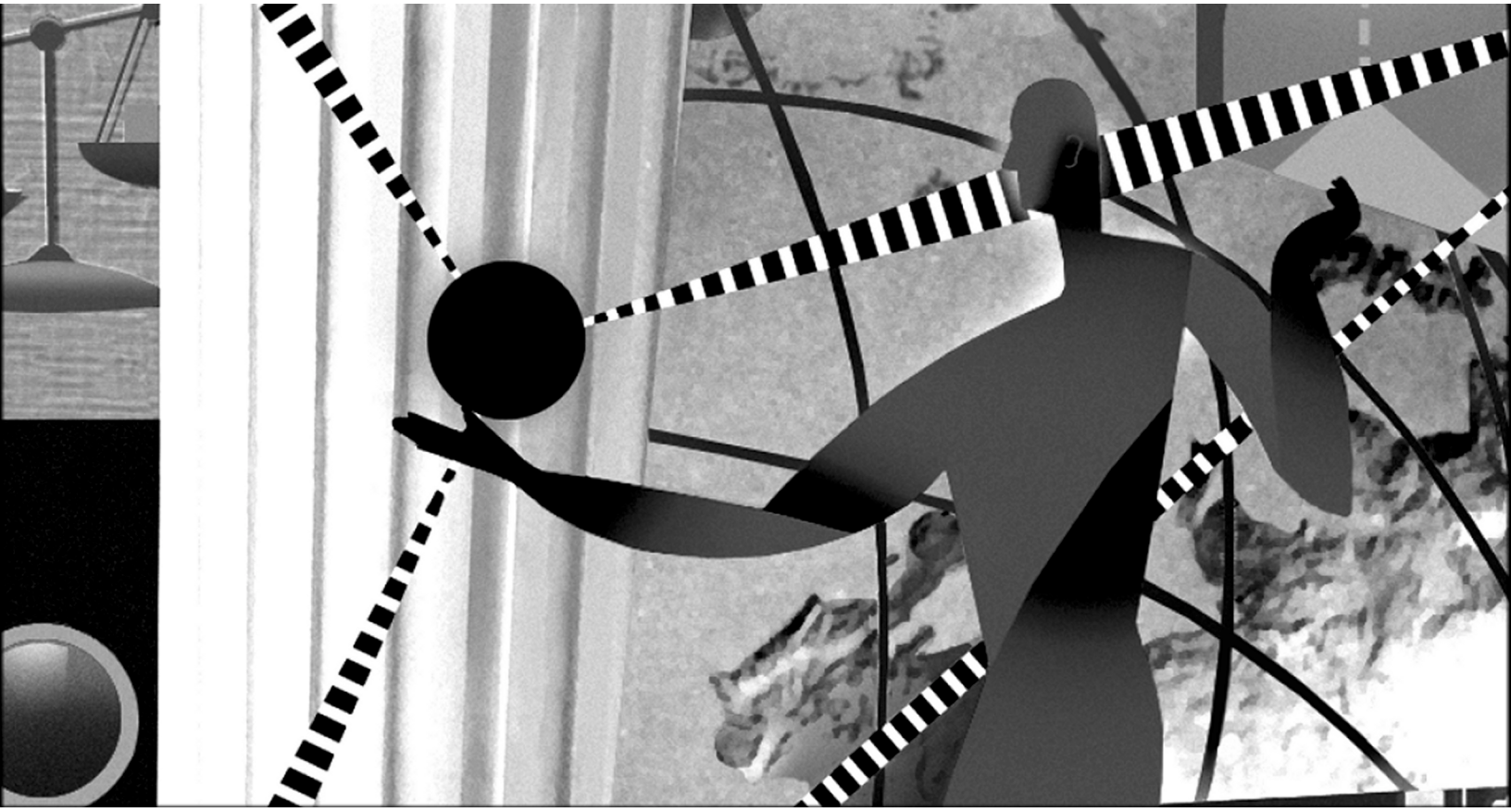
OASYS GLOBAL™ DIRECT



MESSAGE DELIVERY SYSTEM TCP/IP API PROGRAMMER'S GUIDE

Version 3.4.2

6 February 2001

THOMSON FINANCIAL

esg

Thomson Financial ESG
22 Thomson Place
Boston, MA 02210.

