier      (Obtained by TDS._CommOpen())
int      mode,           // in  : Processing mode
out int   devid,         // out : Received message Device ID
out int   sf,            // out : Received message SF-Code
out uint  xid,           // out : Received message Transaction ID
byte[]    msg,           // out : Received message Body      (Data part)
int       len,           // in  : Receiveable message length (Number of bytes)
byte[]    hd)            // out : Received message Header

```

## (4) SECS/HSMS communication : Send processin

```

int  TDS._CommSend(      // Return value
int      fd,              // in  : Control identifier      (Obtained by TDS._CommOpen())
int      mode,           // in  : Processing mode
int      devid,          // in  : Send message Device ID
int      sf,             // in  : Send message SF-Code
uint     xid,            // in  : Send message Transaction ID
byte[]    msg,           // in  : Send message Body      (Data part)
int       len,           // in  : Send message length    (Number of bytes)
byte[]    hd)            // out : Send message Header    (Available only ((mode&0x0f00)==0))

```

## (5) SECS/HSMS communication : Illegal message receiving result sending process

```

int  TDS._CommSendError( // Return value
int      fd,              // in  : Control identifier      (Obtained by TDS._CommOpen())
int      mode,           // in  : Processing mode                (Must be =0)
int      devid,          // in  : Send message Device ID
int      sf,             // in  : Send message SF-Code
byte[]    rhd)           // in  : Target bad header of S9Fx    (Only significant for S9Fx)

```

## (6) Connection status check

```

int  TDS._CommStatus(    // Return value
int      fd,            // in  : Control identifier      (Obtained by TDS._CommOpen())
int      mode)          // in  : Processing mode          (Must be =0)

```

## (7) Checking connection status for each session ID (HSMS-GS)

```

int  TDS._CommSelectStatus( // Return value
int      fd,                // in  : Control identifier      (Obtained by TDS._CommOpen())
int      mode,              // in  : Processing mode          (Must be =0)
int      devid)             // in  : Session ID (Device ID)

```

## (8) Additional output of user comment messages to communication trace file

```

int  TDS._CommUserComment(// Return value
int      fd,              // in  : Control identifier      (Obtained by TDS._CommOpen())
int      mode,            // in  : Processing mode          (Must be =0)
string   comm)            // in  : Comment message to output

```

## B. 2 SECS/HSMS Message communication function (abbreviated API)

In the following explanation, refer to corresponding API described in section 3.1 for return value, parameter contents, etc.

## (1) SECS/HSMS communication : Open processing

```
int  TDS._UDrvOpen(      // Return value
int    mode,             // in  : Processing mode
string *ini,             // in  : .ini file path name
string *sect,            // in  : .ini file section name to use
int    mask)            // in  : Mask of acquisition target even
```

## (2) SECS/HSMS communication : Close processing

```
int  TDS._UDrvClose(     // Return value
int    fd,              // in  : Control identifier      (Obtained by TDS._UDrvOpen())
int    mode)            // in  : Processing mode        (Must be =0)
```

## (3) SECS/HSMS communication : Receive processing

```
int  TDS._UDrvRecv(      // Return value
int    fd,              // in  : Control identifier      (Obtained by TDS._UDrvOpen())
int    mode,            // in  : Processing mode
out int devid,          // out : Received message Device ID
out int sf,             // out : Received message SF-Code
out uint xid,           // out : Received message Transaction ID
byte[] msg,             // out : Received message Body   (Data part)
int    len,            // in  : Receiveable message length (Number of bytes)
byte[] hd)             // out : Received message Header
```

## (4) SECS/HSMS communication : Send processin

```
int  TDS._UDrvSend(      // Return value
int    fd,              // in  : Control identifier      (Obtained by TDS._UDrvOpen())
int    mode,            // in  : Processing mode
int    devid,          // in  : Send message Device ID
int    sf,             // in  : Send message SF-Code
uint   xid,            // in  : Send message Transaction ID
byte[] msg,            // in  : Send message Body       (Data part)
int    len,            // in  : Send message length     (Number of bytes)
byte[] hd)            // out : Send message Header     (Available only ((mode&0x0f00)==0))
```

## (5) SECS/HSMS communication : Illegal message receiving result sending process

```

int  TDS._UDrvSendError( // Return value
int      fd,           // in  : Control identifier      (Obtained by TDS._UDrvOpen())
int      mode,         // in  : Processing mode          (Must be =0)
int      devid,        // in  : Send message Device ID
int      sf,           // in  : Send message SF-Code
byte[]   rhd)          // in  : Target bad header of S9Fx  (Only significant for S9Fx)

```

## (6) Connection status check

```

int  TDS._UDrvStatus( // Return value
int      fd,          // in  : Control identifier      (Obtained by TDS._UDrvOpen())
int      mode)        // in  : Processing mode          (Must be =0)

```

## (7) Checking connection status for each session ID (HSMS-GS)

```

int  TDS._UDrvSelectStatus(// Return value
int      fd,              // in  : Control identifier      (Obtained by TDS._UDrvOpen())
int      mode,            // in  : Processing mode          (Must be =0)
int      devid)           // in  : Session ID (Device ID)

```

## (8) Additional output of user comment messages to communication trace file

```

int  TDS._UDrvUserComment(// Return value
int      fd,              // in  : Message identifier      (Obtained by TDS._UDrvOpen())
int      mode,            // in  : Processing mode          (Must be =0)
string   comm)            // in  : Comment message to output

```

## B. 3 SECS Message construction, analysis function

In the following explanation, refer to corresponding API described in section 3.1 for return value, parameter contents, etc.

## (1) SECS message construction

## (a) Initialize

```

int  TDS._MssgInit(      // Return value
int    mode,            // in  : Processing mode
byte[] msg,             // out : SECS Message storage area
int    len,             // in  : SECS Message storage area byte size
int    fdc)             // in  : SECS communication control identifier (Already opened)
                        //      or Maximum SECS message byte size to process.

```

## (b) End processing

```

int  TDS._MssgEnd(      // Return value
int    md,              // in  : Message identifier (Obtained by TDS._MssgInit())
int    mode,            // in  : Processing mode (Must be =0)
byte[] msg)             // i/o : SECS Message storage area

```

## (c-1) Add data item (usual format)

```

int  TDS._MssgBuild(    // Return value
int    md,              // in  : Message identifier (Obtained by TDS._MssgInit())
int    mode,            // in  : Processing mode (Must be =0)
byte[] msg,             // i/o : SECS Message storage area
int    form,            // in  : Additional item Format code
int    nop,             // in  : Additional item number of parameter
TYPE   item)            // in  : Additional item parameter storage area

```

(Note) Additional item parameter storage area can be specified as TYPE in following format

```

+ void*      + string
+ byte[]     + float[]   + double[]
+ short[]    + int[]     + long[]
+ ushort[]   + uint[]    + ulong[]

```

## (c-2) Add data item (string format)

```

int  TDS._MssgBuildL(   // Return value (Refer to (c-1))
int    md,              // in  : Message identifier (Obtained by TDS._MssgInit())
int    mode,            // in  : Processing mode
byte[] msg,             // i/o : SECS Message storage area
string str)             // in  : SECS data items in given string format

```

## (2) SECS Message analysis

## (a) Initialize

```

int  TDS._MssgFind(      // Return value
int      mode,          // in  : Processing mode
byte[]    msg,          // in  : SECS Message storage area
int      len,          // in  : SECS Message storage area byte size
int      fdc,          // in  : SECS communication control identifier      (Already opened)
                                     //      or Maximum SECS message byte size to process.
byte[]    hd,          // in  : SECS Message Header
out string mname)       // out : SECS Message name

```

## (b) End processing

```

int  TDS._MssgExit(      // Return value
int      md,            // in  : Message identifier      (Obtained by TDS._MssgFind())
int      mode,          // in  : Processing mode        (Must be =0)
byte[]    msg)          // in  : SECS Message storage area

```

## (c-1) Data item acquisition (usual format)

```

int  TDS._MssgNext(      // Return value
int      md,            // in  : Message identifier      (Obtained by TDS._MssgFind())
int      mode,          // in  : Processing mode        (Must be =0)
byte[]    msg,          // in  : SECS Message storage area
out int    form,        // out : Item Format code
out int    parl,        // out : Byte size occupied by one item
out int    noi,         // out : Number of items parameters
void      *item,        // out : Item parameter value storage area
int      xlen,          // in  : Byte size of item parameter value storage area
out string sitem)       // out : If the acquired item is a string, store that string.
                                     //      What is stored is part that could be stored in item.
                                     //      That is, it can not exceed 'xlen'.

```

## (c-2) Data item acquisition (string format)

