MOSCOW EXCHANGE

ASTS Connectivity API

Application programming interface for connecting external systems to the Moscow Exchange ASTS trading & clearing system

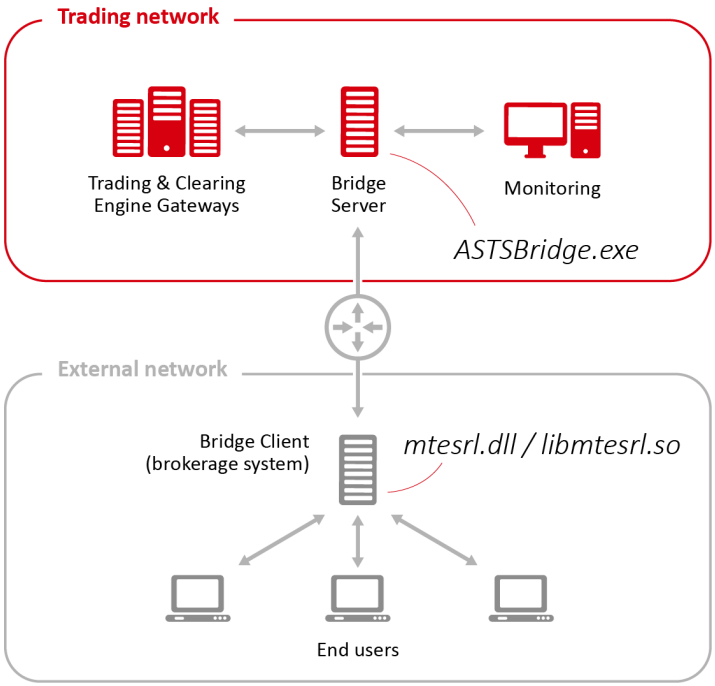
(MTESRL library v. 4.4)

TABLE OF CONTENTS

INTRODUCTION

ASTS Connectivity API should be used to connect any types of external systems to the Moscow Exchange ASTS trading & clearing system. These could be: brokerage systems, market data distribution systems, backoffice applications, HFTs and other client software.

System architecture is shown on the following diagram:



This document details creation of client applications using the ASTS Connectivity API. All the required functions are included into the MTESrl library.

MTESRL library

MTESrl library provides bidirectional connection to the ASTS trading & clearing system (TS) and contains functions for both receiving data from TS (general market data such as trades, quotes, financial instruments as well as company specific trading and clearing information) and executing transactions (order entry and withdrawal). Library supports all the Moscow Exchange markets powered by the ASTS platform:
Equity & bond, FX and precious metals, money (loans and deposits) markets.

HARDWARE AND SOFTWARE REQUIREMENTS

MTESrl library is compatible with the following operating systems:

- Windows 10 or Windows 2016/2019, 32 or 64 bit (`mtesrl[64].dll`);
- Linux OS family, 64 bit (`libmtesrl.so`). Note: The cdecl calling convention is used.

There are two versions of MTESRL library which differ in way of connection to TS:

- Connection to the trading system through ASTS Bridge (using TCP/IP protocol);
- Direct connection to the trading system. This version of library can only be used at the co-location facility.

Minimal hardware requirements for MTESRL:

- CPU: Intel Core or compatible 1.4GHz or higher. 3GHz x4 core is recommended.
- RAM – 4GB or more. 16GB is recommended.
- HDD with 10 GB free space for logging.
- Ethernet network card.

WORK SCENARIO

The typical work scenario is as follows:

1. Connect to the server.
2. Download the information object metadata (types, tables and transactions).
3. Open and refresh tables. Send transactions.
4. Save snapshots (optional).
5. Close the connection.

There are interface modules for library as well as MS Visual C, Java, Delphi and C# samples in the Demo subdirectory of installation folder.

CONNECTION TO THE SERVER

CONNECTING TO ASTS Bridge

`MTEConnect` function is used to connect to the trading/clearing system through the ASTS Bridge Server. This function should be called before proceeding to any other library functions.

C++

```
int32 WINAPI MTEConnect(char *Params, char *ErrorMsg);
```

Pascal

```
function MTEConnect(Params, ErrorMsg: LPSTR): Integer; stdcall;
```

Arguments:

Params

Connection parameters. This is a pointer to an ASCIIZ-string, which contains the list of parameters separated with the “end of line” and “carriage return” symbols (0x0D, 0x0A) with the following syntax:

```
Parameter1=Value1
Parameter2=Value2
...
ParameterN=ValueN
```

Denominations of parameters and their possible values depend on the method of connection of a specific library to the trading system. The following parameters are available for the MTESRL library:

CONNECTING TO ASTS BRIDGE

HOST	List of comma-separated IP addresses with ports of the ASTS Bridge server; for example: “194.186.240.85:20006,194.186.240.73:20006”.
PREFERREDHOST	Preferred host address. If not defined, server with the smallest number of users from the “HOST” list is used.
SERVER	Server ID, for example: “EQ_TEST”.
USERID	User ID in the trading/clearing system.
PASSWORD	User password in the trading/clearing system.
INTERFACE	Trading system interface ID. For example, “IFCBroker_26”.
BOARDS	List of trading boards that user is going to work with; for example:

	"TQBR,TQOB,PSEQ" (this is an optional parameter, if not defined, all boards are available).
COMPRESSION	Compression of transmitted data: "0" – no compression; "1" – ZLIB compression; "2" – compress large network packets with BZIP. 1 is used by default. Support for BZIP may be removed in future.

Encryption and digital signature "Validata" configuration

Signing - constant value:

Validata

Signing.ProfileName - cryptographic "Validata" library profile name (optional; if not defined then neither digital signature nor encryption will be used); the old **PROFILENAME** parameter is still supported;

The profile name must be prefixed by one the following:

xpki: to use qualified certificates (GOST) with Validata CSP version 5;
zpki: to use qualified certificates (GOST) with Validata CSP version 6;
rpki: to use not qualified certificates (RSA).

E.g.: if the digital signature profile is named DefaultGost, then Signing.ProfileName=zpki:DefaultGost parameter should be defined.

Signing.InitFlags - a combination of Validata initialization flags (optional):

1 - Do not update the list of recalled certificates (CRL) at initialization;
4 - Do not use network directories;

Signing.Type, Signing.BasePath and **Signing.LdapPath** - another method to initialize Validata. It could be useful when Validata has been installed with another user's system account – for example, when the client application is started as a service. In such situation there will be no profile name in user's registry branch and the ProfileName could not be used. Values for these parameters should be taken from the appropriate user's branch in the system registry:

Signing.Type - type of the crypto-provider being used:

Validata CSP - Validata CSP crypto-provider (zpki1.dll/xpki1.dll);
Microsoft CSP - Microsoft crypto-provider for non-residents (rpki1.dll);

Signing.BasePath - take file path from the registry key, corresponding to the profile (N = 0,1,2…):

HKEY_CURRENT_USER\Software\Validata\xpki\Profiles\<N>\store_0 (Validata CSP version 5)

HKEY_CURRENT_USER\Software\Validata\zpki\Profiles\<N>\store_0 (Validata CSP version 6)

HKEY_CURRENT_USER\Software\Validata\rpki\Profiles\<N>\store_0 (Microsoft CSP)

E.g., if «`pse://signed/C:\Users\Test\AppData\Roaming\VALIDATA\rcs\TEST_CRYPT\local.pse`» value is stored in registry key, you should specify «`C:\Users\Test\AppData\Roaming\VALIDATA\rcs\TEST_CRYPT\`» in this parameter.

Signing.LdapPath is stored in registry key:

HKEY_CURRENT_USER\Software\Validata\rpki\Profiles\<N>\store_2

Channel encryption configuration

ASTSBridge version 4.4.0 and higher supports TLS 1.2 channel encryption. Validata CSP version 6 and higher does not support channel encryption. Encryption and digital signing can be switched on/off separately.

Encrypt - channel encryption algorithm:

<*empty*> - do not use channel encryption;

OpenSSL - use TLS 1.2 for channel encryption;

Validata - use Validata CSP for channel encryption in packet mode. The Encrypt.ProfileName= parameter should be also specified. This mode is extremely slow and should not be used in production environment;

Any - if TLS 1.2 protocol is supported then use TLS 1.2. Otherwise, if Validata digital signature is turned on then use Validata CSP (either in channel or packet encryption mode). Otherwise, do not use channel encryption;

Parameter is not specified - if Validata digital signature is turned on and TLS 1.2 protocol is supported then use TLS 1.2, otherwise, use Validata CSP (either in channel or packet encryption mode). If Validata signature is turned off then do not use channel encryption.

It is recommended to use the Encrypt=Any parameter if channel encryption is required.

Encryption and digital signature «TUMAR» configuration

You need to install "Validata certificate store" software to work with TUMAR crypto-provider. You should configure two profiles there: one for digital signature and other for encryption. Refer to "Using TUMAR keys in ASTSBridge.pdf" manual for software installation and configuration instructions.

Signing, Encrypt - constant value:

Validata

Signing.ProfileName - «Validata» profile name for digital signature (optional, if not defined then digital signature will not be used).

Encrypt.ProfileName - «Validata» profile name for encryption (optional, if not defined then encryption will not be used).

rpki: prefix should be used before profile name in both parameters. E.g.: if the digital signature profile is named TUMAR_SIGN, then Signing.ProfileName=rpki:TUMAR_SIGN parameter should be defined.

Signing.InitFlags, Encrypt.InitFlags - a set of initialization flags (optional):

1 - Do not update the list of recalled certificates (CRL) at initialization;

Signing.Type, Signing.BasePath,

Encrypt.Type, Encrypt.BasePath - another method to initialize TUMAR. It could be useful when "Validata certificate store" software has been installed with another user's system account – for example, when the client application is started as a service. In such situation there will be no profile name in user's registry branch and the ProfileName could not be used. Values for these parameters should be taken from the appropriate user's branch in the system registry:

Signing.Type, Encryption.Type - constant value:

Microsoft CSP

Signing.BasePath, Encryption.BasePath - take file path from the registry key, corresponding to the profile (N = 0,1,2…):

HKEY_CURRENT_USER\Software\Validata\rpki\Profiles\<N>\store_0

E.g., if «`pse://signed/C:\Users\Test\AppData\Roaming\VALIDATA\rcs\TEST_CRYPT\local.pse`» value is stored in registry key, you should specify «`C:\Users\Test\AppData\Roaming\VALIDATA\rcs\TEST_CRYPT\`» in this parameter.