Printing: 2/6/01

OASYS Global *Direct* Message Delivery System TCP/IP API Programmer's Guide

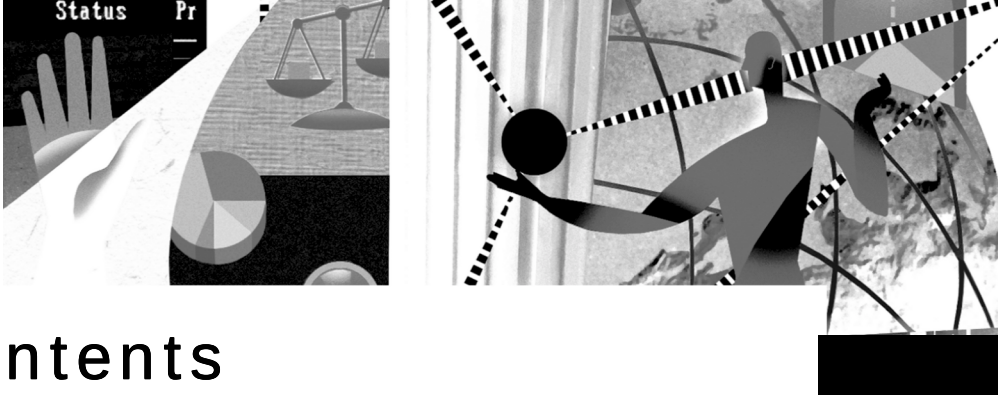
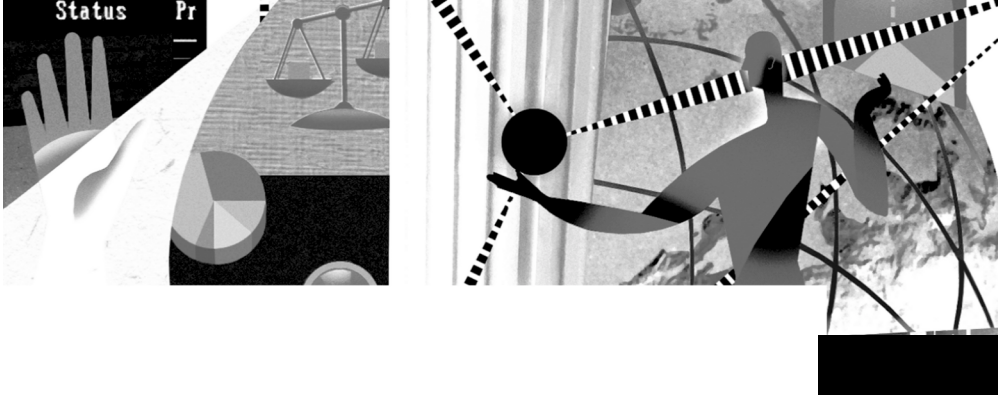


Table of Contents



Preface

The Message Delivery System (MDS) transports MT511 messages needed for trade allocation and confirmation between trade parties to and from the Thomson Electronic Settlements Group (ESG) host. You can access the MDS using ESG's supplied Application Programming Interface (API), called the Messaging Open API (MOA).

The MOA, which provides function calls to access MDS services, is available in platform dependent libraries. ESG currently supports a MOA for selected UNIX and Microsoft NT platforms. The MOA's function calls are statically or dynamically linked into the client application and enable that application to use the MDS services for connecting to the MDS to send messages to and receive messages from other parties. This document provides information required to interface with the MDS.

Intended Audience

This document is directed toward your systems analysts, programmers, and others involved in implementing the link between your internal systems and OGD.