```

int  TDS._MssgNextL(     // Return value
int      md,            // in  : Message identifier      (Obtained by TDS._MssgFind())
int      mode,          // in  : Processing mode
byte[]    msg,          // in  : SECS Message storage area
out int    form,        // out : Item Format code
out int    noi,         // out : Number of items parameters
out string str)          // out : SECS data items of given string format (up to 1000 bytes)

```

## B. 4 SECS message construction and analysis function using SML format message structure definition

In the following explanation, refer to corresponding API described in section 3.1 for return value, parameter contents, etc.

## (0) Message structure definition information acquisition and release

## (a) Message structure definition information acquisition and initialization

```

int  TDS._MDMssgInitialize(    //Return value
int      mode,                // in  : Processing mode
                                //      10
                                //      ----+-----+
                                //      +--- 0:SML Format    2:<reserved>
                                //      1:<reserved>    3:<reserved>
int      fdc,                 // in  : SECS communication control identifier    (Already opened)
string   path)                // in  : Message structure definition file path name

```

## (b) Message structure definition information release

```

void TDS._MDMssgTerminate(
int      md,                  // in  : Message identifier    (Obtained by TDS._MDMssgInitialize())
int      mode)                // in  : Processing mode          (Must be =0)

```

## (1) SECS message construction

## (a) Initialize

```

int  TDS._MDMssgInit(    // Return value
int      md,            // in  : Message identifier   (Obtained by TDS._MDMssgInitialize())
int      mode,          // in  : Processing mode                               (Must be =0)
byte[]   msg,           // out : SECS Message storage area
int      len,           // in  : SECS Message storage area byte size
string   mname)         // in  : Message name in message structure definition file

```

## (b) End processing

```

int  TDS._MDMssgEnd(    // Return value
int      md,            // in  : Message identifier   (Obtained by TDS._MDMssgInitialize())
int      mode,          // in  : Processing mode                               (Must be =0)
byte[]   msg)           // i/o : SECS Message storage area

```

## (c) Data item parameter value setting

```

int  TDS._MDMssgBuild(  // Return value
int      md,            // in  : Message identifier   (Obtained by TDS._MDMssgInitialize())
int      mode,          // in  : Processing mode                               (Must be =0)
byte[]   msg,           // i/o : SECS Message storage area
string   iname,         // in  : Additional item name
int      nop,           // in  : Additional item number of parameters
TYPE     item)          // in  : Addttional item parameter strage area

```

(Note) Additional item parameter storage area can be specified as TYPE in following format

|            |           |            |
|------------|-----------|------------|
| + void*    | + string  |            |
| + byte []  | + float[] | + double[] |
| + short [] | + int []  | + long []  |
| + ushort[] | + uint [] | + ulong [] |



## (2) SECS Message analysis

## (a) Initialize

```

int  TDS._MDMssgFind(    // Return value
int      md,            // in  : Message identifier   (Obtained by TDS._MDMssgInitialize())
int      mode,          // in  : Processing mode
byte[]   msg,           // in  : SECS Message storage area
int      len,           // in  : SECS Message storage area byte size
int      sf,            // in  : SF-Code to be acquired
out string mname)       // out : Message name in message structure definition file

```

## (b) End processing

```

int  TDS._MDMssgExit(    // Return value
int      md,            // in  : Message identifier   (Obtained by TDS._MDMssgInitialize())
int      mode,          // in  : Processing mode                      (Must be =0)
byte[]   msg)           // in  : SECS Message storage area

```

## (c) Get parameter value of specified item

```

int  TDS._MDMssgNext(    // Return value
int      md,            // in  : Message identifier   (Obtained by TDS._MDMssgInitialize())
int      mode,          // in  : Processing mode                      (Must be =0)
byte[]   msg,           // in  : SECS Message storage area
string   iname,         // in  : Item name of acquisition target
int      nop,           // in  : Number of items parameters that can be acquired
TYPE     item)          // out : Item parameter storage area

```

(Note 1) If the item is a string, you can use

```

int  TDS._MDMssgNext(    // Return value
int      md,            // in  : Message identifier   (Obtained by TDS._MDMssgInitialize())
int      mode,          // in  : Processing mode                      (Must be =0)
byte[]   msg,           // in  : SECS Message storage area
string   iname,         // in  : Item name of acquisition target
int      nop,           // in  : Number of items parameters that can be acquired
out string item)        // out : Item parameter storage area

```

(Note 2) If the item is not a string, you can use

```

int  TDS._MDMssgNext(    // Return value
int      md,            // in  : Message identifier   (Obtained by TDS._MDMssgInitialize())
int      mode,          // in  : Processing mode                      (Must be =0)
byte[]   msg,           // in  : SECS Message storage area
string   iname,         // in  : Item name of acquisition target
int      nop,           // in  : Number of items parameters that can be acquired
TYPE[]   item)          // out : Item parameter storage area
//
//      'TYPE' is one of the following
//      + byte      + float    + double
//      + short     + int      + long
//      + ushort    + uint     + ulong

```

## (3) Create reply SECS message for received message

## (a) Create reply message for received message

```

int  TDS._MDMssgAutoRes( // Return value
int      md,           // in  : Message identifier   (Obtained by TDS._MDMssgInitialize())
int      mode,         // in  : Processing mode                               (Must be =0)
byte[]   rhd,          // in  : Received SECS Message header
byte[]   msg,          // i/o : SECS Message storage area                     (Size is 'xlen')
int      rlen,         // in  : Byte length of received SECS message
int      xlen,         // in  : Byte size of SECS message storage area (msg)
out string rname,      // out : Received message name
out string sname,      // out : Message name for reply
out int   sf)          // out : Reply SF-Code

```

## C. SECS/HSMS Communication library (Visual Basic version) API specification

- (1) SECS/HSMS Message communication function (usual API)
- (2) SECS/HSMS Message communication function (abbreviated API)
- (3) SECS Message construction, analysis function
- (4) SECS Message construction and analysis function using SML format message structure definition

(Note 1) Each function shown in this chapter has the following attributes

```
+ namespace : TDVBL
+ class      : TDS
```

The functions described in this chapter operate as instance functions of class TDS. Therefore, create an instance of class TDS before calling each function and use it as a member function of that instance.

(Note 2) Refer to Chapter 3 for processing of each function, parameter details, etc.

(Note 3) The following DLLs are required to use this library.

```
+ In case of Windows
- TDVBS.dll, TDS.dll
```

## C.1 SECS/HSMS Message communication function (usually API)

In the following explanation, refer to corresponding API described in section 3.1 for return value, parameter contents, etc.

## (0) Create instance

```
dim td as TDS
```

(Note) We use td for the following description of member functions.

## (1) SECS/HSMS communication : Open processing

```
function td._TDSCommOpen(    // Return value
byval mode    as integer,    // in  : Processing mode
byval ini     as string,     // in  : .ini file path name
byval sect    as string)     // in  : .ini file section name to use
    as integer
```

(Note 1) It is not able to set Callback function like C function.

If you need Callback function, you need to perform Callback processing on the AP side with the following code, for example.

+ Global variable declaration

```
dim Td    as TDS
dim Fd    as integer
```

+ Start thread for Callback function

```
dim th    as Thread
th=new Thread(new ParameterizedThreadStart(AddressOf RecvProcThread))
th.Start("");
```

+ Thread for Callback function

```
private sub RecvProcThread(byval param as object)
dim  hd(12),msg(512)    as byte
dim  rtn,req,devid,sf,xid as integer
do
    req=0
    rtn=Td._TDSCommRecv(Fd,0,devid,sf,xid,msg,512,hd)
    if rtn=(-951) then
        Sleep(100)
    else
        if (-1000)<rtn and rtn<(-959) then req = (-rtn)-900
        CBRecvProc(req,rtn,devid,sf,xid,hd,msg)
    endif
loop
```

+ Callback processing function

```
private function CBRecvProc(byval req as integer, byval rtn as integer, _
    byval devid as integer, byval sf as integer, byval xid as integer, _
    byval thd() as byte,    byval msg as byte)    as integer
// Callback processing
// The parameters are same as each member variable of structure TDSCBData shown
// in 2.2(3).
end function
```

## (2) SECS/HSMS communication : Close processing