CONNECTING VIA «EMBEDDED» BRIDGE AT COLOCATION FACILITY

SERVER	Trading system server name, e.g., «GATEWAY».
SERVICE	Trading system service name, e.g., «gateway». Port numbers may be specified instead, e.g. 18011/18012.
BROADCAST	Broadcast address for the server search to access the trading system, e.g., «10.63.1.255,10.63.3.255,10.61.1.255,10.61.3.255».
PREFBROADCAST	Preferred broadcast server address.
USERID	Client user ID in trading/clearing system.
PASSWORD	User password in trading/clearing system.
INTERFACE	ID of the bridge interface to work with.
BOARDS	List of boards that the user is going to work with; for example: “TQBR,TQOB,PSEQ” (this is the optional parameter; if not defined, all boards are available).
CACHEFOLDER	Directory for caching interface description, downloaded from trading system. If this parameter is not defined, caching is not performed, and interface is downloaded from trading system at each connection.
LOGLEVEL	Level of internal logging: “0” – logging is disabled (default value); “1“ – “30“ – logging level.
COMPRESSION	Compression: “0” – no compression; “1” – compression is enabled (default value).
IPSRCORDER	List of IP addresses of network interfaces that are allowed to connect to Trading System. The order of IP addresses in the list defines the priority. If RestrictList=0, connection attempts from all other addresses are allowed, but with a lower priority. If RestrictList=1, only attempts from specified addresses are available, e.g.“192.168.126.1, 192.168.56.1”.
RESTRICTLIST	“0“ – searching for gateways is allowed from all available network interfaces (default value); “1“ – searching for gateways is allowed only from interfaces, listed in IpSrcOrder attribute.
DIRECTCONNECT	“0” – use server UDP discovering (default value); “1” - do not use server UDP discovering, connect directly to *Broadcast* addresses via TCP.

ALSO, IN ALL OF THE CASES THE FOLLOWING PARAMETERS ARE SUPPORTED:

TIMEOUT	Server (i.e. trading system) request execution timeout. For mtesrl.dll – in milliseconds, for embedded mtesrl.dll – in seconds. Default value is 30 seconds. If reply from server is not received within specified time, the reconnection procedure will be initiated. If connection interrupt is registered before the timeout expire – reconnection procedure will begin earlier.
LOGGING	String in the format “N,M”, where first digit “N” – API MTESRL calls

	logging level. “0” – no logging (do not create log-file); “1” – log errors only; “2” – log library function calls; “3” – log contents of table; “4” – log contents of table and field numbers; “5” - log TSMR protocol messages (only for embedded version). Second digit “M” – connection statistics logging level. Statistics is stored in a separate file formatted «mtesrl-YYYMMDD-<userid>-stats.log». “0” – do not collect statistics; “1” – collect statistics on query execution time and the trading system response size; “2” - Collect statistics and requests distribution on requests to the tables. Default value for logging is 2,2. For a complete logging disabling, use “LOGGING=0,0” Log files are kept for 7 calendar days. All the older logs are deleted when the MTEConnect function is called.
KEEPLOGS	Number of days to keep the log-files (7 by default). Value “0” means do not delete old log-files.
RETRIES	Number of attempts to reconnect after the loss of connection with ASTS Bridge Server (10 by default).
CONNECTTIME	Maximum reconnect time. For mtesrl.dll – in milliseconds, for embedded mtesrl.dll – in seconds. Default is 1 minute. Any value between 5 and 300 sec. can be specified. Reconnection lasts not more than [RETRIES] attempts and no longer than [CONNECTTIME] ms, depending on which event comes first. This value is approximate and may differ from a real one for several seconds.
LOGFOLDER	A folder to store the log files. By default, library folder is used.
LOGPREFIX	Specify unique string prefix to differentiate log-file names when connecting to several trading systems with the same user identifier.
FEEDBACK	Free formatted text string, describing the client system, connected to the bridge. For example, «FondAnalytic v3.5.456, e-mail: admin@fondru.ru».
LANGUAGE	Specify the language for messages issued by the Bridge and MTESRL client library. To change the language use transaction CHANGE_LANGUAGE. Possible values are “Russian” and “English”.
TRANSPORT	Transport library name: TSMR or Mustang. If not specified, TSMR is used.
JUMBOSIZE	The parameter can be used with Mustang transport only. Turns on large data packets received from Trading System. “0” – 60000 (use standard packet size), default value; “1” – 128КБ; “2” – 256КБ; “3” – 512КБ.

ErrorMsg

A pointer to a buffer of at least 256 bytes to store error description, in case an error occurs.

Returned value:

If connection is successful, the function returns a descriptor of the established connection (value that is greater or equal to MTE_OK). The received connection descriptor is used during execution of all MTExxxx functions.

If error occurs, one of the MTE_xxxx error codes is returned and error description is placed to ErrorMsg argument.

Example:

Connect to the ASTS Bridge server.

C++

```
int32 Idx;
char ErrorMsg[255];
...
Idx = MTEConnect("HOST=192.168.0.10:15005\rSERVER=EQ_TEST\r
USERID=MU0000100001\rINTERFACE=IFCBroker_26", ErrorMsg);
if( Idx < MTE_OK )
{
fprintf(stderr, "Error while establishing the connection: %s",
ErrorMsg);
      exit(1);
}
else
  fprintf(stdout, "Connection established.");
```

Pascal

```
Idx: Integer;
ErrorMsg: TMTEErrorMsg;
...
Idx := MTEConnect('HOST=192.168.0.10:15005'#13#10'SERVER=EQ_TEST'
#13#10'USERID=MU0000100001'#13#10'INTERFACE=IFCBroker_26',
@ErrorMsg);
if Idx < MTE_OK then
begin
  Writeln(Error while establishing the connection: ' + ErrorMsg);
  Halt;
end
else
  Writeln('Connection established.');
```

SELECTING THE LIST OF BOARDS

Usually, the list of boards is defined in "BOARDS=" parameter, when calling `MTEConnect` function. But also can be selected later, using `MTESelectBoards`. It's allowed to use only one method of these two for selecting boards. After calling `MTESelectBoards`, close all the tables and open them again, because all the tables content depends on selected boards.

C++

```
int32 WINAPI MTESelectBoards(int32 Idx, char * BoardsList,
                        char *result);
```

Pascal

```
function MTESelectBoards(Idx: Integer; BoardList: LPSTR;
                        ResultMsg: LPSTR): Integer; stdcall;
```

Аргументы:

Idx

A descriptor of connection, for which, the data should be received.

BoardList

A pointer to the string, containing a list of boards' identifiers, separated by comma. For example, "TQBR,TQNE,RPMA".

ResultMsg

A pointer to a buffer of at least 256 bytes to store the text string with transaction result in case of successful execution.

Returned value:

If the transaction has been processed by the trading system, it returns the following:

MTE_OK – boards selected;
MTE_TRANSREJECTED - request has been processed, but rejected by the trading system (an invalid board specified, no rights to perform, etc.);
MTE_TSMR - fatal error occurred, when executing the query (the loss of connection with the trading system, etc.).

A text string with query result is placed into *ResultMsg* argument.

If error occurs, one of the MTE_xxxx error codes is returned. In this case a value of *ResultMsg* is not defined.

GETTING SYSTEM AND SERVICE INFORMATION

BRIDGE SERVER DETAILS

To get additional details about the server side of ASTS Bridge, use `MTEGetServInfo` function.

C++

```
int32 WINAPI MTEGetServInfo(int32 Idx, char ** ServInfo, int *Len);
```

Pascal

```
function MTEGetServInfo(Idx: Integer; var ServInfo: LPSTR;
                        var Len: Integer): Integer; stdcall;
```

Arguments:

Idx

A descriptor of connection, for which, the data should be received.

ServInfo

A pointer to a buffer to store returned values.

Len

A pointer to a variable to store the length of returned data.

Returned value:

If successful, MTE_OK is returned and ServInfo points to a buffer of the following structure:

Field	**Data type (IBM PC)**	**Length, bytes**	**Description**
Connected_To_ASTS	INTEGER	4	Connection status. Possible values: 0 – not connected; 1 - connected to production environment; 2 - connected to test environment; -1 – connected to production environment, test trading session is in progress.
Session_Id	INTEGER	4	Current trading session internal ID. Changes each session.
ASTS_Server_Name	CHAR	33	Access server logical name. For example, GATEWAY, FOND_GATEWAY, etc. Can be used to identify a market and type of the system (test or production).
Version_Major	CHAR	1	ASTS Bridge major version number.
Version_Minor	CHAR	1	ASTS Bridge minor version number.
Version_Build	CHAR	1	ASTS Bridge build number. This and two previous fields identify the version as Major.Minor.Build.
Beta_version	CHAR	1	ASTS Bridge beta version flag. If not 0, then, this is beta version with a corresponding number.
Debug_flag	CHAR	1	ASTS Bridge debug version flag. If not 0, then this is a debug version.

Test_flag	CHAR	1	ASTS Bridge test release flag. If not 0 then, this is a test version.
Start_Time	INTEGER	4	Session start time (defined in the Bridge configuration). Specified as HHMMSS. Note that this is the integer.
Stop_Time_Min	INTEGER	4	Bridge shutdown time (defined in the Bridge configuration). Specified as HHMMSS. Note that this is the integer.
Stop_Time_Max	INTEGER	4	Equals to Stop_Time_Min.
Next_Event	INTEGER	4	Next expected event in the server schedule. Possible values: 0 – waiting for a new trading session startup; 1 – waiting for a current trading session end.
Event_Date	INTEGER	4	Date of an expected event as DDMMYYYY. Note that this is the integer.
BoardsSelected	ASCIIZ string	variable	Comma separated list of selected trading boards.
UserID	CHAR, null terminated string	13	User ID used by the server for current connection.
SystemId	CHAR	1	Trading system type: “P” – equities & bonds or money market; “C” – FX market; “F” – derivatives market.
ServerIp	ASCIIZ string	variable	Gateway IP, e.g., «195.1.3.51».

If error occurs, one of the MTE_xxxx error codes is returned.

BRIDGE CLIENT LIBRARY VERSION

`MTEGetVersion` function is used to get the client library version number.

C++

```
char * WINAPI MTEGetVersion();
```

Pascal

```
function MTEGetVersion: LPSTR; stdcall;
```

Arguments:

none

Returned value:

A pointer to an ASCIIZ string containing a text description of client library version. For example: “MTESrl library 3.8.93”.