How This Manual Is Organized

This manual contains the following chapters and appendices:

- Chapter 1, "Message Delivery System Overview," describes the MDS and its usage.
- Chapter 2, "Messaging Open API (MOA)," describes the MOA environmental requirements as well as message handling rules and responsibilities. It provides a summary of MOA services calls, explains known limitations, and shows how message flow works.
- Chapter 3, "MOA Configuration," describes MOA configuration at a client site and contains information needed for communications and internal MDS purposes used by all MOA services.
- Chapter 4, "Programming Interface," describes MOA services as they are used in the MDS.
- Appendix A, "Glossary of Terms," defines the MDS-specific terms used in this document.
- Appendix B, "MOA Return/NAK Codes," includes information on MOA error codes and NAK codes.

Typographic Conventions

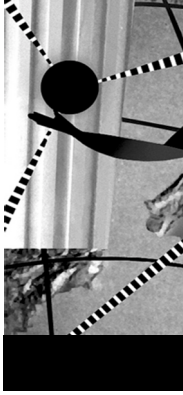
Unless otherwise noted in the text, this manual uses the following typographic conventions:

`Monospace`	Commands, printed text examples, function names and parameters, constants, variables, field names, literal values, return values, arguments, transaction names, configuration parameters, default values, format strings, MT511 tags and assigned values, path variables and paths, and C code samples; for example: `OASYS LOG REPORT`
`Monospace Bold`	Data format specifications. For example, **`dd-mmm-yy`** date format.
Italics	Trade and message statuses (for example *Reject, Affirm, Cancel*).
UPPERCASE	Electronic Trade Confirmation acronyms (such as ETC and API), and message types (such as BLIM).
UPPERCASE ITALICS	MT511 message types (for example, *AE*, *CN* (*CNA CNB*), and *TA*), and return codes (for example, *SUCCESS*, *FAILURE*).
Bold	File names (such as **import.dat** and **trans.map**), and library names (such as **wsock32.dll** and **moa.lib)**.

Related Documents

These are other Thomson ESG documents related to this publication:

- *OASYS Global Direct MT511 Messaging Specification*
- *OASYS Global Direct MT511 Parser API Programmer's Guide and Reference*
- *OASYS Global Direct Broker and Institution Conformance Requirements*
- *OASYS Global Direct Migration Guide for OASYS Global Automated Workstation Clients*
- *OASYS Global Direct Sample MT511 Data - Block and Contract Level Data Flow Examples*
- *OASYS Global Direct Overview*
- *OASYS Global Direct Release Notes, Version 3.4.2*



Thomson Financial ESG

Thomson Financial ESG is the worldwide leader in providing technology-based workflow solutions to the global investment community. Thomson Financial ESG is committed to partnering with its clients to dramatically reduce the risk and cost of processing trades, and thereby improving their investment performance. Thomson Financial ESG services are used in 37 countries for domestic and cross-border trading. Thomson Financial ESG supports approximately 3,450 clients from 18 offices located in major financial centers across the globe.

Thomson Financial ESG is part of Thomson Financial (TF), a leading provider of information services and work solutions to the worldwide financial community. TF employs more than 7,000 people in more than 40 locations dedicated to the success of our clients and is part of The Thomson Corporation (TTC). With annual revenues approaching US$6 billion, TTC is one of the world's leading information and publishing companies. TTC's common shares are traded on the Toronto, Montreal and London stock exchanges

Thomson Financial ESG's products and services include ALERT®, ALERT*Direct*™, OASYS Global™, OASYS Global *Direct*™, OASYS™, OASYS *Direct*™, MarketMatch™, Thomson Report™, and AutoMatch™.