```

function td._TDSCommClose( // Return value
byval fd      as integer, // in : Control identifier      (Obtained by _TDSCommOpen())
byval mode    as integer) // in : Processing mode          (Must be =0)
                        as integer

```

## (3) SECS/HSMS communication : Receive processing

```

function td._TDSCommRecv( // Return value
byval fd      as integer, // in : Control identifier      (Obtained by _TDSCommOpen())
byval mode    as integer, // in : Processing mode
byref devid   as integer, // out : Received message Device ID
byref sf      as integer, // out : Received message SF-Code
byref xid     as integer, // out : Received message Transaction ID
byval msg()   as byte,    // out : Received message Body   (Data part)
byval len     as integer, // in : Receiveable message length (Number of bytes)
byval hd()    as byte)    // out : Received message Header
                        as integer

```

## (4) SECS/HSMS communication : Send processing

```

function td._TDSCommSend( // Return value
byval fd      as integer, // in : Control identifier      (Obtained by _TDSCommOpen())
byval mode    as integer, // in : Processing mode
byval devid   as integer, // in : Send message Device ID
byval sf      as integer, // in : Send message SF-Code
byval xid     as integer, // in : Send message Transaction ID
byval msg()   as byte,    // in : Send message Body       (Data part)
byval len     as integer, // in : Send message length     (Number of bytes)
byval hd()    as byte)    // out : Send message Header    (Available only ((mode&h0f00)==0))
                        as integer //

```

## (5) SECS/HSMS communication : Illegal message receiving result sending process

```

function td._TDSCommSendError( // Return value
byval fd      as integer, // in : Control identifier      (Obtained by _TDSCommOpen())
byval mode    as integer, // in : Processing mode
byval devid   as integer, // in : Send message Device ID
byval sf      as integer, // in : Send message SF-Code
byval rhd()   as byte)    // in : Target bad header of S9Fx (Only significant for S9Fx)
                        as integer

```

## (6) Connection status check

```
function td._TDSCommStatus( // Return value
byval fd      as integer,  // in  : Control identifier      (Obtained by _TDSCommOpen())
byval mode    as integer)  // in  : Processing mode        (Must be =0)
                        as integer
```

## (7) Checking connection status for each session ID (HSMS-GS)

```
function yd._TDSCommSelectStatus( // Return value
byval fd      as integer,  // in  : Control identifier      (Obtained by _TDSCommOpen())
byval mode    as integer,  // in  : Processing mode        (Must be =0)
byval devid   as integer)  // in  : Session ID (Device ID)
                        as integer
```

## (8) Additional output of user comment messages to communication trace file

```
function td._TDSCommUserComment( // Return value
byval fd      as integer,  // in  : Control identifier      (Obtained by _TDSCommOpen())
byval mode    as integer,  // in  : Processing mode
byval comm    as string)   // in  : Comment message to output
                        as integer
```

## C. 2 SECS/HSMS Message communication function (abbreviated API)

In the following explanation, refer to corresponding API described in section 3.1 for return value, parameter contents, etc.

## (1) SECS/HSMS communication : Open processing

```
function td._TDSUDrvOpen(    // Return value
byval mode    as integer,    // in  : Processing mode
byval ini     as string,     // in  : .ini file path name
byval sect    as string,     // in  : .ini file section name to use
byval mask    as integer)    // in  : Mask of acquisition target even
                        as integer
```

## (2) SECS/HSMS communication : Close processing

```
function td._TDSUDrvClose(    // Return value
byval fd      as integer,     // in  : Control identifier          (Obtained by _TDSUDrvOpen())
byval mode    as integer)    // in  : Processing mode            (Must be =0)
                        as integer
```

## (3) SECS/HSMS communication : Receive processing

```
function td._TDSUDrvRecv(    // Return value
byval fd      as integer,     // in  : Control identifier          (Obtained by _TDSUDrvOpen())
byval mode    as integer,     // in  : Processing mode
byref devid   as integer,     // out : Received message Device ID
byref sf      as integer,     // out : Received message SF-Code
byref xid     as integer,     // out : Received message Transaction ID
byval msg()   as byte,        // out : Received message Body      (Data part)
byval len     as integer,     // in  : Receiveable message length (Number of bytes)
byval hd()    as byte)        // out : Received message Header
                        as integer
```

## (4) SECS/HSMS communication : Send processin

```
function td._TDSUDrvSend(    // Return value
byval fd      as integer,     // in  : Control identifier          (Obtained by _TDSUDrvOpen())
byval mode    as integer,     // in  : Processing mode
byval devid   as integer,     // in  : Send message Device ID
byval sf      as integer,     // in  : Send message SF-Code
byval xid     as integer,     // in  : Send message Transaction ID
byval msg()   as byte,        // in  : Send message Body          (Data part)
byval len     as integer,     // in  : Send message length        (Number of bytes)
byval hd()    as byte)        // out : Send message Header        (Available only ((mode&h0f00)==0))
                        as integer    //
```

## (5) SECS/HSMS communication : Illegal message receiving result sending process

```

function td._TDSUDrvSendError(    // Return value
byval fd      as integer,    // in : Control identifier      (Obtained by _TDSUDrvOpen())
byval mode    as integer,    // in : Processing mode
byval devid   as integer,    // in : Send message Device ID
byval sf      as integer,    // in : Send message SF-Code
byval rhd()   as byte)       // in : Target bad header of S9Fx  (Only significant for S9Fx)
                                as integer

```

## (6) Connection status check

```

function td._TDSUDrvStatus( // Return value
byval fd      as integer,    // in : Control identifier      (Obtained by _TDSUDrvOpen())
byval mode    as integer)    // in : Processing mode        (Must be =0)
                                as integer

```

## (7) Checking connection status for each session ID (HSMS-GS)

```

function td._TDSUDrvSelectStatus( // Return value
byval fd      as integer,    // in : Control identifier      (Obtained by _TDSUDrvOpen())
byval mode    as integer,    // in : Processing mode        (Must be =0)
byval devid   as integer)    // in : Session ID (Device ID)

```

## (8) Additional output of user comment messages to communication trace file

```

function td._TDSUDrvUserComment( // Return value
byval fd      as integer,    // in : Control identifier      (Obtained by _TDSUDrvOpen())
byval mode    as integer,    // in : Processing mode
byval comm    as string)     // in : Comment message to output
                                as integer

```



## C.3 SECS Message construction, analysis function

In the following explanation, refer to corresponding API described in section 3.1 for return value, parameter contents, etc.

## (1) SECS message construction

## (a) Initialize

```
function td._TDSMsgInit( // Return value
byval mode as integer, // in : Processing mode
byval msg() as byte, // out : SECS Message storage area
byval len as integer, // in : SECS Message storage area byte size
byval fdc as integer) // in : SECS communication control identifier (Already opened)
                        // or Maximum SECS message byte size to process.
```

## (b) End processing

```
function td._TDSMsgEnd( // Return value
byval md as integer, // in : Message identifier (Obtained by _TDSMsgInit())
byval mode as integer, // in : Processing mode (Must be =0)
byval msg() as byte) // i/o : SECS Message storage area
as integer
```

## (c-1) Add data item (usual format)

```
function td._TDSMsgBuild( // Return value
byval md as integer, // in : Message identifier (Obtained by _TDSMsgInit())
byval mode as integer, // in : Processing mode (Must be =0)
byval msg() as byte, // i/o : SECS Message storage area
byval form as integer, // in : Additional item Format code
byval nop as integer, // in : Additional item number of parameter
byval item as TYPE) // in : Additional item parameter storage area
as integer
```

(Note 1) Additional item parameter storage area can be specified as TYPE in following format

```
+ string
+ sbyte() + integer () + long () + single()
+ byte () + uinteger () + ulong () + double()
```

## (c-2) Add data item (string format)

```
function td._TDSMsgBuildL( // Return value
byval md as integer, // in : Message identifier (Obtained by _TDSMsgInit())
byval mode as integer, // in : Processing mode (Must be =0)
byval msg() as byte, // i/o : SECS Message storage area
byval str as string) // in : SECS data items in given string format
as integer
```

## (2) SECS Message analysis

## (a) Initialize

```

function td._TDSMssgFind(    // Return value
byval  mode    as integer,  // in  : Processing mode
byval  msg()   as byte,     // in  : SECS Message storage area
byval  len     as integer,  // in  : SECS Message storage area byte size
byval  fdc     as integer,  // in  : SECS communication control identifier (Already opened)
                                //      or Maximum SECS message byte size to process.
byval  hd()    as byte,     // in  : SECS Message Header
byref  mname   as string)   // out : SECS Message name
                                as integer

```

## (b) End processing