GETTING CONNECTION STATUS

To obtain the current status of connection to ASTS Bridge server, `MTEConnectionStatus` function should be used.

C++

```
int32 WINAPI MTEConnectionStatus(int32 Idx);
```

Pascal

```
function MTEConnectionStatus(Idx: Integer): Integer; stdcall;
```

Arguments:

Idx

A descriptor of connection, for which, the data should be received.

Returned value:

One of the following MTE_xxx codes:

MTE_OK	Connection established.
MTE_INVALIDCONNECT	Invalid connection descriptor.
MTE_SRVUNAVAIL	ASTS Bridge server is not available.
MTE_TEUNAVAIL	Trading system is not available.

GETTING CONNECTION STATISTICS

To obtain a statistical data on the connection (connection flags, amount of transferred data, etc.) `MTEConnectionStats` function can be used.

C++

```
int32 WINAPI MTEConnectionStats(int32 Idx, ConnectionStats * Stats);
```

Pascal++

```
function MTEConnectionStats(Idx: Integer; var Stats: TMTEConnStats):
            Integer; stdcall;
```

Idx

A descriptor of connection, for which, the data should be received.

Returned value:

In case of success function returns MTE_OK and fills the Stats structure by statistical data on the connection. Stats structure has the following format:

Size	int32	Input field, must be filled `sizeof(Stats)`.
Properties	uint32	Connection flags, combination of values ZLIB_COMPRESSED, FLAG_ENCRYPTED, FLAG_SIGNING_ON.
SentPackets	uint32	Number of packets, sent to ASTS Bridge server.
RecvPackets	uint32	Number of packets, received from ASTS Bridge server.
SentBytes	uint32	Number of bytes, sent to ASTS Bridge server, considering compression.
RecvBytes	uint32	Number of bytes, received from ASTS Bridge server, considering compression.
ServerIpAddress	uint32	ASTS Bridge server IP-address.
ReconnectCount	uint32	Number of reconnections to ASTS Bridge server.
SentUncompressed	uint32	Number of bytes, sent to ASTS Bridge server, not taking compression into account.
RecvUncompressed	uint32	Number of bytes, received from ASTS Bridge server, not taking compression into account.
ServerName	char[64]	ASTS Bridge server identifier.
TsmrPacketSize	uint32	Size of packet of TSMR protocol, bytes (only for colocation version).
TsmrSent	uint32	Number of bytes, sent to TS via TSMR protocol (only for colocation version).
TsmrRecv	uint32	Number of bytes, received from TS via TSMR protocol (only for colocation version).

If error occurs, one of the MTE_xxxx error codes is returned.

GETTING DESCRIPTION OF INFORMATION OBJECTS

Information objects description contains a list of tables, transactions, their fields and some additional objects, available to the client. `MTEStructure`, `MTEStructure2` and `MTEStructureEx` functions are used to get the description. `MTEStructure2` and `MTEStructureEx` functions return an expanded set of trading system objects characteristics (see Appendix 1).

`MTEStructureEx` completely covers all the capabilities of two other functions: `MTEStructure` call is similar to `MTEStructureEx` call with *Version=0* attribute, `MTEStructure2` call is similar to `MTEStructureEx` call with *Version=2* attribute.

C++

```
int32 WINAPI MTEStructure(int32 Idx, MTEMsg **Msg);
int32 WINAPI MTEStructure2(int32 Idx, MTEMsg **Msg);
int32 WINAPI MTEStructureEx(int32 Idx, int32 Version, MTEMsg **Msg);
```

Pascal

```
function MTEStructure(Idx: Integer; var Msg: PMTEMsg): Integer; stdcall
function MTEStructure2(Idx: Integer; var Msg: PMTEMsg):Integer;stdcall;
function MTEStructureEx(Idx: Integer; Version: Integer; var Msg:
      PMTEMsg):Integer; stdcall;
```

Arguments:

Idx

A descriptor of connection, for which, the data should be received.

Version

[Only for `MTEStructureEx`]. The version of the information objects description. Possible values are in range from 0 to 5. The higher the value is, the more detailed description will be received.

Starting from version 3 this argument allows to get additional information about the trading system interface. Below are the supported value options. They can be combined with each other and with the version number using the binary OR operator.

Version	Option	Description

>= 3	STRUCTURE_LOCALIZATION = 0x0100	The 'title' and the 'description' fields will be provided on all the supported languages. Instead of the String field type the following structure will be used:
		`NumberOfLanguages Integer`
		`String1 String`
		`String2 String`
		`...`
		`StringN String`
		Each string will start with one of the language prefixes, such as 'ru:', 'en:' or 'uk:'. For example, 'ru:Номер заявки', 'en:Order number'.

Msg

An address of a variable (of the type "a pointer to a TMTEMsg/MTEMSG") to store a pointer to the buffer, containing information objects description. Memory for this buffer is allocated by the library. Buffer format for `MTEStructure` and `MTEStructure2` functions is described in Appendix 1. TMTEMsg structure is defined as follows:

C++

```
typedef struct {
      int32_t  DataLen; // The length of the data to follow
      char     Data[1]; // Pseudo variable
} MTEMSG;

// data of the DataLen length directly follows this structure.
```

Pascal

```
PMTEMsg = ^TMTEMsg;
TMTEMsg = record
  DataLen: Integer;  // The length of the data to follow
  Data: record end;  // Variable length data
end;
```

Returned value:

In case of success, function returns MTE_OK and places a buffer with the description to Msg argument.

If error occurs, one of the MTE_xxxx error codes is returned. If MTE_TSMR error code is returned, then the `data` field of Msg structure contains the error message of [DataLen] length.

Example:

Get the description of available information objects for the Idx session.

C++

```
int32 Idx;               // Initiated by the MTEConnect
char ErrorMsg[255];
MTEMsg *Msg;
char *Data;
```

```
int32 err;
...
if ((err = MTEStructure(Idx, &Msg)) != MTE_OK ) {
      if (Err == MTE_TSMR) {
            Data = (char *)(Msg + 1);
            fprintf(stderr, "Error: %s\n", Data );
      } else
            fprintf(stderr, "Error: %s\n", MTEErrorMsg(Err));
} else
      fprintf("Information objects description has been
received.\n");
Data = (char *)(Msg + 1); // Actual data
```

Pascal

```
Idx: Integer;           // Initiated by the MTEConnect
Err: Integer;
Msg: PMTEMsg;
S: string;
Data: PAnsiChar;
...
Err := MTEStructure(Idx, Msg);
if Err <> MTE_OK then
  if Err = MTE_TSMR then begin
    SetString(S, @Msg.Data, Msg.DataLen);
    Writeln('Error: ' + S);
  end else
    Writeln('Error: ' + MTEErrorMsg(Err))
else
  Writeln(Information objects description has been received.);
Data := @Msg.Data;      // Actual data
```

WORKING WITH INFORMATION OBJECTS

Working with information objects includes working with tables and transactions execution.

TRANSACTIONS EXECUTION

All the transactions, such as order entry, withdrawal, etc. are executed with `MTEExecTrans`, `MTEExecTransIP` and `MTEExecTransEx` functions.

C++

```
int32 WINAPI MTEExecTrans(int32 Idx, char *TransName, char *Params,
                          char *ResultMsg);
int32 WINAPI MTEExecTransIP(int32 Idx, char *TransName, char *Params,
                          char *ResultMsg, int32 ClientIP);
```

Pascal

```
function MTEExecTrans(Idx: Integer; TransName, Params,
            ResultMsg: LPSTR): Integer; stdcall;
function MTEExecTransIP(Idx: Integer; TransName, Params,
            ResultMsg: LPSTR; ClientIP: Integer): Integer; stdcall;
```

Arguments:

Idx

A descriptor of connection, on which, the transaction is being executed.

TransName

A pointer to an ASCIIZ string containing the name of transaction. Available names can be obtained with `MTEStructure`, `MTEStructure2` or `MTEStructureEx` functions.

Params

A pointer to an ASCIIZ string containing the transaction parameters. The length of the string and its value must match the description of transaction input fields (obtained with `MTEStructure`, `MTEStructure2` or `MTEStructureEx` functions). All fields have to be submitted as text, according to the following trading system formatting:

ftChar	Blank spaces are appended to correspond to the string length, defined in the field description. For example, for a ftChar(12) field the string "`USER`" has to be presented as "`USER `".
ftInteger	Zeros are added to the left side to reach the required length. For example, the value 127 of the ftInteger(10) type has to be presented as “0000000127”.
ftFixed	Two symbols after the decimal point are kept, the decimal point itself is deleted, and zeros are added to the left side to reach the required length. For example, value 927.4 of the ftFixed(8) type has to be transformed into “00092740” string.
ftFloat	N symbols after the decimal point are kept, the decimal point itself is deleted, zeros are added to the left side to reach the required length. The value of N depends on the price precision of a given financial instrument. For example, value 26.75 of the ftFloat(9) type for the instrument with N=4, has to be presented as “000267500”.
ftDate	Specified as YYYYMMDD. For example, 24 August 1999 has to be presented as "19990824".
ftTime	Specified as HHMMSS. For example, 16:27:39 is to be presented as "162739".
ftFloatPoint	Zeros are added to the left side to reach the required length, the decimal point is kept. For example, value 5.617 of the ftFloatPoint(9) type has to be transformed into "`00005.617`" string
Note:	An empty value (NULL) can be specified in a field of any type; for this, a string of all blanks of the required length is used.

ClientIp

(For `MTEExecTransIP` function) IP-address of the client, on whose behalf, the transaction is performed. To be used in interfaces for technical centers and regional exchanges.

ResultMsg

A pointer to a buffer of at least 256 bytes to store a text string containing the result of transaction execution, in case of success.

Returned value:

If transaction has been processed by the trading system, then, one of the following codes is returned:

MTE_OK – transaction executed;
MTE_TRANSREJECTED – the transaction has been received, but rejected by the trading system (incorrect arguments, no rights to execute transactions, etc.);
MTE_TSMR - fatal error during the transaction execution (connection to the trading system is lost, etc.).

A text string with the result of the transaction processing is stored in *ResultMsg* argument.

If error occurs, one of the MTE_xxxx error codes is returned. In this case a value of *ResultMsg* is not defined.