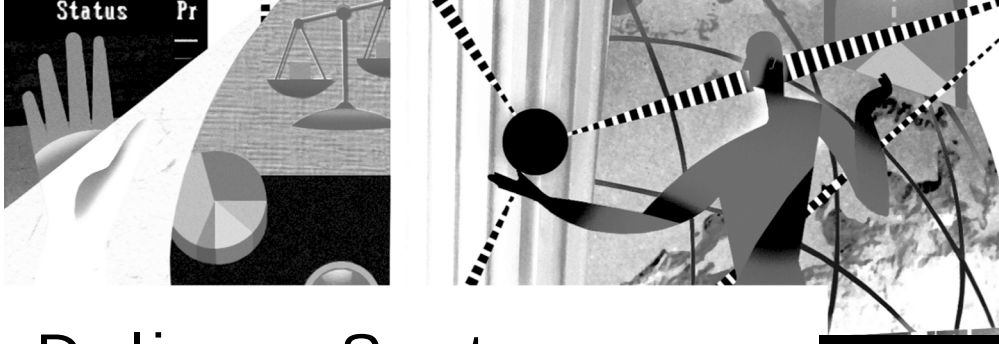
Sales Executives

Call your sales executives to learn more about other Thomson Financial ESG services that can automate your post-trade operations, including ALERT®, ALERT*Direct*™, AutoMatch™, OASYS Global™, OASYS Global *Direct*™, OASYS™, OASYS *Direct*™, MarketMatch™, and Thomson Report™. For assistance call the following sales offices.

Boston	1-800-407-4264	Paris	33-1-4453-7878
Chicago	1-312-629-0957	San Francisco	1-415-989-9797
Edinburgh	44-131-220-8417	Sao Paulo	5511-816-5022
Frankfurt	49-69-971-75200	Singapore	65-295-6383
Hong Kong	852-2905-3145	Stockholm	46-8-587-67160
London	44-207-369-7349	Sydney	612-9225-3158
Mexico City	525-535-6070	Thailand	001-800-65-1555
New York	1-212-612-9604	Tokyo	813-5218-6621

Electronic Trade Confirmation Code of Practice

See the "Electronic Trade Confirmation Code of Practice" section in *OASYS Global Direct Overview* for more information about the Electronic Trade Confirmation Code of Practice.



1: Message Delivery System Overview

This chapter describes the MDS and its usage. It contains the following sections:

The Message Delivery System (MDS) is a data communication system that enables development of applications that have to communicate with each other.

ESG applications exchange information with their clients by messages. For example, when you enter an allocation, as an institution, a message travels from your processing environment (workstation or host) to the Thomson processing center. The processors at Thomson save the information in a trade database, perform value added processing (adding delivery instructions, security code translation, etc.), and send a copy of the message to the broker's processing environment (workstation or host).

The MDS enables this message exchange among its users. The users of the MDS are ESG application hosts (i.e. OASYS Host and OASYS Global Host), and ESG clients' hosts.

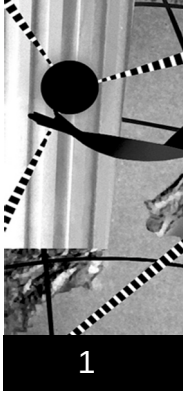
The services provided by the MDS to its users free the application developer from contending with a large number of connectivity issues. Because the MDS handles the underlying network protocols and network topology, application developers can concentrate on business issues rather than on technology issues.

Message Delivery System Functionality

The purpose of the MDS is to transport messages produced by business application programs, including your system. The MDS guarantees complete and error-free delivery of your messages. Guaranteed message delivery means that the MDS takes responsibility for delivering your message. The MDS neither loses messages, nor delivers them more than once, without letting the receiver know.

The MDS delivers guaranteed messages in a “store-and-forward” fashion. In other words, the message recipient does not need to be online when you send your message. This also means that if you do not log in to check your messages, there can be a long delay between the time your counterparty sends a message and the time you read the message.

The MDS delivers messages in “first-in-first-out” (FIFO) order. You may receive messages from multiple senders, but the MDS delivers all messages from one counterparty to you in the order that it received them.