```

function td._TDSMssgExit(    // Return value
byval  md      as integer,   // in  : Message identifier      (Obtained by _TDSMssgFind())
byval  mode    as integer,   // in  : Processing mode        (Must be =0)
byval  msg()   as byte)     // in  : SECS Message storage area
                                as integer

```

## (c-1) Data item acquisition (usual format)

```

function td._TDSMssgNext(    // Return value
byval  md      as integer,   // in  : Message identifier      (Obtained by _TDSMssgFind())
byval  mode    as integer,   // in  : Processing mode        (Must be =0)
byval  msg()   as byte,     // in  : SECS Message storage area
byref  form    as integer,   // out : Item Format code
byref  parl    as integer,   // out : Byte size occupied by one item
byref  noi     as integer,   // out : Number of items parameters
byval  item()  as byte,     // out : Item parameter value storage area
byval  xlen    as integer,   // in  : Byte size of item parameter value storage are
byref  sitem   as string)   // out : If the acquired item is a string, store that string.
                                //      What is stored is part that could be stored in item.
                                //      That is, it can not exceed 'xlen'.

```

## (c-2) Data item acquisition (string format)

```

function td._TDSMssgNextL(    // Return value
byval  md      as integer,   // in  : Message identifier      (Obtained by _TDSMssgFind())
byval  mode    as integer,   // in  : Processing mode
byval  msg()   as byte,     // in  : SECS Message storage area
byref  form    as integer,   // out : Item Format code
byref  noi     as integer,   // out : Number of items parameter
byref  str     as string)   // out : SECS data items of given string form. (up to 1000bytes)
                                as integer

```

## C. 4 SECS message construction and analysis function using SML format message structure definition

In the following explanation, refer to corresponding API described in section 3.1 for return value, parameter contents, etc.

## (0) Message structure definition information acquisition and release

## (a) Message structure definition information acquisition and initialization

```
function td._TDSMDMssgInitialize( // Return value
byval mode    as integer, // in : Processing mode
                                //          10
                                //          -----+
                                //          +--- 0:SML Format    2:<reserved>
                                //          1:<reserved>    3:<reserved>
byval fdc     as integer, // in : SECS communication control identifier (Already opened)
byval path    as string)  // in : Message structure definition file path name
                                as integer
```

## (b) Message structure definition information release

```
sub      td._TDSMDMssgTerminate(
byval md    as integer, // in : Message identifier (Obtained by _TDSMDMssgInitialize())
byval mode  as integer)  // in : Processing mode (Must be =0)
```

## (1) SECS message construction

## (a) Initialize

```

function td._TDSMDMssgInit( // Return value
byval md      as integer,   // in  : Message identifier (Obtained by _TDSMDMssgInitialize())
byval mode    as integer,   // in  : Processing mode                               (Must be =0)
byval msg()   as byte,      // out : SECS Message storage area
byval len     as integer,   // in  : SECS Message storage area byte size
byval mname   as string)    // in  : Message name in message structure definition file
                        as integer

```

## (b) End processing

```

function td._TDSMDMssgEnd( // Return value
byval md      as integer,   // in  : Message identifier (Obtained by _TDSMDMssgInitialize())
byval mode    as integer,   // in  : Processing mode                               (Must be =0)
byval msg()   as byte)      // i/o : SECS Message storage area
                        as integer

```

## (c) Data item parameter value setting

```

function td._TDSMDMssgBuild( // Return value
byval md      as integer,   // in  : Message identifier (Obtained by _TDSMDMssgInitialize())
byval mode    as integer,   // in  : Processing mode                               (Must be =0)
byval msg()   as byte,      // i/o : SECS Message storage area
byval iname   as string,    // in  : Additional item name
byval nop     as integer,   // in  : Additional item number of parameters
byval item    as TYPE)      // in  : Addttional item parameter strage area
                        as integer

```

(Note) Additional item parameter storage area can be specified as TYPE in following format

```

+ string
+ sbyte()      + integer ()   + long ()      + single()
+ byte ()      + uinteger ()  + ulong ()     + double()

```

## (2) SECS Message analysis

## (a) Initialize

```

function td._TDSMDMssgFind( // Return value
byval md      as integer,   // in  : Message identifier (Obtained by _TDSMDMssgInitialize())
byval mode    as integer,   // in  : Processing mode
byval msg()   as byte,      // in  : SECS Message storage area
byval len     as integer,   // in  : SECS Message storage area byte size
byval sf      as integer,   // in  : SF-Code to be acquired
byref mname   as string)    // out : Message name in message structure definition file
                        as integer

```

## (b) End processing

```

function td._TDSMDMssgExit( // Return value
byval md      as integer,   // in  : Message identifier (Obtained by _TDSMDMssgInitialize())
byval mode    as integer,   // in  : Processing mode                                (Must be =0)
byval msg()   as byte)      // in  : SECS Message storage area
                        as integer

```

## (c) Get parameter value of specified item

```

function td._TDSMDMssgNext( // Return value
byval md      as integer,   // in  : Message identifier (Obtained by _TDSMDMssgInitialize())
byval mode    as integer,   // in  : Processing mode                                (Must be =0)
byval msg()   as byte,      // in  : SECS Message storage area
byval iname   as string,    // in  : Item name of acquisition target
byval nop     as integer,   // in  : Number of items parameters that can be acquire
byval item    as TYPE)      // out : Item parameter storage area
                        as integer

```

(Note 1) Additional item parameter storage area can be specified as TYPE in following format

```

+ string      (Note 2) In case of string, pass 'byref'.
+ sbyte()    + integer ()  + long ()    + single()
+ byte ()    + uinteger () + ulong ()  + double()

```

## (3) Create reply SECS message for received message

## (a) Create reply message for received message

```

function td._TDSMDMssgAutoRes(    // Return value
byval md      as integer,    // in  : Message identifier (Obtained by _TDSMDMssgInitialize())
byval mode    as integer,    // in  : Processing mode                      (Must be =0)
byval rhd()   as byte,       // in  : Received SECS Message header
byval msg()   as byte,       // i/o : SECS Message storage area          (Size is 'xlen')
byval rlen    as integer,    // in  : Byte length of received SECS message
byval xlen    as integer,    // in  : Byte size of SECS message storage area (msg)
byref rname   as string,     // out : Received message name
byref sname   as string,     // out : Message name for reply
byref sf      as integer)    // out : Reply SF-Code
                                as integer

```

## D. SECS/HSMS Communication library (Java version) API specification

- (1) SECS/HSMS Message communication function (usual API)
- (2) SECS/HSMS Message communication function (abbreviated API)
- (3) SECS Message construction, analysis function
- (4) SECS Message construction and analysis function using SML format message structure definition

(Note 1) Each function shown in this chapter has the following attributes

```
+ package    : TDL.TDS
+ class      : TDJVS
```

Therefore, to use this library, write the following import statement

```
import static TDL.TDS.TDJVS.*
```

(Note 2) Refer to Chapter 3 for processing of each function, parameter details, etc.

(Note 3) Build and execution using this library requires the following. All should be pre-installed on the system by downloading from the official site etc.

```
+ JavaVM 1.6 or later (32bit/64bit)
+ JDK    1.6 or later (32bit/64bit)
+ JNA     4.1 or later (32bit/64bit)
```

(Note 4) ) This library realizes its function by Java JNA. Therefore, include JNA execution .jar after "jna-4.1.0.jar" and this package TDJVS.jar in CLASSPATH. Also, place following shared library in folder that passes the PATH, as following will be used as Native library.