Example:

Let the description of an object (received with `MTEStructure`) contains “Enter an order” transaction with the following fields:

```
ORDER                       // Transaction name
  BuySell: ftChar(1)        // "B" - buy, "S" - sell
  SecBoard: ftChar(4)       // Board code
  SecCode: ftChar(12)       // Security code
  Price: ftFloat(9)         // Price
```

```
Quantity: ftInteger(10)  // Number of lots
```

The following code is used to submit an order to buy 14 items of the "USD000000TOD" on "CETS" board at the price of 26.15 (for this security, the price precision is 4 symbols after the decimal point):

C++

```
int32 Idx; // Initiated by MTEConnect
int32 Err;
char *ResultMsg;
...
Err = MTEExecTrans(Idx, "ORDER",
"BCETSUSD000000TOD00026150000000000014", ResultMsg);
if( Err == MTE_OK )
  fprintf(stdout,"Transaction executed: %s\n", ResultMsg);
else if( Err == MTE_TSMR )
  fprintf(stdout,"Transaction IS NOT executed: %s\n", ResultMsg);
else fprintf(stderr,"Error: %s\n", MTEErrorMsg(Err));
```

Pascal

```
Idx: Integer;           // Initiated by MTEConnect
Err: Integer;
ResultMsg: TErrorMsg;
...
Err := TEExecTrans(Idx, 'ORDER',
'BCETSUSD000000TOD00026150000000000014', @ResultMsg);
case Err of
  MTE_OK: Writeln('Transaction executed: ' + ResultMsg);
  MTE_TSMR, MTE_TRANSREJECTED: Writeln('Transaction IS NOT
executed: ' + ResultMsg);
  else Writeln('Error: ' + MTEErrorMsg(Err));
end;
```

Note 1: all transactions or table data requests are sent sequentially within one connection. It means that a transaction or a table data request can be sent to the trading system only after the reply to the previous one is received. To avoid any related delays it is recommended:

- To use separate connections to perform transactions and to request table data.
- To use load balancer to distribute transactions between connections in case of high transaction volume.

Note 2: when connected to independent trading and clearing systems, the "change password" transaction should be executed as follows: first, send the CHANGE_PASSWORD transaction to the Trading System, and after its successfully executed, send the same CHANGE_PASSWORD transaction to the Clearing System. This is necessary for the automatic reconnect to the Clearing System to perform smoothly.

New transactions supported by the trading system can return multiple replies or string, longer than 255 symbols. For that kind of transactions, it's recommended to use `MTEExecTransEx` function, which returns an array of replies and text messages of unlimited length:

C++

```
int32 WINAPI MTEExecTransEx(int32 Idx, char *TransName, char *Params,
                  int32 ClientIp, MTEExecTransResult *Reply);
```

Pascal

```
function MTEExecTransEx(Idx: Integer; TransName, Params: LPSTR;
  ClientIp: Integer; var Reply: TMTEExecTransResult): Integer; stdcall;
```

Arguments:

Idx

A descriptor of connection, on which, the transaction is being executed.

TransName

A pointer to an ASCIIZ string containing the name of transaction. Possible names can be obtained by calling `MTEStructure`, `MTEStructure2` or `MTEStructureEx` functions.

Params

A pointer to an ASCIIZ string containing the transaction parameters. The length of the string and its value must match the description of transaction input fields, obtained by calling `MTEStructure/MTEStructure2` or `MTEStructureEx` functions. All fields have to be submitted as text, with the proper formatting (see. `MTEExecTrans`).

ClientIp

IP-address of the client, on whose behalf, the transaction is performed. To be used in interfaces for technical centers and regional exchanges.

Reply

A pointer to a text string, in which, the transaction execution result and trading system reply are stored. The `TMTEExecTransResult / MTEExecTransResult` structure is defined as:

C++

```
typedef struct TransResult {
      // a number of entries in "replies" field
      uint32_t replyCount;
      // a pointer to an array of MTETransReply entries
      MteTransReply* replies;
} MteTransResult;

typedef struct TransReply {
      int32_t errCode;  // Returned code (see. Returned values)
      int32_t msgCode;  // A number of message in Trading System
(which is indicated by brackets in the text)
      char*   msgText;  // Trading System text message
      int32_t paramCount; // A number of parameters in the reply
      MteTransParam* params;  // An array of parameters in the
reply
} MteTransReply;
```

Pascal

```
TMTEExecTransResult = record
  // a number of entries in "Replies" field
  ReplyCount: Longword;
  // a pointer to an array of TMTETransReply entries
  Replies: PMTETransReplies;
end;

// single reply of the Trading System
TMTETransReply = record
  ErrCode: TMTEResult;  // Returned code (see. Returned values)
  MsgCode: Integer;     // A number of message in Trading System
(which is indicated by brackets in the text)
  MsgText: PAnsiChar;   // Trading System text message
  ParamCount: Integer;  // A number of parameters in the reply
  Params: PMTETransParams;    // An array of parameters in the
reply
end;
```

Most of transactions return only one single reply, so `ReplyCount` value is "1" and `Replies` contains 1 entry. An example of transaction, which returns more than one reply is ORDER_AMEND.

Returned value:

If the transaction has been processed by trading system, the following is returned:

- MTE_OK – transaction successfully executed;
- MTE_TRANSREJECTED – the transaction has been processed, but rejected by the trading server (invalid board specified, no rights to perform, etc.);
- MTE_TSMR - fatal error occurred, when processing the transaction (the loss of connection with the trading system, etc.).

Additional parameters that may be present in a reply. Number of parameters is specified in ParamCount.

- ST – time when transaction processing started by the trading engine in the following format: ST=HHMMSSmicroseconds
- ON – order number
- IN – public order number in the FAST UDP Market Data feed; only available for orders that are published in this feed.

WORKING WITH TABLES

Working with tables includes the following steps:

1. Opening a table
2. Periodically requesting for updates
3. Closing the table

OPENING A TABLE

To start working with a table, first it's necessary to call `MTEOpenTable` function. This function opens a table and returns the content of the table partially or at once..

C++

```
int32 WINAPI MTEOpenTable(int32 Idx, char *TableName, char *Params,
                          int32 Complete, MTEMSG **Msg);
```

Pascal

```
function MTEOpenTable(Idx: Integer; TableName, Params: LPSTR;
                  Complete: BOOL; var Msg: PMTEMsg): Integer; stdcall;
```

Arguments:

Idx

A descriptor of connection, obtained by `MTEConnect`.

TableName

A pointer to an ASCIIZ string containing the name of the table. Available names can be obtained with `MTEStructure, MTEStructure2` or `MTEStructureEx` functions.

Params

A pointer to an ASCIIZ string containing the parameters of the table. The length of the string and its value must match the description of table input fields, received with `MTEStructure, MTEStructure2` or `MTEStructureEx`. All fields have to be submitted as a text with the proper formatting (see `MTEExecTrans`).

Complete

Flag to request either all the table data at once or only a part of it:

TRUE Return all the table data. Function will query the trading system as many times as needed to obtain all the data. In case of big table size (e.g. TRADES or SETTLECODES) it may take a long time and even lead to disconnection on timeout. If the content is not needed all at once, then in order to decrease execution time, the FALSE value should be used.

FALSE Depending on the table type, the function returns only a part of the data or nothing at all. Function will query the trading system one time, maximum. The remaining data will considered as an update and should be read during the update request cycle, initiated with `MTEAddTable/MTERefresh`.

Msg

An address of a variable (of the type "a pointer to a TMTEMsg/MTEMSG"), to store a pointer to the buffer, containing the data of opened table. Buffer format is described in Appendix 2.

Returned value:

If successful, a descriptor of an open table is returned (value that is greater or equal to MTE_OK). Received descriptor can be used when calling `MTEAddTable` function.

If error occurs, one of the MTE_xxxx error codes is returned. If MTE_TSMR error code is returned, then "Data" field of the Msg structure contain error message with a length of [DataLen] symbols.

REQUEST FOR UPDATE

Request for a table content update is performed in a batch mode, i.e. requests to update several open tables are processed simultaneously. A set of tables to be refreshed is formed by calling `MTEAddTable` function for every table. Then all the updates can be received with `MTERefresh` function. Execution of other library functions (except `MTEErrorMsg`) is not allowed between those two functions.

`MTEAddTable` function adds a table to the update queue (changes that occurred since the last request).

C++

```
int32 WINAPI MTEAddTable(int32 Idx, int32 HTable, int32 Ref);
```

Pascal

```
function MTEAddTable(Idx, HTable, Ref: Integer): Integer; stdcall;
```

Arguments:

Idx

Connection descriptor received with `MTEConnect`.

HTable

Table descriptor received with `MTEOpenTable`.

Ref

Optional parameter to store arbitrary data. Usually used to match the data with a table in a buffer, received with `MTERefresh`.

Returned value:

One of the `MTE_xxxx` error codes.

`MTERefresh` function performs the batch table updates (the request is formed with the `MTEAddTable`)

C++

```
int32 WINAPI MTERefresh(int32 Idx, MTEMSG **Msg);
```

Pascal

```
function MTERefresh(Idx: Integer; var Msg: PMTEMsg): Integer; stdcall;
```

Arguments:

Idx

Connection descriptor obtained by calling `MTEConnect`.

Msg

An address of a variable (of the type "a pointer to a TMTEMsg/MTEMSG") to store the received updates. The buffer format is described in appendix 3.

Returned value:

If successful then MTE_OK is returned and pointer to the update is saved into Msg argument

If error occurs, one of the MTE_xxxx error codes is returned. If MTE_TSMR error code is returned, then the Data field of the Msg structure will contain the error message and have the DataLen length of string.

CLOSING THE TABLE

Upon the end of work with a table it should be closed with `MTECloseTable`. The table descriptor cannot be used after this function execution.

C++

```
int32 WINAPI MTECloseTable(int32 Idx, int32 HTable);
```

Pascal

```
function MTECloseTable(Idx, HTable: Integer): Integer; stdcall;
```

Arguments:

Idx

Connection descriptor received with `MTEConnect`.

HTable

A descriptor of the closing table, received with `MTEOpenTable`.

Returned value:

One of `MTE_xxxx` error codes.

EXAMPLE

Let the structure of input fields (received with `MTEStructure`) of SECURITIES and TRADES tables, is as follows:

```
SECURITIES                // Table name (Securities)
  Market: ftChar(4)       // Market code
  Board: ftChar(4)        // Trading board (mode) code

TRADES                    // "TRADES" table has no input fields
```

The following code shows how to work with tables. Tables are opened, their content is periodically updated and then the tables are closed.