Scope of the Message Delivery System

The MDS encompasses the message delivery switch (MDSwitch), an interface protocol, and an MDS Application Programming Interface, called the Messaging Open API (MOA), to enable business applications to easily interface with message delivery services.

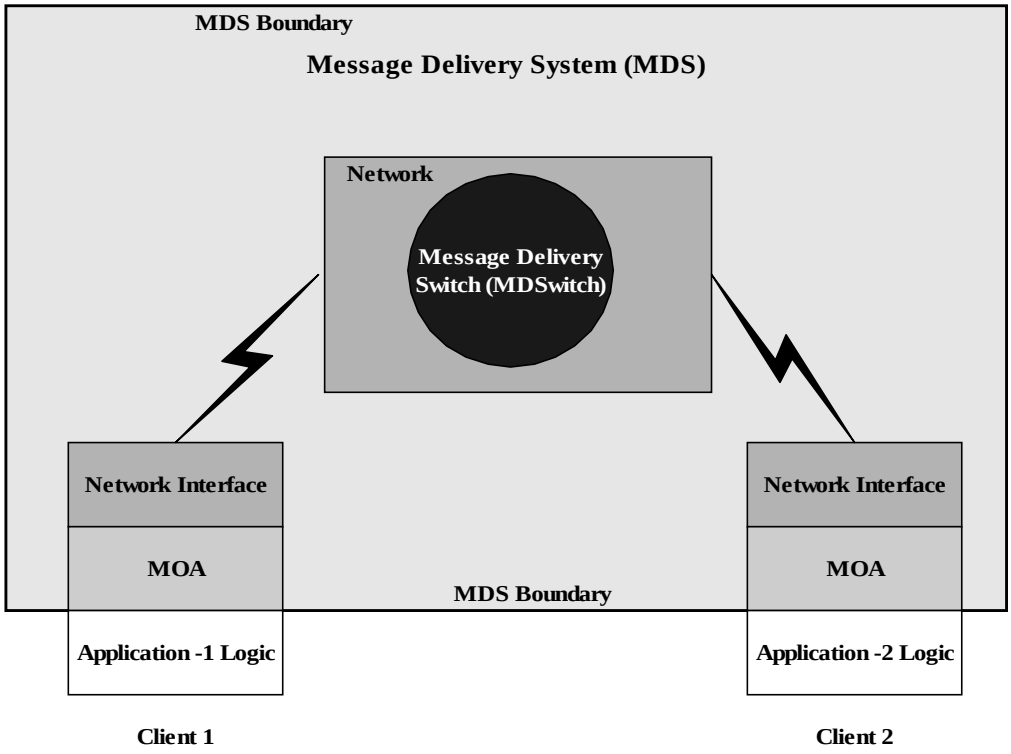
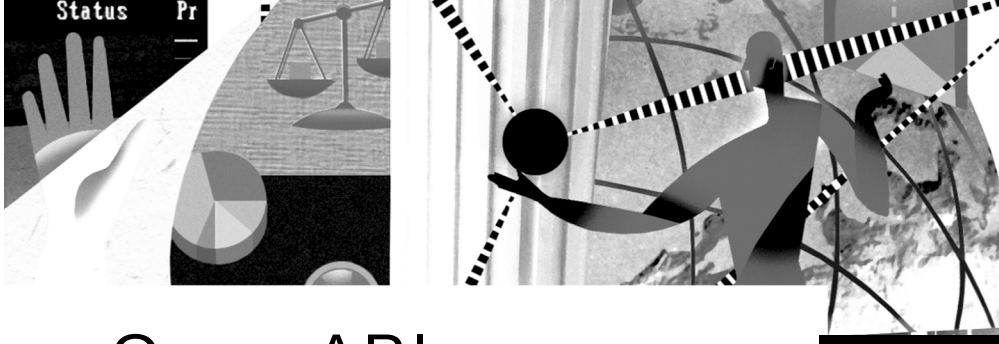


Figure 1.1

Thomson ESG provides the MOA software to clients who wish to make use of the MDS to interface to one or more ESG business applications. The MOA supports only a TCP/IP networking environment.