```
+ Windows : TDS.dll          (PATH)
+ HP-UX    : libTDS.sl       (SHLIB_PATH)
+ Linux    : libTDS.so       (LD_LIBRARY_PATH)
+ FreeBSD  : libTDS.so       (LD_32_LIBRARY_PATH)
+ MacOS X  : libTDS.dylib    (DYLD_LIBRARY_PATH)
```

(Note 5) Be aware of JNA compatible architecture (32bit or 64bit) and make shared library (TDS.dll, libTDS.so, libTDS.dylib, libTDS.sl) architecture (32bit or 64bit) compatible.

(Note 6) When operating this library (DLL) in Java/JNA environment, it is necessary to specify sufficient size for the stack size to be used. For details, see the explanation of THSTACKUSR and THSTACKSYS in Section 2.1.

## D.1 SECS/HSMS Message communication function (usually API)

In the following explanation, refer to corresponding API described in section 3.1 for return value, parameter contents, etc.

## (1) SECS/HSMS communication : Open processing

```
int _TDJVSCommOpen(    // Return value
int      mode,        // in  : Processing mode
String   ini,         // in  : .ini file path name
String   sect)        // in  : .ini file section name to use
```

(Note 1) It is not able to set Callback function like C function.

If you need Callback function, you need to perform Callback processing on the AP side with the following code, for example.

+ Start thread for Callback function

```
RecvProcThread th;
th=new RecvProcThread(fd);
th.start();
```

+ Callback thread class

```
class RecvProcThread extends Thread{
private int fd;
public RecvProcThread(int fd)
{
    this.fd=fd;
}
public void run()
{
    byte      msg[]=new byte[512],hd[]=new byte[12];
    int      devid[]=new int[1],sf[]=new int[1],xid[]=new int[1],rtn,req;
    for(;;){
        req=0;
        if((rtn=_TDJVSCommRecv(fd,0,devid,sf,xid,msg,512,hd))==(-951)){
            Thread.sleep(100);
        }else{
            if((-1000)<rtn && rtn<(-959)) req=(-rtn)-900;
            CBRecvProc(req,rtn,devid[0],sf[0],xid[0],hd,msg);
        }
    }
}
```

+ Callback processing function

```
static private int
CBRecvProc(int req, int rtn, int devid, int sf, int xid, byte thd[], byte msg[])
{
    // Callback processing
    // The parameters are same as each member variable of structure TDSCBData shown
    // in 2.2(3).
}
```



## (2) SECS/HSMS communication : Close processing

```

int _TDJVSClose(      // Return value
int      fd,          // in : Control identifier      (Obtained by _TDJVSCloseOpen())
int      mode)        // in : Processing mode          (Must be =0)

```

## (3) SECS/HSMS communication : Receive processing

```

int _TDJVSCloseRecv( // Return value
int      fd,          // in : Control identifier      (Obtained by _TDJVSCloseOpen())
int      mode,        // in : Processing mode
int      devid[],     // out : Received message Device ID
int      sf [],       // out : Received message SF-Code
int      xid [],      // out : Received message Transaction ID
byte     msg [],      // out : Received message Body   (Data part)
int      len,         // in : Receiveable message length (Number of bytes)
byte     hd [])       // out : Received message Header

```

## (4) SECS/HSMS communication : Send processing

```

int _TDJVSCloseSend( // Return value
int      fd,          // in : Control identifier      (Obtained by _TDJVSCloseOpen())
int      mode,        // in : Processing mode
int      devid,       // in : Send message Device ID
int      sf,          // in : Send message SF-Code
int      xid,         // in : Send message Transaction ID
byte     msg [],      // in : Send message Body      (Data part)
int      len,         // in : Send message length    (Number of bytes)
byte     hd [])       // out : Send message Header   (Available only ((mode&0x0f00)==0))

```

## (5) SECS/HSMS communication : Illegal message receiving result sending process

```

int _TDJVSCloseSendError( // Return value
int      fd,          // in : Control identifier      (Obtained by _TDJVSCloseOpen())
int      mode,        // in : Processing mode          (Must be =0)
int      devid,       // in : Send message Device ID
int      sf,          // in : Send message SF-Code
byte     rhd [])      // in : Target bad header of S9Fx (Only significant for S9Fx)

```

## (6) Connection status check

```

int _TDJVSCommStatus( // Return value
int      fd,          // in : Control identifier      (Obtained by _TDJVSCommOpen())
int      mode)        // in : Processing mode          (Must be =0)

```

## (7) Checking connection status for each session ID (HSMS-GS)

```

int _TDJVSCommSelectStatus( // Return value
int      fd,                // in : Control identifier      (Obtained by _TDJVSCommOpen())
int      mode,              // in : Processing mode          (Must be =0)
int      devid)             // in : Session ID (Device ID)

```

## (8) Additional output of user comment messages to communication trace file

```

int _TDJVSCommUserComment( // Return value
int      fd,                // in : Control identifier      (Obtained by _TDJVSCommOpen())
int      mode,              // in : Processing mode          (Must be =0)
String   comm)              // in : Comment message to output

```

## D. 2 SECS/HSMS Message communication function (abbreviated API)

In the following explanation, refer to corresponding API described in section 3.1 for return value, parameter contents, etc.

## (1) SECS/HSMS communication : Open processing

```
int _TDJVSUDrvOpen(    // Return value
int    mode,          // in  : Processing mode
String *ini,          // in  : .ini file path name
String *sect,         // in  : .ini file section name to us
int    mask)          // in  : Mask of acquisition target even
```

## (2) SECS/HSMS communication : Close processing

```
int _TDJVSUDrvClose(    // Return value
int    fd,              // in  : Control identifier      (Obtained by _TDJVSUDrvOpen())
int    mode)            // in  : Processing mode        (Must be =0)
```

## (3) SECS/HSMS communication : Receive processing

```
int _TDJVSUDrvRecv(    // Return value
int    fd,              // in  : Control identifier      (Obtained by _TDJVSUDrvOpen())
int    mode,            // in  : Processing mode
int    devid[],         // out : Received message Device ID
int    sf [],           // out : Received message SF-Code
int    xid [],          // out : Received message Transaction ID
byte    msg [],         // out : Received message Body    (Data part)
int    len,             // in  : Receiveable message length (Number of bytes)
byte    hd [])          // out : Received message Header
```

(Note 1) Refer to 'D. 1 (3) (Note 1)'

## (4) SECS/HSMS communication : Send processin

```
int _TDJVSUDrvSend(    // Return value
int    fd,              // in  : Control identifier      (Obtained by _TDJVSUDrvOpen())
int    mode,            // in  : Processing mode
int    devid,           // in  : Send message Device ID
int    sf,              // in  : Send message SF-Code
int    xid,             // in  : Send message Transaction ID
byte    msg [],         // in  : Send message Body    (Data part)
int    len,             // in  : Send message length    (Number of bytes)
byte    hd [])          // out : Send message Header    (Available only ((mode&0x0f00)==0))
```

## (5) SECS/HSMS communication : Illegal message receiving result sending process

```

int _TDJVSUDrvSendError( // Return value
int      fd,           // in : Control identifier      (Obtained by _TDJVSUDrvOpen())
int      mode,         // in : Processing mode          (Must be =0)
int      devid,        // in : Send message Device ID
int      sf,           // in : Send message SF-Code
byte     rhd [ ])      // in : Target bad header of S9Fx  (Only significant for S9Fx)

```

## (6) Connection status check

```

int _TDJVSUDrvStatus( // Return value
int      fd,           // in : Control identifier      (Obtained by _TDJVSUDrvOpen())
int      mode)         // in : Processing mode          (Must be =0)

```

## (7) Checking connection status for each session ID (HSMS-GS)

```

int _TDJVSUDrvSelectStatus(// Return value
int      fd,           // in : Control identifier      (Obtained by _TDJVSUDrvOpen())
int      mode,         // in : Processing mode          (Must be =0)
int      devid)        // in : Session ID (Device ID)

```

## (8) Additional output of user comment messages to communication trace file

```

int _TDJVSUDrvUserComment(// Return value
int      fd,           // in : Control identifier      (Obtained by _TDJVSUDrvOpen())
int      mode,         // in : Processing mode          (Must be =0)
String   comm)         // in : Comment message to output

```

## D.3 SECS Message construction, analysis function

In the following explanation, refer to corresponding API described in section 3.1 for return value, parameter contents, etc.

## (1) SECS message construction

## (a) Initialize

```

int _TDJVSMmsgInit(    // Return value
int      mode,         // in  : Processing mode
byte     msg [],       // out : SECS Message storage area
int      len,         // in  : SECS Message storage area byte size
int      fdc)         // in  : SECS communication control identifier (Already opened)
                        // or Maximum SECS message byte size to process.

```

## (b) End processing

```

int _TDJVSMmsgEnd(    // Return value
int      md,          // in  : Message identifier (Obtained by _TDJVSMmsgInit())
int      mode,         // in  : Processing mode (Must be =0)
byte     msg [])      // i/o : SECS Message storage are

```

## (c-1) Add data item (usual format)

```

int _TDJVSMmsgBuild(  // Return value
int      md,          // in  : Message identifier (Obtained by _TDJVSMmsgInit())
int      mode,         // in  : Processing mode (Must be =0)
byte     msg [],       // i/o : SECS Message storage are
int      form,         // in  : Additional item Format code
int      nop,          // in  : Additional item number of parameter
TYPE     item)         // in  : Additional item parameter storage area

```

(Note) Additional item parameter storage area can be specified as TYPE in following format  
 When specifying ByteBuffer, data in a format that conforms to form must be stored in  
 ByteOrder.nativeOrder(). If specified in any other form, it must be in the form of a  
 variable that conforms to form.