C++

```
int32 Idx; // Initiated by MTEConnect
MTEMsg *Msg;
char *Data;
int32 HSecurs, Htrades;
...

HSecurs = MTEOpenTable(Idx, "SECURITIES", "CETS    ", 1 /*True*/,
&Msg);
Data = (char *)(Msg + 1);
...
// Processing the received data
...
HTrades = MTEOpenTable(Idx, "TRADES", "", 0/*False*/, Msg);
Data = (char *)(Msg + 1);
...
// Processing the received data
...
```

```
do
{
  MTEAddTable(Idx, HSecurs, 0);
  MTEAddTable(Idx, HTrades, 1);
  MTERefresh(Idx, &Msg);
  Data = (char *)(Msg + 1);
  ...
  // Processing the updates
  ...
}while( !Terminated );

MTECloseTable(Idx, HSecurs);
MTECloseTable(Idx, HTrades);
```

Pascal

```
Idx: Integer;           // Initiated by MTEConnect
Msg: PMTEMsg;
HSecurs, HTrades: Integer;
Data: PAnsiChar;
...

HSecurs := MTEOpenTable(Idx, 'SECURITIES', 'CETS    ', True,
Msg);
...
// Processing the received data
...
HTrades := MTEOpenTable(Idx, 'TRADES', '', False, Msg);
...
// Processing the received data
...

repeat
  MTEAddTable(Idx, HSecurs, 0);
  MTEAddTable(Idx, HTrades, 1);
  MTERefresh(Idx, Msg);
  Data := @Msg.Data;
  ...
  // Processing the updates
  ...
until Terminated;

MTECloseTable(Idx, HSecurs);
MTECloseTable(Idx, HTrades);
```

NOTES ON WORKING WITH TABLES

Note 1. Follow these steps to avoid disconnections on timeout: 1. do not to set too small (less than 60 seconds) values for the DisconnectIfIdleFor parameter in ASTS Bridge configuration file; 2. maintain active connection (heartbeat) by regular (approximately every 30 seconds) requests – for example, to update TESYSTIME table. The USER_HEARTBEAT transaction can be used to monitor the connection status.

Note 2. Most of the tables can be opened and closed anytime and as many times as needed during the connection session with a server. Any number of table copies can be opened. However, some of the tables can be opened only once during the session. These tables are: ORDERS, TRADES, NEGDEALS, ALL_TRADES, POSITIONS, HOLDINGS, RM_INDICATIVE. If such table is closed and then opened again, then initial content of the table will not be received again – only content updates will come.

Consequently, it is recommended to open such tables only once during the connection session and close them only at the end of the session.

Note 3. For tables having the "tfClearOnUpdate – Clear on update" flag (except for the EXT_ORDERBOOK table) the following updates processing order is defined: when a table is to be cleared, then the RowsNumber is set to 1, i.e. only a single string with DataLength=0 is returned (see Appendix 2).

There are two types of requests for orderbook (quotations) for the EXT_ORDERBOOK table:

1. To get information on one security, the request has to have non-empty values of “Board” and “Security” fields;
2. To get orderbook (quotations) for all available securities with one request, fields “Board” and “Security” have to be filled-in with spaces.

For the first type, when the orderbook table has to be cleared as the result of request, a table with a single row is received that contains the following values: NumberOfFields=2 and DataLength=(length of “Board” field + length of “Security” field). This string contains only the “Board” and “Security” fields. For the second type, the reply on request can contain several such strings (which contain only the values of “Board” and “Security” fields) – for given financial instruments this will mean the deletion of orderbook (quotation) values.

Note that during the first request for all securities (i.e. at opening time), strings with initial zero values of orderbook can be received. This is explained by the Trading system data transfer mechanism: the status of these instruments has changed, so the Trading system only sends updates of the orderbook fields, which are not reflected in clients’ systems. That is why all the updated orderbooks are transmitted even if they are empty. The consequent requests will return data only on the orderbooks that have changed.

Also note that the test TEClient.exe application only shows instruments with changes since the last request when opening the orderbook for the whole market, i.e. only those of the instruments that have updates in the orderbook. Information on instruments with no orderbook changes will not be shown.

Note 4. The maximum refresh interval is governed by a document "Requirements for external systems and their interfacing with ASTS Trading system". To avoid any delays at peak times, it’s possible to use the adaptive refresh model: if the received data buffer is greater than 30 Kbytes, then ask for another update immediately. If the buffer is less than 30 Kbytes then send the next request with standard interval (in 1 second, for example).

Note 5. When processing the data buffer with table rows all the records with matching values in key fields should be merged into one table row. In certain cases, for example when opening the SECURITIES table, even a single buffer may include several records for one row. Besides that, as will be explained in the appendices below, one record in a buffer may either represent a whole table row (including static values) or changes only. It is recommended to always implement the scenario when a partial set of fields may be received for any table.

MEMORY USE OPTIMIZATION

All the functions of MTESRL library that return pointers to data buffer (pointer to the PMTEMsg/MTEMSG structure; for example, `MTEStructure`, `MTERefresh`) use the same memory region as the reception buffer (this is for one connection; with multiple connections multiple memory areas are used). Let’s call these functions “informational functions”.

If informational function call returns data buffer that is larger than allocated, then the reallocation of a larger block of memory will occur. Thus the maximum size of allocated memory equals to the largest block of data received. All the allocated memory is released when connection is closed with `MTEDisconnect`.

It is also possible to free the memory allocated for the buffer at any time, without closing the connection. `MTEFreeBuffer` function is used for this purpose. This function should be called only after all the received data has been processed. It should be kept in mind that before the next call of any of the informational functions, memory should be allocated again. Frequent use of `MTEFreeBuffer` can negatively influence the performance.

C++

```
int32 WINAPI MTEFreeBuffer(int32_t conno);
```

Pascal

```
function MTEFreeBuffer(Idx: Integer): Integer; stdcall;
```

Arguments:

Idx

A descriptor of connection received with `MTEConnect` to free the memory for.

Returned value:

One of the `MTE_xxxx` error codes.

This is the legacy function that is kept for compatibility with old client software.

RECOVERY AFTER FAILURES AT ASTS Bridge SERVER

During operation, external system or ASTS Bridge sometimes needs to be restarted in case of a critical error. In that case, it is necessary to restore the system as soon as possible. In such situations, it is recommended to use the following technology: external system makes a backup of loaded tables and state of internal structures in files with a certain periodicity; in case of failure, data from the saved files is used to restore last saved state of the external system.

MTESRL library allows to initiate the data transfer from ASTS Bridge Server, not only from the beginning of a trading session, but from a certain point as well. To do so, the snapshot of opened tables status should be made beforehand. Afterwards (if, for example, the connection to ASTS Bridge Server has been lost) it will be possible to recover the status of open tables and continue getting data.

BACKING UP BRIDGE INTERNAL STRUCTURE

Backing up the state of the Bridge internal structures is performed after requesting and processing tables' updates. This operation can be performed after each request for changes or after certain number of them. As a rule, along with saving of bridge internal structures, the current state of all tables of the external system is backed up. This ensures complete preservation of the current state of the whole system, consisting of an external system and ASTS Bridge. A detailed scenario of operation in this case is shown below:

To obtain a current state of tables opened on the server, use `MTEGetSnapshot` function.

C++

```
int32 WINAPI MTEGetSnapshot(int32 Idx, char ** Snapshot, int *Len);
```

Pascal

```
function MTEGetSnapshot(Idx: Integer; var Snapshot: LPSTR;
                        var Len: Integer): Integer; stdcall;
```

Arguments:

Idx

Descriptor of connection, for which, the snapshot of opened tables should be received.

Snapshot

Address of the variable where pointer to the snapshot will be placed in case of success.

Len

Address of the variable, where the snapshot (i.e. buffer at which the *Snapshot* points) length will be placed in case of success.

Returned value:

In case of success the function returns MTE_OK.

If error occurs, one of the error MTE_xxxx codes is returned. If MTE_TSMR error code is returned, then the *Snapshot* will point to the error message and the *Len* will contain the length of this message.

The snapshot of tables, loaded on the server side, can be considered just as a buffer with some binary data. Its content does not have any meaning for the client.

The following code assumes that external system has connected to ASTS Bridge, received a data structure, opened tables and moved to the cycle of getting tables updates:

C++

```
int32 Idx;        // Initiated by MTEConnect
MTEMsg *Msg;
char *DataPtr;
int32 *TablesIdx; // array of indexes received with MTEOpenTable
int32 i,NumTables;// number of the updated tables
char *SnapshotBuf;// pointer to the buffer for the emergency
int32 SnapshotLen;// length of the buffer for the emergency saving
...
do
{
  for( i = 0; i < NumTables; i++ )
      MTEAddTable(Idx, TablesIdx[i], i);
  MTERefresh(Idx, &Msg);
  DataPtr = (char *)(Msg + 1);
  ...
  // Processing the updates
  ...
  // Receive of the buffer for the Bridge internal structure
  MTEGetSnapshot(Idx, &SnapshotBuf, &SnapshotLen);
  // saving the buffer to the file
  ...
  // saving the status
  ...
}while( !Terminated );
```

Pascal

```
Idx: Integer;           // Initiated by MTEConnect
Msg: PMTEMsg;
DataPtr: PChar;
TablesIdx: array of Integer; // of indexes received with
MTEOpenTable
i, NumTables: Integer;  // number of the updated tables
SnapshotBuf: PChar;     // pointer to the buffer for the
emergency
SnapshotLen: Integer;   // length of the buffer for the emergency
saving
...
repeat
  for i := 0 to NumTables - 1 do
      MTEAddTable(Idx, TablesIdx[i], i);
  MTERefresh(Idx, Msg);
  DataPtr = @Msg.Data;
  ...
  // Processing the updates
  ...
  // Receive of the buffer for the Bridge internal structure
  MTEGetSnapshot(Idx, SnapshotBuf, SnapshotLen);
  // saving the buffer to the file
  ...
  // saving the status
```

```
  ...
until Terminated;
```

BRIDGE INTERNAL STRUCTURE RECOVERY

To get the list of opened tables, contained in a given snapshot, use `MTEGetTablesFromSnapshot` function. This function can be called both before and after `MTESetSnapshot`.