2: Messaging Open API (MOA)

This chapter provides a more in-depth look at the Messaging Open API (MOA). It addresses MOA environmental requirements as well as message handling rules and responsibilities. It also provides a summary of MOA services calls, explains known limitations, and shows how message flow works. It contains the following sections:

Environment Requirements

You may use the MOA on Sun Solaris version 2.5.1 (Y2K patch applied) using ANSI C compiler (e.g. Sun WorkShop 4.2 or greater) and/or Windows NT 4.0 with Service Pack 5 applied using MS Visual C++ version 5.0.

The MDS assumes that there is an established network over which TCP/IP can operate. It supports local area networks (LANs), leased lines, and backup lines.

Message Handling Rules and Responsibilities

Before an application that transmits and processes messages via the MDS acknowledges a message, that message goes through various phases. During this process, the MDS and the applications using the MOA must ensure that they send and receive messages in the appropriate manner. Here is a summary of the rules and responsibilities of MOA message handling:

- It will never lose a message in end-to-end transit.
- The MDS is responsible for the message from acknowledgment (ACK) to the sender to receipt of receiver's ACK.
- If any problems occur outside the sending and the receiving of the ACK, the application is responsible for troubleshooting and resolving the problem.
- After receiving a message from the MDS, the application must safe-store the message before sending an ACK to the MDS.
- Either the sender or the MDS can duplicate messages. If the MDS retransmits a message due to recovery from communication or system failures, then it marks that message as a “Possible Duplicate.”
- The application must be ready to identify the incoming messages as duplicates, based on the message contents.
- The MDS delivers messages it has received on a single stream (this identifies all the messages from one originator [sender] to the MDS) in FIFO order. An institutional client can send parts of a block trade (for example, the allocations) as consecutive messages. FIFO means that the MDS will send the received allocations to the destination (ESG Host) in the same order that it received them.
- When you receive these messages, they may be intermixed with messages from other senders.

MOA Summary

The MOA is a straightforward interface. Applications establish a session with the MDS and then send messages to and receive messages from it. The MOA consists of a number of functions, referred to as services, that it uses to interact with the MDS. The services currently offered by the MOA include the following:

Function/Service	Description
`MOA_open`	Starts a session with the MDS.
`MOA_close`	Closes the session previously established with `MOA_open`.
`MOA_send_msg`	Sends a message to the MDS.
`MOA_rtrv_no_ack`	Retrieves a message from the MDS and instructs the MOA not to acknowledge the message. The application will later acknowledge the retrieval of the message.
`MOA_send_ack`	Used by the application to report to the MDS when the application has completed processing of the message or at least safe-stored it.

For complete details on these services, including descriptions, common parameters, return codes, and examples of their use, see the section titled “Messaging Open API (MOA) Services” in Chapter 4, “Programming Interface.”

Message Flow

This section describes the overall flow of messages between the parties and how the MOA manages this process. See Appendix A, “Glossary of Terms,” for definitions of key terms.

Order and Priority Rules

As the MDSwitch receives messages within a logical stream, it stores them on the destination queue in FIFO order. Only a single application may service any given queue.

Session Management

Before an application can send messages to or receive messages from the MDS, it must establish a session. To do so, the application calls the `MOA_open` service.

The MOA maintains sessions transparently to the application. If a session breaks between two I/O requests, the MOA attempts to reopen it. This capability makes `MOA_open` an optional call. However, using this call allows easier handling of communication errors by detecting them before data exchange.

The MOA assigns sequence numbers to messages to guarantee that it does not lose them. When a session starts, the two session partners—the MOA and the MDSwitch—coordinate the sequence numbers to use and verify that the sender’s sequence number is the one expected by the receiver. When a session synchronization effort fails due to unmatched sequence numbers, the session fails and the API software takes measures to correct the unmatched sequence number automatically. A subsequent attempt to establish a session with the MDSwitch normally results in a proper connection.