```

+ ByteBuffer  + String
+ byte []     + float[]   + double[]
+ short[]     + int []    + long []

```

## (c-2) Add data item (string format)

```

int _TDJVSMmsgBuildL( // Return value (Refer to (c-1))
int      md,          // in  : Message identifier (Obtained by _TDJVSMmsgInit())
int      mode,         // in  : Processing mode
byte     msg [],       // i/o : SECS Message storage area
String   str)          // in  : SECS data items in given string format

```

## (2) SECS Message analysis

## (a) Initialize

```

int _TDJVSMssgFind(    // Return value
int      mode,        // in  : Processing mode
byte     msg [],       // in  : SECS Message storage area
int      len,         // in  : SECS Message storage area byte size
int      fdc,         // in  : SECS communication control identifier (Already opened)
                        //      or Maximum SECS message byte size to process.
byte     hd [],       // in  : SECS Message Header
StringBuffer mname)    // out : SECS Message name

```

## (b) End processing

```

int _TDJVSMssgExit(    // Return value
int      md,          // in  : Message identifier (Obtained by _TDJVSMssgFind())
int      mode,        // in  : Processing mode (Must be =0)
byte     msg [])       // in  : SECS Message storage area

```

## (c-1) Data item acquisition (usual format)

```

int _TDJVSMssgNext(    // Return value
int      md,          // in  : Message identifier (Obtained by _TDJVSMssgFind())
int      mode,        // in  : Processing mode (Must be =0)
byte     msg [],       // in  : SECS Message storage area
int      form [],      // out : Item Format code
int      parl [],      // out : Byte size occupied by one item
int      noi [],       // out : Number of items parameters
ByteBuffer item,       // out : Item parameter value storage area
int      xlen)         // in  : Byte size of item parameter value storage area

```

(Note) Acquired item value is stored in item by `ByteOrder.nativeOrder()`. In AP, item value is acquired by `item.array()`, `item.getInt()` etc according to `form[0]` value.

In the case of a string, it is acquired by `String str=new String(item.array(),0,noi[0]);`

## (c-2) Data item acquisition (string format)

```

int _TDJVSMssgNextL(    // Return value
int      md,          // in  : Message identifier (Obtained by _TDJVSMssgFind())
int      mode,        // in  : Processing mode
byte     msg [],       // in  : SECS Message storage are
int      form [],      // out : Item Format code
int      noi [],       // out : Number of items parameter
StringBuffer str)       // out : SECS data items of given string form. (up to 1000bytes)

```

## D. 4 SECS message construction and analysis function using SML format message structure definition

In the following explanation, refer to corresponding API described in section 3.1 for return value, parameter contents, etc.

## (0) Message structure definition information acquisition and release

## (a) Message structure definition information acquisition and initialization

```

int _TDJVSMDMssgInitialize(    //Return value
int      mode,                // in  : Processing mode
                                //      10
                                //      ----+-----+
                                //      +-+ 0:SML Form    2:<reserved>
                                //      1:<reserved>  3:<reserved>
int      fd,                  // in  : SECS communication control identifier (Already opened)
String   path)                // in  : Message structure definition file path name

```

## (b) Message structure definition information release

```

void _TDJVSMDMssgTerminate(
int      md,                  // in  : Message identifier (Obtained by _TDJVSMDMssgInitialize())
int      mode)                // in  : Processing mode (Must be =0)

```

## (1) SECS message construction

## (a) Initialize

```

int _TDJVSMDMssgInit( // Return value
int      md,          // in : Message identifier (Obtained by _TDJVSMDMssgInitialize())
int      mode,        // in : Processing mode (Must be =0)
byte     msg [],       // out : SECS Message storage area
int      len,         // in : SECS Message storage area byte size
String   mname)       // in : Message name in message structure definition file

```

## (b) End processing

```

int _TDJVSMDMssgEnd( // Return value
int      md,          // in : Message identifier (Obtained by _TDJVSMDMssgInitialize())
int      mode,        // in : Processing mode (Must be =0)
byte     msg [])       // i/o : SECS Message storage area

```

## (c) Data item parameter value setting

```

int _TDJVSMDMssgBuild( // Return value
int      md,           // in : Message identifier (Obtained by _TDJVSMDMssgInitialize())
int      mode,         // in : Processing mode (Must be =0)
byte     msg [],       // i/o : SECS Message storage area
String   iname,        // in : Additional item name
int      nop,          // in : Additional item number of parameters
TYPE     item)         // in : Additional item parameter storage area

```

(Note) Additional item parameter storage area can be specified as TYPE in following format.

If ByteBuffer is specified, data in a format that matches the format of item specified by 'iname' should be stored in `ByteOrder.nativeOrder()`. If specified in any other format, it must be in the form of a variable that matches the format of item specified by 'iname'.

|              |            |             |
|--------------|------------|-------------|
| + ByteBuffer |            | + String    |
| + byte []    | + float [] | + double [] |
| + short []   | + int []   | + long []   |



## (2) SECS Message analysis

## (a) Initialize

```

int _TDJVSMDMssgFind(    // Return value
int      md,             // in  : Message identifier (Obtained by _TDJVSMDMssgInitialize())
int      mode,           // in  : Processing mode
byte     msg [],         // in  : SECS Message storage area
int      len,            // in  : SECS Message storage area byte size
int      sf,             // in  : SF-Code to be acquired
StringBuffer mname)      // out : Message name in message structure definition file

```

## (b) End processin

```

int _TDJVSMDMssgExit(    // Return value
int      md,             // in  : Message identifier (Obtained by _TDJVSMDMssgInitialize())
int      mode,           // in  : Processing mode (Must be =0)
byte     msg [])         // in  : SECS Message storage area

```

## (c) Get parameter value of specified item

```

int _TDJVSMDMssgNext(    // Return value
int      md,             // in  : Message identifier (Obtained by _TDJVSMDMssgInitialize())
int      mode,           // in  : Processing mode (Must be =0)
byte     msg [],         // in  : SECS Message storage area
String    iname,         // in  : Item name of acquisition target
int      nop,            // in  : Number of items parameters that can be acquired
TYPE     item)           // out : Item parameter storage area

```

(Note) The item parameter storage area can be specified as TYPE in the following format.

If ByteBuffer is specified, ByteOrder.nativeOrder() stores data in a format that matches the format of the item specified by 'iname'. When specifying in other form, specify variable form variable that matches the form of item specified by 'iname'.

|              |                |            |
|--------------|----------------|------------|
| + ByteBuffer | + StringBuffer | + String[] |
| + byte []    | + float[]      | + double[] |
| + short[]    | + int []       | + long []  |

## (3) Create reply SECS message for received message

## (a) Create reply message for received message

```

int _TDJVSMDMssgAutoRes( // Return value
int      md,             // in : Message identifier (Obtained by _TDJVSMDMssgInitialize())
int      mode,           // in : Processing mode (Must be =0)
byte     rhd [],         // in : Received SECS Message header
byte     msg [],         // i/o : SECS Message storage area (Size is 'xlen')
int      rlen,           // in : Byte length of received SECS message
int      xlen,           // in : Byte size of SECS message storage area (msg)
StringBuffer rname,      // out : Received message name
StringBuffer sname,      // out : Message name for reply
int      sf [])          // out : Reply SF-Code

```

## E. SECS/HSMS library (Function limited Java (not using JNA) version) API specification

Describes Java API communication library that restricts functions of TDS that only operates in Java execution environment without using JNA and C language libraries.

This library has the following limited functions compared to C language version TDS.

1. SECS-1 and HSMS-GS communication not possible, supports only HSMS-SS communication.
2. The address format in HSMS-SS supports only IPV4.
3. The operation mode supports only the operation as a thread in user AP.
4. Communication data queue supports only local memory table.
5. It is not possible to receive processing that has registered the Callback function.
6. There is no facility to build SECS messages from SECS message text strings with MssgBuildL().
7. The code system conversion function can not be used for multibyte text string (022) in the SECS-2 message item format.
8. The use of message definition files is not possible and affects the following:
  - + Automatic response function for secondary messages, etc.
  - + Unable to display message name and item name in trace file
9. Even if the configuration file (.ini file) is changed after startup (CommOpen(), UDrvOpen()), the change can not be reflected in operation.
10. Trace output is only output to the communication trace file.
11. There is no function to output user comments to the communication trace file.

(0) SECS/HSMS Communication parameter configuration file (.ini file) format

(1) SECS/HSMS Message communication function (usual API)

(2) SECS/HSMS Message communication function (abbreviated API)

(3) SECS Message construction, analysis function

(Note 1) Each function shown in this chapter has the following attributes.