C++

```
int32 WINAPI MTEGetTablesFromSnapshot(int32 Idx, char * Snapshot,
      int Len, MTESnapTable **SnapTables);
```

Pascal

```
function MTEGetTablesFromSnapshot(Idx: Integer; Snapshot: LPSTR;
      Len: Integer, var SnapTables: PMTESnapTables): Integer; stdcall;
```

Arguments:

Idx

Connection descriptor, obtained by `MTEConnect` function.

Snapshot

A pointer to a buffer, where the snapshot, taken by `MTEGetSnapshot`, is stored.

Len

Buffer length.

SnapTables

An address of a variable containing a pointer to `MTESnapTable` structure, where, in case of success, a pointer to a buffer of opened tables will be placed. A memory for this buffer is allocated by a library. In case of repeated calls to this function, the same buffer is used, so, result should be saved by external system. The buffer has following format:

C++

```
typedef struct SnapTable {
      int32 Htable;     // Descriptor of the opened table
      char* TableName   // A poiner to an ASCIIZ-string,
containing table name.
      char* Params;     // A pointer to an ASCIIZ-string,
containing the parameters, used when opening the table.
} MteSnapTable;
```

Pascal

```
TMTESnapTable = record
  HTable: Integer;          // A table descriptor
  TableName: PAnsiChar;     // char, Zero-byte terminated, Table
Name
  Params: PAnsiChar;        // char, Zero-byte terminated,
Parameters provided on open table
end;

PMTESnapTables = ^TMTESnapTables;
TMTESnapTables = array [0..999999] of TMTESnapTable;
```

Returned value:

In case of negative value, return code is interpreted as MTE_xxxx error code.

In case of success, function returns non-negative value, equal to the number of opened tables, and a pointer to a formed array of tables structures `MTESnapTable` through `SnapTables` parameter.

Internal structures recovery is performed when restarting Bridge or external system after failures to restore the system to the moment of last snapshot. This operation should be performed only within the current

trading session (see. MTEGetSnapshot). As a result, all opened tables and their descriptors will be restored. So, previously used descriptors can be used again right after recovery. MTESetSnapshot function can be used to restore Bridge last saved state.

C++

```
int32 WINAPI MTESetSnapshot(int32 Idx, char * Snapshot, int Len,
                            char *ErrorMsg);
```

Pascal

```
function MTESetSnapshot(Idx: Integer; Snapshot: LPSTR; Len: Integer;
                        ErrorMsg: LPSTR): Integer; stdcall;
```

Arguments:

Idx

A descriptor of connection, for which, the last state is restored.

Snapshot

A pointer to the buffer, which stores previously taken “snapshot”.

Len

The length of the buffer, pointed by a snapshot.

ErrorMsg

A pointer to at least 256 bytes buffer, to store a text string containing the result of restoring.

Returned value:

If function was successfully processed by the trading system, the following will be returned:

MTE_OK – restoring complete;
MTE_TSMR - trading system is unable to restore the state.

A text string containing result, returned by trading system, will be placed to *ErrorMsg* argument.

If error occurs, one of the MTE_xxxx error codes is returned. ErrorMsg field value is not defined.

The following code assumes that external system has backed up own and Bridge’s state before the failure. Complete restart of the system, including Bridge server, is performed (acts similar when restarting only external system or just ASTS Bridge server). System has connected to Bridge server and obtained data structure description:

C++

```
int32 Idx;              // Initialized MTEConnect call
MTEMsg *Msg;
char *DataPtr;
int32 *TablesIdx;       // array of indexes of opened tables
int32 i, NumTables;     // a number of updated tables
char *SnapshotBuf;      // a pointer to a data buffer that will
be used when restoring the state of Bridge server
int32 SnapshotLen;      // buffer length
...
// Recovery of the external system from the stored data
// At the same time NumTables values and index array of open tables
...
// Loading of the saved buffer from the file,
// which was backed up after last MTEGetSnapshot call,
// (initialization and loading SnapshotBuf buffer)
...
//Restoring the internal structures last state
MTESetSnapshot(Idx, SnapshotBuf, SnapshotLen);
//start of the normal operation cycle of the external system
do
{
  for( i = 0; i < NumTables; i++ )
      MTEAddTable(Idx, TablesIdx[i], i);
  MTERefresh(Idx, &Msg);
```

```
  DataPtr = (char *)(Msg + 1);
  ...
  // Processing the updates
  ...
}while( !Terminated );
```

Pascal

```
Idx: Integer;           // Initialized MTEConnect call
Msg: PMTEMsg;
DataPtr: PChar;
TablesIdx: array of Integer; // array of indexes of opened tables
i, NumTables: Integer;  // a number of updated tables
SnapshotBuf: PChar;     // a pointer to a data buffer that will
be used when restoring the state of Bridge server
SnapshotLen: Int32;     // buffer length
...
// Recovery of the external system from the stored data
// At the same time NumTables values and index array of open tables
...
// Loading of the saved buffer from the file,
// which was backed up after last MTEGetSnapshot call,
// (initialization and loading SnapshotBuf buffer)
...
// Restoring the internal structures last state
MTESetSnapshot(Idx, SnapshotBuf, SnapshotLen);
// start of the normal operation cycle of the external system
repeat
  for i := 0 to NumTables - 1 do
      MTEAddTable(Idx, TablesIdx[i], i);
  MTERefresh(Idx, Msg);
  DataPtr = @Msg.Data;
  ...
  // Processing the updates
  ...
until Terminated;
```

EXAMPLE OF RECOVERY AFTER THE FAILURE

Suppose that we have:

1. Established connection with ASTS Bridge Server with `MTEConnect`.
2. Opened several tables by calling `MTEOpenTable` and saved their descriptors in variables named hTable1, hTable2, ..., hTableN.
3. Executed some transactions, requested updates of informational tables, periodically saved the snapshots with `MTEGetSnapshot`.
4. Now suppose that at certain point the connection with ASTS Bridge Server has been lost. The recovery procedure will be as follows.
5. Reconnect to the Bridge Server with `MTEConnect`;
6. Call `MTESetSnapshot` with the last saved snapshot
7. Now we can use previously defined table handles hTable1, hTable2, ..., hTableN. There is no need to call `MTEOpenTable` again. All the following `MTERefresh` calls will return tables updates, accumulated after saving Snapshot.

If the data, received before the connection loss, have been saved, Get / Set Snapshot mechanism can significantly reduce the time of reception of all tables' updates after the reconnection.

SELECTIVE OPEN OF TABLES FROM THE SNAPSHOT

There is also an alternative way to restore the system after failure. Instead of saving and loading complete state of all tables, it's possible to restore only certain large tables (e.g. "ORDERS", "TRADES"), and open other tables in the usual way – with `MTEOpenTable` function. This way eliminates the need for storage a list of open tables along with their descriptors. It's enough to retain only the snapshot, and then open the tables, using the `MTEOpenTableAtSnapshot` function. The data from tables, opened this way, will not come from scratch but from the moment when an appropriate snapshot was taken. There is no need to call `MTESetSnapshot` in that scenario.

C++

```
int32 WINAPI MTEOpenTableAtSnapshot (int32 Idx, char* TableName,
      char* Params, char* Snapshot, int SnapshotLen, MTEMsg **Msg);
```

Pascal

```
function MTEOpenTableAtSnapshot(Idx: Integer;
      TableName, Params, Snapshot: PAnsiChar;
      SnapshotLen: Integer; var Msg: PMTEMsg): Integer; stdcall;
```

Arguments:

Idx

A descriptor of connection, obtained by calling `MTEConnect`.

TableName

A pointer to ASCIIZ string containing a table name. Possible names can be obtained by calling `MTEStructure`, `MTEStructure2` or `MTEStructureEx` functions.

Params

A pointer to ASCIIZ string containing parameters of the table. The length of the string and its value must match the description of table input fields, received with `MTEStructure` or `MTEStructure2` or `MTEStructureEx`. All fields have to be submitted as a text with trading system formatting.

Snapshot

A pointer to a buffer containing a snapshot. The requested table with the specified parameters should be included in this snapshot, otherwise the function returns an `MTE_TSMR` error. If null pointer is passed in this parameter, the function behaves like a call to `MTEOpenTable` with `Complete = FALSE` option.

SnapshotLen

A length of the buffer containing the snapshot.

Msg

Address of variable (of type "pointer to `TMTEMsg/MTEMSG`"), which, if successful, will store a pointer to a buffer containing a portion of updates for an open table. The buffer format is described in Appendix 2.

Returned value:

In case of success, function returns descriptor of the opened table (value greater or equal MTE_OK). Obtained descriptor is used when calling `MTEAddTable` function.

If error occurs, one of the `MTE_xxxx` error codes is returned. If the returned error code is `MTE_TSMR`, the `Data` field of Msg structure contains error message of `DataLen` characters length.

The following code shows selective opening of «Orders» table from the snapshot:

C++

```
int32 Idx;              // Initialized by calling MTEConnect
MTEMsg *Msg;
char *DataPtr;
```

```
char *Snapshot;
int32 Len;
int32 HSecurs, HTrades;
...
HSecurs = MTEOpenTable(Idx, “SECURITIES”, “EQBR    “, 1 /*True*/,
      &Msg);
// Processing the received data
...
HTrades = MTEOpenTable(Idx, “TRADES”, “”, 0 /*False*/, &Msg);
// Processing the received data
...
// Fail occurred!. Saving the snapshot and closing the tables
MTEGetSnapshot(Idx, &Snapshot, &Len);
MTECloseTable(Idx, HSecurs);
MTECloseTable(Idx, HTrades);
...
// Recovery starts. Loading the snapshot and opening the tables
HSecurs = MTEOpenTable(Idx, “SECURITIES”, “EQBR    “,
      1 /*True*/, &Msg);
// SECURITIES table is opened from scratch, processing the data
...
HTrades = MTEOpenTableAtSnapshot(Idx, “TRADES”, “”, Snapshot,
      Len, &Msg);
// TRADES table is opened from the snapshot, processing the data
...
do {
  MTEAddTable(Idx, HSecurs, 0);
  MTEAddTable(Idx, HTrades, 1);
  MTERefresh(Idx, &Msg);
  DataPtr = (char *)(Msg + 1);
  // Processing the updates
  ...
} while (!Terminated);

MTECloseTable(Idx, HSecurs);
MTECloseTable(Idx, HTrades);
```

Pascal

```
Idx: Integer;           // Initialized by calling MTEConnect
Msg: PMTEMsg;
HSecurs, HTrades: Integer;
Snapshot: PAnsiChar;
Len: Integer;
Data: PAnsiChar;
...
HSecurs := MTEOpenTable(Idx, 'SECURITIES', 'EQBR    ', True,Msg);
// Processing the received data
...
HTrades := MTEOpenTable(Idx, 'TRADES', '', False, Msg);
// Processing the received data
...
// Fail occurs here. Saving the snapshot and closing the tables
MTEGetSnapshot(Idx, Snapshot, Len);
MTECloseTable(Idx, HSecurs);
MTECloseTable(Idx, HTrades);
...
// Recovery starts. Loading the snapshots and opening the tables
HSecurs := MTEOpenTable(Idx, 'SECURITIES', 'EQBR    ', True,Msg);
// SECURITIES table is opened from scratch, processing the data
...
```

```
HTrades := MTEOpenTableAtSnapshot(Idx, 'TRADES', '', Snapshot,
Len, Msg);
// TRADES table is opened from the snapshot, processing the data
...
repeat
  MTEAddTable(Idx, HSecurs, 0);
  MTEAddTable(Idx, HTrades, 1);
  MTERefresh(Idx, Msg);
  Data := @Msg.Data;
  // Processing the updates
  ...
until Terminated;

MTECloseTable(Idx, HSecurs);
MTECloseTable(Idx, HTrades);
```

CLOSING CONNECTION SESSION

Upon the end of work on the market, the client has to execute the `MTEDisconnect` function.