```
+ package   : TDL.TDS
+ class     : TDSComm ... SECS/HSMS Communication (Usual API)
              TDSUDrv ... SECS/HSMS Communication (abbreviated API)
              TDSMssg ... SECS-2 Message structure construction, analysis
```

Therefore, describe the following import statement when using this library.

```
import TDL.TDS.TDSComm;
import TDL.TDS.TDSUDrv;
import TDL.TDS.TDSMssg;
```

(Note 2) Build and execution using this library requires following. All should be pre-installed on the system by downloading from the official site etc.

```
+ JavaVM 1.6 or later
+ JDK    1.6 or later
```

## E. 0 SECS/HSMS Communication parameter configuration file (.ini file) format

Among the descriptions in the configuration file (.ini) shown in Section 2.1, valid tokens are shown below. See section 2.1 for details.

```

SECSMODE = 0xffff // SECS communication parameter
//          98 7654 3210
// +---+---+---+---+
//      || ||| |||+--- Communication      (0:<reserved> 1:HSMS      )
//      || ||| |||   Type                  (2:<reserved> 3:<reserved> )
//      || ||| |+--- HSMS Passive IP-Address force open
//      || ||| |      (0:No                1:Yes          )
//      || ||| +----- HSMS Address type   (0:IPV4         1:<reserved> )
//      || |||+----- Device type          (0:Host         1:Equipment  )
//      || ||+----- <reserved>
//      || |+----- HSMS Connection type  (0:Passive      1:Active      )
//      || +----- HSMS Resolve host name (0:OFF         1:<reserved> )
//      |+--- HSMS Passive side accept priority
//      +--- The meaning of the 15th bit of HSMS session

DEVMODE = 0xffff // Device control mode
// F C BA98 7654 3210
// +---+---+---+---+
// | | ||| ||| |||+--- Device ID check
// | | ||| ||| |||+--- Processing when receiving a secondary message that
// | | ||| ||| |||   is not in receiving waiting state
// | | ||| ||| |+--- <reserved>
// | | ||| ||| +----- <reserved>
// | | ||| |||+----- Transaction management (User data)
// | | ||| |||+----- Transaction management (Control data)
// | | ||| |+----- End transaction in S9FX receiving
// | | ||| +----- In case of HSMS, Transaction ends upon receipt
// | | |||+----- T3Timeout report          (S9F9) Auto send
// | | |||+----- Undef. Device ID report   (S9F1) Auto send
// | | |+----- Data size over report      (S9F11) Auto send
// | | +----- When a secondary message is not received, etc.
// | | Abnormal data report (S9F7) Auto send
// | +----- Reject request at HSMS        Auto send
// +----- T6 Timeout occure                Auto shutdown

```

|          |              |                                                                    |              |
|----------|--------------|--------------------------------------------------------------------|--------------|
| XDEV     | = 9999       | // Maximum number of connected devices                             |              |
| DEVID    | = "999,0xff" | // Connected device ID                                             |              |
| XMSGSIZE | = 999999999  | // Maximum SECS message byte length                                |              |
| XTRANX   | = 999999     | // Maximum number of transactions to process simultaneously        |              |
| SRCID0   | = 99999      | // SourceID given to user AP sending message                       | ( 0 - 32767) |
| SRCID1   | = 99999      | // SourceID given to HSMS control message                          | ( 0 - 32767) |
|          |              |                                                                    |              |
| XIDMIN   | = 99999      | // Minimum value of transaction ID to be assigned                  | ( 1 - 65535) |
| XIDMAX   | = 99999      | // Maximum value of transaction ID to be assigned                  | ( 1 - 65535) |
|          |              |                                                                    |              |
| INTER0   | = 99900      | // Communication control unit processing interval                  | (unit : ms)  |
| INTER1   | = 99900      | // Overall control unit processing interval                        | (unit : ms)  |
| INTER2   | = 990        | // Serial communication receiving interval                         | (unit : ms)  |
| INTER3   | = 999        | // File related processing interval                                | (unit : s)   |
|          |              |                                                                    |              |
| HOST     | = "XXXXXXXX" | // HSMS TCP/IP Destination host name or IP-Address                 |              |
| PORT     | = 99999      | // HSMS TCP/IP Connection port number                              | (1 - 65535)  |
| LINKINT  | = 9999       | // HSMS Link test execution interval                               | (Unit : s)   |
| T3       | = 999        | // T3 Timeout (Message reply)                                      | (Unit : s)   |
| T4       | = 999        | // T4 Timeout (Inter block)                                        | (Unit : s)   |
| T5       | = 999        | // T5 Timeout (Connect, Shutdown)                                  | (Unit : s)   |
| T6       | = 999        | // T6 Timeout (Control transaction)                                | (Unit : s)   |
| T7       | = 999        | // T7 Timeout (NOT SELECT)                                         | (Unit : s)   |
| T8       | = 999        | // T8 Timeout (Inter network packet)                               | (Unit : s)   |
| TORECV   | = 99999999   | // Time to disconnect due to no message received                   | (Unit : s)   |
|          |              |                                                                    |              |
| QUEREC   | = 9999       | // Send/receive queue file storage directory                       |              |
| QUEMODE  | = 0xff       | // Processing mode of SECS/HSMS communication message send/receive |              |

```

TRCDIR      = "XXXXXXXX" // Trace file storage directory
TRCTTYPE    = 0xffff      // Communication message output format to communication trace
// BA98 7654 3210
// +---+---+---+
// ||++ |+++ |||+-- List format output (0:No 1:Yes)
// || | | | | | +--- Hexadecimal format output (0:No 1:Yes)
// || | | | | | +--- Item data display format
// || | | | | | (0:1Line 1:Multiple lines)
// || | | | | | +--- Hexadecimal display
// || | | | | | (0:16Bytes/line 1:20Bytes/line )
// || | | | +----- List output format
// || | | | (0:TDS 2:SML 1,3-7:<reserved>)
// || | +----- <reserved>
// || +----- <reserved>
// | +----- Number of leading spaces in list (0:2 1:0 )
// +----- <reserved>

TRCTOUT     = 0xffff      // Communication trace output mode
// D 8 765 10
// +---+---+---+---+
// +-+---+ |++ +--- Trace output destination
// | | +----- <reserved>
// | | +- Trace file daily switch specification at output
// +----- Number of trace files to keep

TRCTLEVEL   = 9          // Communication trace output level
TRCTATTR    = 0xffff      // Communication trace output attribute (Usually not changeable)
// F 8 54 3210
// +---+---+---+---+
// | | | | | ++ Output time YYMMDD. hhmmss
// | | | | | ++ Output time (ms) .999
// | | | | | ++ <reserved>
// | | | | | ++ Trace output level :99
// | | | | | ++ <reserved>
// | | | | | ++ <reserved>
// | | | | | ++ Key string of trace line :XX
// +----- =0: Make end of line only LF
// 1: Make end of line CR/LF

TRCTSIZE    = 999999999 // Communication trace file size
TRCDELTIME  = hhmmss    // Old trace deletion execution time
TRCDELDAY   = YYMMDD    // Old trace deletion target progress date

```

## E.1 SECS/HSMS Message communication function (usually API)

In the following explanation, refer to corresponding API described in section 3.1 for return value, parameter contents, etc.

## (0) Create instance

```
TDSComm      Scb=new TDSComm();
```

## (1) SECS/HSMS communication : Open processing

```
int  Scb._TDSCommOpen(    // Return value
int    mode,              // in  : Processing mode
String ini,               // in  : .ini file path nam
String sect)              // in  : .ini file section name to use
```

(Note 1) It is not able to set Callback function like C function.

If you need Callback function, you need to perform Callback processing on the AP side with the following code, for example.

+ Start thread for Callback function