C++

```
int32 WINAPI MTEDisconnect(int32 Idx);
```

Pascal

```
function MTEDisconnect(Idx: Integer): Integer; stdcall;
```

Arguments:

Idx

Connection handle received with `MTEConnect`, that has to be closed.

Returned value:

One of the `MTE_xxxx` error codes.

Example:

Close the connection with Idx descriptor.

C++

```
int32 Idx; // Initiated by MTEConnect
int32 Err;
...
Err = MTEDisconnect(Idx);
if (Err != MTE_OK)
   fprintf(stderr, "Error: %s\n", MTEErrorMsg(Err));
else
   fprintf(stdout,"Session has ended\n");
```

Pascal

```
Idx: Integer;           // Initiated by MTEConnect
Err: Integer;
...
Err := MTEDisconnect(Idx);
if Err <> MTE_OK then Writeln(MTEErrorMsg(Err)
  else Writeln('Session has ended');
```

ERROR MESSAGES

All the library functions support MTE_xxxx error codes. `MTEErrorMsg` or `MTEErrorMsgEx` functions can be used to get the error code text description

C++

```
char * WINAPI MTEErrorMsg(int32 ErrorCode);
char * WINAPI MTEErrorMsgEx(int32 ErrorCode, char *Language);
```

Pascal

```
function MTEErrorMsg(ErrCode: Integer): LPSTR; stdcall;
function MTEErrorMsgEx(ErrCode: Integer; Language: PAnsiChar): LPSTR;
      stdcall;
```

Arguments:

ErrorCode

One of the MTE_xxxx error codes.

Language

Appropriate language to use in error messages. Possible values are: "`English`", "`Russian`", "`Ukrainian`". If invalid language is specified, English will be used instead. `MTEErrorMsg` function always returns messages in English.

Returned value:

Pointer to an ASCIIZ-string that contains text description of an error.

ERROR CODES

ID	Code	Description
MTE_OK	0	No errors.
MTE_CONFIG	-1	Configuration error: trying to connect to the wrong server, no services specified on a server, wrong parameter values in configuration file.
MTE_SRVUNAVAIL	-2	Server is not available. ASTS Bridge Server is not running, Trading system is not available or connection is disrupted.
MTE_LOGERROR	-3	Could not create log file when calling `MTEConnect`.
MTE_INVALIDCONNECT	-4	Invalid connection handle was given. `MTEConnect` has not been called or `MTEDisconnect` function has already been called.
MTE_NOTCONNECTED	-5	Connection with a given descriptor has been lost due to an error (and not as the result of `MTEDisconnect` function). Error on ASTS Bridge Server, Trading System has been shut down or connection is disrupted.
MTE_WRITE	-6	Error writing to port. Error on ASTS Bridge Server or port connection is disrupted.
MTE_READ	-7	Error reading from port. Error on ASTS Bridge Server or port connection is disrupted.
MTE_TSMR	-8	Error related to the protocol of interaction with the Trading system, or trading system is not available.
MTE_NOMEMORY	-9	Not enough memory to perform the operation.
MTE_ZLIB	-10	Error in compression/decompression of transmitted data.
MTE_PKTINPROGRESS	-11	`MTEAddTable` function has been called without the following call of `MTERefresh`. Other functions cannot be called while the request package is being prepared.
MTE_PKTNOTSTARTED	-12	`MTERefresh` function has been called without the prior call of `MTEAddTable`. The update request package has to be prepared first.
MTE_FATALERROR	-13	An unexpected fatal error has occurred.
MTE_INVALIDHANDLE	-14	Invalid table descriptor. Either the descriptor hasn't been received with `MTEOpenTable` or a table has already been closed with `MTECloseTable`.
MTE_DSROFF	-15	Serial port connection has been disrupted (no DSR signal). Probably the serial cable is damaged or the serial port is

		closed at one of the connecting sides. Available in the old ASTS Bridge versions.
MTE_UNKNOWN	-16	Unexpected error occurred when executing a function.
MTE_BADPTR	-17	Invalid pointer argument has been passed to a one of `MTExxxx`() function.
MTE_TRANSREJECTED	-18	Trading system has processed the request and returned an error code. Transaction has been rejected.
MTE_TEUNAVAIL	-19	Trading system is temporary unavailable. The server attempts to recover the connection with the Trading system, or waits for a trading session.
MTE_NOTLOGGEDIN	-20	Client attempts to execute a request after the server has established a new connection session with the trading session. Client re-connection required.
MTE_WRONGVERSION	-21	Current version of client library is not supported by server.
MTE_LOGON	-30	Wrong login data (USERID, PASSWORD, etc.) provided.
MTE_TOOSLOWCONNECT	-31	Too slow connection channel does not allow to finalize connection/reconnection procedure correctly.
MTE_CRYPTO_ERROR	-32	Encryption/decryption error when creating/verifying the digital signature.
MTE_THREAD_ERROR	-33	The client is trying to use one connection in two threads. For example, trying to call an `MTExxxx`() function while the previously executed `MTExxxx`() function has not finished its operation yet.
MTE_NOTIMPLEMENTED	-34	The requested function is not supported by this version of client library.
MTE_ABANDONED	-35	Returned by `MTEDisconnect` function (called in another thread), in case of working thread has been stopped by calling TerminateThread.
MTE_BADINTERFACE	-36	Returned by `MTEConnect/MTEConnectEx` function if error occurs when loading the interface from the Trading System (e.g. the ASTSBridge version is outdated andt doesn't support current interface format).

APPENDIX 1. BUFFER FORMAT OF THE MTESTRUCTURE, MTESTRUCTURE2 AND MTESTRUCTUREEX FUNCTIONS

The Data field of the `TMTEMsg/MTEMSG` structure, pointer to which is returned by the `MTEStructure` function, has the following format (for the description of basic types e.g. String, Integer, etc, see Appendix 4; in case of `MTEStructure` each String field is preceded with 4 bytes that indicate the length of the following string). Fields and values, passed only in `MTEStructure2` function (similar to `MTEStructureEx` with *`Version=2`* attribute) are marked with red:

```
       Field                  type
TInterface:
       InterfaceName          String
       InterfaceTitle         String
       InterfaceDescription   String       // only MTEStructureEx with Version>=2
       MsgSetNumber           String       // only MTEStructureEx with Version>=4
       EnumeratedTypes        TEnumTypes
       Tables                 TTables
       Transactions           TTransactions
```

The description of information objects consists of three blocks: enumerated types, tables and transactions.

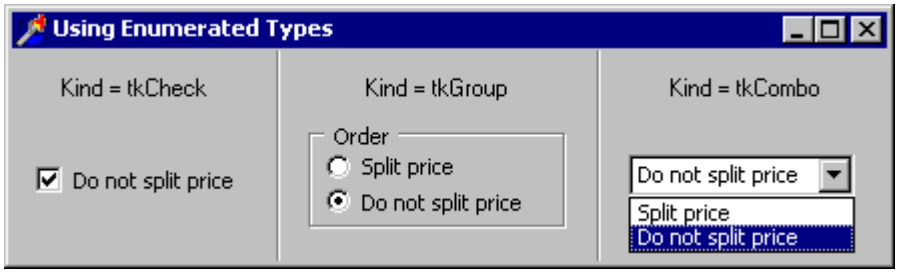
```
TEnumTypes:
       NumberOfTypes          Integer
       Type1                  TEnumType
       Type2                  TEnumType
       ...
       TypeN                  TEnumType

TEnumType:
       Name                   String
       Title                  String
       Description            String       // only MTEStructureEx with Version>=2
       Size                   Integer
       Type                   TEnumKind
       NumberOfConstants      Integer
       Constant1              TEnumConst
       Constant2              TEnumConst
       ...
       ContstantN             TEnumConst

TEnumConst for MTEStructure:
       String                 string       // formatted «Value=LongDescription»


TEnumConst for MTEStructureEx with Version>=2:
       Value                  string
       LongDescription        string
       ShortDescription       string

TEnumKind:                    Integer
       ekCheck = 0
       ekGroup = 1
       ekCombo = 2
```

Enumerated types are used to describe available values of table fields and transactions. A type description looks similar to the following:

```
       'TCurrency'            // Name
       'Currency'             // Description
       4                      // Size
       ekCombo                // Preferred representation - "Type"
       3                      // Number of constants
       'RUR =Roubles'         // Constant 1
       'USD =U.S. Dollars'    // Constant 2
```

```
'EUR =Euro'          // Constant 3
```

"Size" field (=4) indicates the size of available values for the fields of this type.
"Type" field (=ekCombo) assigns the preferred way of field representation that is used during the creation of a form for parameters entry. For example, the ekCombo field type can be presented as a list of values. Available values are illustrated below:



For `MTEStructure` constants consist of two parts – acceptable value (always with the length of "Length") and description of this value. Two parts are separated by the equals sign (=).
For `MTEStructure2` and `MTEStructureEx with Version>=2` constants value and their description are passed in separate fields.

```
TTables:
       NumberOfTables       Integer
       Table1               TTable
       Table2               TTable
       ...
       TableN               TTable

TTable:
       Name                 String
       Title                String
       Description          String        // only MTEStructureEx with Version>=2
       SystemIndex          Integer       // only MTEStructureEx with Version>=2
       Attributes           TTableFlags
       InputFields          TFields
       OutputFields         TFields

TTableFlags:                Integer
       tfUpdateable     = 1
       tfClearOnUpdate  = 2
       tfOrderbook      = 4               // only MTEStructureEx with Version>=2
```

The list of table input fields is used when forming a parameter string for `MTEOpenTable` function.
The list of output parameters allows to parse buffers, returned by `MTEOpenTable` and `MTERefresh` functions.

"SystemIndex" field contains a number of subsystem of ASTS Trading system, which processes current request. Updates packet, formed by `MTEAddTable` calls, may contain only queries with the same "SystemIndex". Currently, for all markets, except for Derivatives market, the index is 0, and all the tables can be updated with a single `MTERefresh` call. There are two subsystems in the derivatives market: Trading system and Risk Management system – so all requests for update should be divided into two packets according to the "SystemIndex".

Table attributes can be combined (i.e. the value will be equal to 3) and have the following values:

tfUpdateable - the table is updateable. Functions `MTEAddTable/MTERefresh` can be used to get updates;
tfClearOnUpdate - the old table contents should be cleared before each update with `MTEAddTable/MTERefresh` functions.
tfClearOrderbook – the table has a orderbook (quotebook) format and should be appropriately processed (see. Working with tables -> Notes).

```
TFields:
       NumberOfFields              Integer
       Field1                TField
       Field2                TField
       ...
       FieldN                TField

TField:
       Name                  String
       Title                 String
       Description           String       // only MTEStructureEx with Version>=2
       Size                  Integer
       Type                  TFieldType
       NumbDecimalPlaces     Integer      // only MTEStructureEx with Version>=2
       Attributes            TFieldFlags
       EnumeratedType        String
       DefaultValue          String       // for input fields only

TFieldType:                  Integer
       ftChar     = 0
       ftInteger  = 1
       ftFixed    = 2
       ftFloat    = 3
       ftDate     = 4
       ftTime     = 5
       ftFloatPoint = 6                  // only MTEStructureEx with Version>=3
       ftMemo     = 7                    // only MTEStructureEx with Version>=5

TFieldFlags:                 Integer
       ffKey       = 0x01
       ffSecCode   = 0x02
       ffNotNull   = 0x04
       ffVarBlock  = 0x08                // only MTEStructureEx with Version>=2
```

Field attributes (TFieldFlags) can be combined and have the following values:

ffKey	Key field. Table rows with the same key field values should be merged in one string.
ffSecCode	This field contains security ID. It's recommended to consider this flag when automating the procedure of counting decimal places in fields of type ftFloat.
ffNotNull	Cannot be null.
ffVarBlock	This field may be repeated several times.

Note. "DefaultValue" field is available only as input field.
"Size" defines the lengths of a field in characters.
"NumbDecimalPlaces" specifies the number of decimal places for fields of type ftFixed.
"EnumeratedType" can contain either name of an enumerated type to which a field refers or an empty string.
"Default Value" can be used when creating a form for parameters entry.
All fields are represented in trading system text format (see `MTEExecTrans`).

```
TTransactions:
       NuOfTransactions      Integer
       Transaction1          TTransaction
       Transaction2          TTransaction
       ...
       TransactionN          TTransaction

TTransaction:
       Name                  String
       Title                 String
       Description           String       // only MTEStructureEx with Version>=2
       SystemIndex           Integer      // only MTEStructureEx with Version>=2
```

```
InputFields          TFields
```

The list of transaction input fields is used when forming a parameter string for the MTEExecTrans function.

APPENDIX 2. BUFFER FORMAT OF THE MTEOPENTABLE FUNCTION

The Data field of the TMTEMsg/MTEMSG structure (pointer returned by the MTEOpenTable function) contains rows of a requested table and has the following format (for the description of the basic types e.g. String, Integer, etc, see Appendix 4):

field	type
TMTETable:	
Ref	Integer
NuOfRows	Integer
Row_1	TMTERow
Row_2	TMTERow
...	
Row_N	TMTERow

"Ref" field is used when requesting updates for several tables simultaneously with MTEAddTable/MTERefresh functions. It contains the value of a third parameter passed to MTEAddTable(Idx, Htable, Ref) function. By value of this field, it's possible to determine which table (HTable descriptor) the received TMTETable structure corresponds to. The "Ref" field value is set to "0" in the buffer returned by the MTEOpenTable.

```
TMTERow:
       NumberOfFields      Byte
       DataLength          Integer
       FieldNumber         Byte[NumberOfFields]
       FieldData           Byte[DataLength]
```

Table rows have variable length and can contain different number of fields.

"NumberOfFields" field contains the number of table fields, present in a given string. If the value is 0 then strings contains all the fields of the table (see. MTEStructure).

"DataLength" field contains the total length of table fields, present in a given string.

"FieldsNumber" field has a variable length. Its size equals to the value of "NumberOfFields" field. This field contains numbers of fields (one byte per number), present in a given string. The number of field corresponds to the number of an output field in the description of information objects (see MTEStructure). If "NumberOfFields" is 0 then "FieldsNumber" is not available and all the fields' numbers should be taken sequentially: 0, 1, 2, 3 … N.

"FieldsData" field (size equals to the size of "DataLength", in bytes) contains set of table fields values. The number of fields is defined by "NumberOfFields" and their total length – by "DataLength". Length and type of each field are defined in the description of an information object (see MTEStructure). All fields are represented in trading system text format (see appendix 5).

Example:

Let the description of information objects, received with MTEStructure, defines the "Trades" table with the following input fields:

```
TRADES                    // "Trades"
  TradeNum: ftInteger(12) // Number of a trade
  TradeTime: ftChar(6)    // Time of a trade
  BuySell: ftChar(1)      // "B" - buy, "S" - sell
  SecBoard: ftChar(4)     // board code
  SecCode: ftChar(12)     // financial instrument code
  Price: ftFloat(9)       // price
  Qty: ftInteger(10)      // quantity of lots
```

The function is invoked:

```
MTEOpenTable(Idx, 'TRADES', '', True, Msg);
```

As the result, Msg.Data field contains the following data:

```
{
  0x00000000,              // "Ref" field
  0x00000002,              // Two rows received
  0x05,                    // First row has 5 fields
  0x0000002F,              // Data length is 47 bytes
  #0#3#4#5#6,              // Numbers of fields 0, 3, 4, 5, 6:
// these are "TradeNum","SecBoard","SecCode","Price","Qty" fields from
description
  '000000120567CETSUSD000000TOD0002579000000000037'
// Fields values: 120567, "CETS", "USD000000TOD", 25.79, 37
  0x03,                    // Second row contains 3 fields
  0x16,                    // Data length is 22 bytes
  #1#3#4,                  // Numbers of fields 1, 3, 4:
// these are "TradeTime","SecBoard","SecCode" fields from description
  '102953CETSUSD000000TOM'
// Fields values: "10:29:53", "CETS", "USD000000TOM"
}
```

APPENDIX 3. BUFFER FORMAT OF THE MTEREFRESH FUNCTION

The Data field of the TMTEMsg/MTEMSG structure (pointer returned by the `MTEOpenRefresh` function) contains several tables from the trading system and has the following format (for the description of the basic types e.g. String, Integer, etc, see Appendix 4):

field	type
TMTETables:	
NuOfTables	Integer
$Table_1$	TMTETable
$Table_2$	TMTETable
...	
$Table_N$	TMTETable

So the buffer can contains several tables. The format of this buffer is described in appendix 2.

APPENDIX 4. BASIC TYPES

MTESRL library uses the following structures to represent basic types:

`Byte`

One byte.

`Integer`

Four bytes in a format of x86 CPU (the little-endian byte goes first).

`String`

Structure as follows:

```
StringLength: Integer
StringText: Byte[StringLength]
```

`Byte[N]`

Byte array of the length of N.

APPENDIX 5. FORMATTING OF A TABLE DATA RECEIVED FROM THE TRADING SYSTEM

ftChar

String of characters right padded with blank spaces to correspond the required length.

ftInteger

Values of fields of ftInteger type (integer numbers) are transmitted in text format and left padded with zeroes to correspond the required length.

ftFloat

(corresponds to the “PRICE” type in text interface specs)

Values of fields of ftFloat type (real numbers) are transmitted in text format without a decimal point. The number of digits after the decimal point in ftFloat type fields for a specific security is defined by the “DECIMALS” field of “SECURITIES” table.

The ftFloat type fields must contain [DECIMALS] number of digits after the decimal point. For example, the number 465.39 for a security with DECIMALS =4 must be represented as “4653900”. The value of “46539” would have been processed by the trading system as 4.6539.

ftFixed

The ftFixed type fields are also passed as text strings without a decimal point. By default, fields of this type have two digits after the decimal point. However, when using MTEStructure2 and MTEStructureEx with Version>=2 (see Appendix 1), the exact number of decimal places is passed.

ftDate

The ftDate type fields are strings with YYYYMMDD format.

ftTime

The ftTime type fields are strings with HHMMSS format.

ftFloatPoint

(corresponds to the “FLOAT” type in html interface specs)

Values of fields of FloatPoint type (real numbers) are transmitted in a text format with the decimal point and should be supplemented to the required value with zeros on the left. This type is available when obtaining information objects structure with MTEStructureEx function with Version>=3 attribute (see. Appendix 1). In case of using MTEStructure and MTEStructure2 the type is transmitted as a string (ftChar). Decimal point position is not strictly regulated. Decimal point and, if needed, sign of the number (positive or negative), are considered at length calculation. For example: ftFloatPoint(9): "001.45712", ftFloatPoint(16): "-0000012071000.5".

ftMemo

String of characters of arbitrary length. This type is available when obtaining information objects structure with MTEStructureEx function with Version>=5 attribute (see. Appendix 1). In case of using MTEStructure and MTEStructure2 the type is trimmed to the size specified and transmitted as a fixed length string (ftChar). The structure containing:

- *MemoLen* – string length in text format and left padded with zeroes to correspond the length specified;
- *Memo* – character sequence of the length *MemoLen* bytes

For example: ftMemo(6): "000015Hello, world!!!" stands for 15 byte length string "Hello, world!!! ".

Note:

An empty value (NULL) can be specified in a field of any type; for this, a string of all blanks of the required length is used.