```
RecvProcThread th;
th=new RecvProcThread();
th.start();
```

+ Callback thread class

```
class RecvProcThread extends Thread{
public void run()
{
    byte    msg[]=new byte['Maximum message length'],hd[]=new byte[12];
    int     devid[]=new int[1],sf[]=new int[1],xid[]=new int[1],rtn,req;
    for(;;){
        req=0;
        if((rtn=Scb._TDSCommRecv(0,devid,sf,xid,msg,'Maximum message length',hd))==(-951)){
            Thread.sleep(100);
        }else{
            if((-1000)<rtn && rtn<(-959)) req=(-rtn)-900;
            CBRcvProc(req,rtn,devid[0],sf[0],xid[0],hd,msg);
            if(rtn<0) Thread.sleep(100);
        }
    }
}
```

+ Callback processing function

```
static private int
CBRcvProc(int req, int rtn, int devid, int sf, int xid, byte thd[], byte msg[])
{
    // Callback processing
    // The parameters are same as each member variable of structure TDSCBData shown
    // in 2.2(3).
}
```

## (2) SECS/HSMS communication : Close processing

```

int Scb._TDSClose( // Return value
int mode) // in : Processing mode (Must be =0)

```

## (3) SECS/HSMS communication : Receive processing

```

int Scb._TDSCRecv( // Return value
int mode, // in : Processing mode
int devid[], // out : Received message Device ID
int sf [], // out : Received message SF-Code
int xid [], // out : Received message Transaction ID
byte msg [], // out : Received message Body (Data part)
int len, // in : Receiveable message length (Number of bytes)
byte hd []) // out : Received message Header

```

## (4) SECS/HSMS communication : Send processing

```

int Scb._TDSCSend( // Return value
int mode, // in : Processing mode
int devid, // in : Send message Device ID
int sf, // in : Send message SF-Code
int xid, // in : Send message Transaction ID
byte msg [], // in : Send message Body (Data part)
int len, // in : Send message length (Number of bytes)
byte hd []) // out : Send message Header (Available only ((mode&0x0f00)==0))

```

## (5) SECS/HSMS communication : Illegal message receiving result sending process

```

int Scb._TDSCSendError( // Return value
int mode, // in : Processing mode (Must be =0)
int devid, // in : Send message Device ID
int sf, // in : Send message SF-Code
byte rhd [], // in : Target bad header of S9Fx (Only significant for S9Fx)
byte shd [], // out : Send message header
byte sdt []) // out : Send message body

```

## (6) Connection status check

```

int Scb._TDSCStatus( // Return value
int mode) // in : Processing mode (Must be =0)

```



## E. 2 SECS/HSMS Message communication function (abbreviated API)

In the following explanation, refer to corresponding API described in section 3.1 for return value, parameter contents, etc.

## (1) SECS/HSMS communication : Open processing

```
int Scb._TDSUDrvOpen(    // Return value
int      mode,           // in  : Processing mode
String   ini,            // in  : .ini file path nam
String   sect,           // in  : .ini file section name to us
int      mask)           // in  : Mask of acquisition target even
```

## (2) SECS/HSMS communication : Close processing

```
int Scb._TDSUDrvClose(    // Return value
int      mode)            // in  : Processing mode                                     (Must be =0)
```

## (3) SECS/HSMS communication : Receive processing

```
int Scb._TDSUDrvRecv(    // Return value
int      mode,           // in  : Processing mode
int      devid[],        // out : Received message Device ID
int      sf  [],         // out : Received message SF-Code
int      xid  [],        // out : Received message Transaction ID
byte     msg  [],        // out : Received message Body      (Data part
int      len,           // in  : Receiveable message length (Number of bytes)
byte     hd   [])        // out : Received message Header
```

(Note 1) Refer to 'D.1 (3) (Note 1)'

## (4) SECS/HSMS communication : Send processin

```
int Scb._TDSUDrvSend(    // Return value
int      mode,           // in  : Processing mode
int      devid,          // in  : Send message Device ID
int      sf,             // in  : Send message SF-Code
int      xid,            // in  : Send message Transaction ID
byte     msg  [],        // in  : Send message Body      (Data part)
int      len,           // in  : Send message length    (Number of bytes)
byte     hd   [])        // out : Send message Header    (Available only ((mode&0x0f00)==0))
```

## (5) SECS/HSMS communication : Illegal message receiving result sending process

```

int Scb._TDSUDrvSendError( // Return value
int      mode,           // in  : Processing mode                (Must be =0)
int      devid,          // in  : Send message Device ID
int      sf,             // in  : Send message SF-Code
byte     rhd [],         // in  : Target bad header of S9Fx      (Only significant for S9Fx)
byte     shd [],         // out : Send message header            (Refer to '2.2 (1) (Note 1)')
byte     sdt [],         // out : Send message body
//                      //      Make the area of sufficient size (16 bytes or more).

```

## (6) Connection status check

```

int Scb._TDSUDrvStatus( // Return value
int      mode)          // in  : Processing mode                (Must be =0)

```

## E.3 SECS Message construction, analysis function

In the following explanation, refer to corresponding API described in section 3.1 for return value, parameter contents, etc.

## (0) Create instance

```
TDSMssg      Mcb=new TDSMssg();
```

## (1) SECS message construction

## (a) Initialize

```
int  Mcb._TDSMssgInit(    // Return value
int      mode,           // in  : Processing mode                (Must be =0)
byte     msg [],          // out : SECS Message storage area
int      len)            // in  : SECS Message storage area byte size
```

## (b) End processing

```
int  Mcb._TDSMssgEnd(     // Return value
int      mode,           // in  : Processing mode                (Must be =0)
byte     msg [])         // i/o : SECS Message storage area
```

## (c-1) Add data item (usual format)

```
int  Mcb._TDSMssgBuild(   // Return value
int      mode,           // in  : Processing mode                (Must be =0)
byte     msg [],         // i/o : SECS Message storage area
int      form,           // in  : Additional item Format code
int      nop,            // in  : Additional item number of parameter
TYPE     item)           // in  : Additional item parameter storage area
```

(Note) Additional item parameter storage area can be specified as TYPE in following format.

When specifying ByteBuffer, data in a format that conforms to form must be stored in ByteOrder.nativeOrder(). If specified in any other form, it must be in the form of a variable that conforms to form.

```
+ ByteBuffer  + String
+ byte[]      + float[]      + double[]
+ short[]     + int[]        + long[]
```

## (c-2) Add data item (string format)

```
int  Mcb._TDSMssgBuildL( // Return value                (Refer to 3.3 (c-2))
int      mode,           // in  : Processing mode
byte     msg [],         // i/o : SECS Message storage area
String    str)           // in  : SECS data items in given string forma
```

## (2) SECS Message analysis

## (a) Initialize

```

int Mcb._TDSMssgFind(    // Return value
int      mode,          // in  : Processing mode
byte     msg [],        // in  : SECS Message storage area
int      len,           // in  : SECS Message storage area byte size
int      dmy,           // in  : Dummy parameter
byte     hd  [])        // in  : SECS Message Header

```

(Must be =0)

## (b) End processing

```

int Mcb._TDSMssgExit(    // Return value
int      mode,          // in  : Processing mode
byte     msg [])        // in  : SECS Message storage area

```

(Must be =0)

## (c-1) Data item acquisition (usual format)

```

int Mcb._TDSMssgNext(    // Return value
int      mode,          // in  : Processing mode
byte     msg [],        // in  : SECS Message storage area
int      form [],       // out : Item Format code
int      parl [],       // out : Byte size occupied by one item
int      noi [],        // out : Number of items parameters
ByteBuffer item,        // out : Item parameter value storage area
int      xlen)          // in  : Byte size of item parameter value storage area

```

(Must be =0)

(Note) Acquired item value is stored in item by `ByteOrder.nativeOrder()`. In AP, item value is acquired by `item.array()`, `item.getInt()` etc according to `form[0]` value.

In the case of a string, it is acquired by `String str=new String(item.array(),0,noi[0]);`

## (c-2) Data item acquisition (string format)

```

int Mcb._TDSMssgNextL(    // Return value
int      mode,          // in  : Processing mode
byte     msg [],        // in  : SECS Message storage area
int      form [],       // out : Item Format code
int      noi [],        // out : Number of items parameter
StringBuffer str)        // out : SECS data items of given string form.

```

(up to 1000bytes)