When calling `MOA_open`, make three attempts, sleeping two seconds between each attempt, before giving up, unless an error other than “comm problem,” “bad sequence number,” or “open error” is received. If a different error is received, an operator must intervene to troubleshoot the problem.

Before an application can conclude its message exchange with the MDSwitch, it must call the `MOA_close` service to close the active session.

Sending a Message

To send a message, the application must do the following:

1. Create the message.
2. Complete the input parameter structure with information such as message content, the message recipient, and special handling instructions.
3. Mark the message as being in the process of being sent.
4. Call the `MOA_send_msg` service (step 1 as illustrated below).
5. Wait for the service completion.
6. Mark the message as sent.

The following diagram shows the steps that occur when an application sends a message. The paragraphs that follow the diagram explain the process in detail.

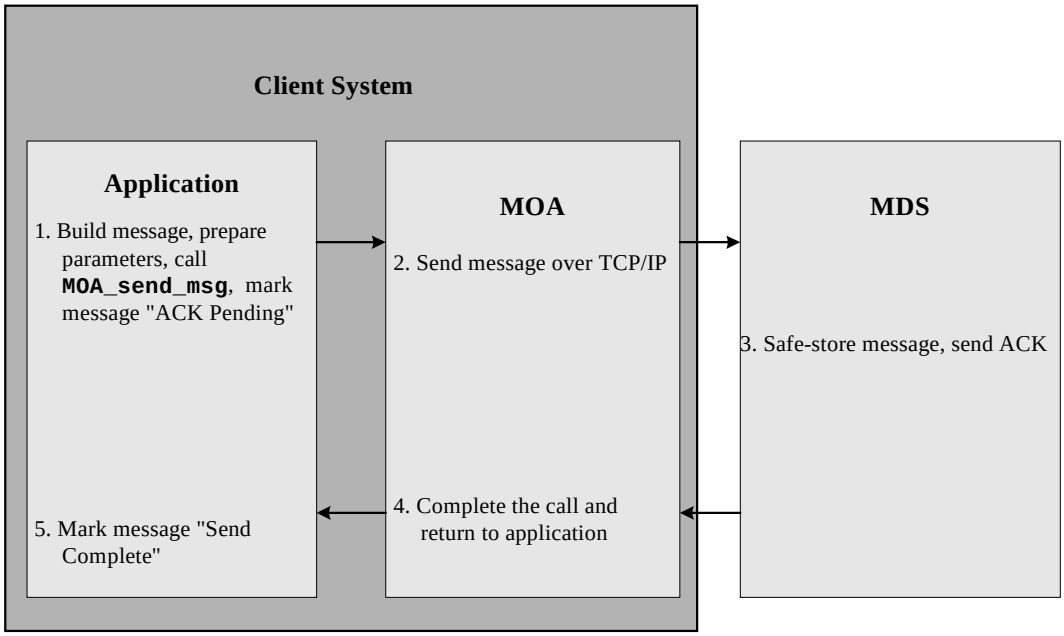


Figure 2.1

MOA formats the message, wraps it in the TCP/IP protocol and sends it to the MDSwitch (step 2). After MOA sends the message, it blocks delivery/receipt until either a positive (ACK) or negative acknowledgment (NAK) is received from the MDSwitch, or the session terminates due to communication or system problems.

A positive acknowledgment (ACK) indicates that the MDSwitch safe-stored the message and placed it on the appropriate destination queue (step 3). The MDSwitch delivers the message to its final destination after the destination system establishes a session with the MDS and requests receipt of the messages queued to it.

A negative acknowledgment (NAK) indicates that the MDSwitch could not process the message and rejected it. If this occurs, the MDSwitch will send back to the caller the reason (i.e. the NAK code) for rejecting the message. Common reasons for rejection are that the message has corrupted headers or mismatched numbers. If either of these conditions occur, an operator must intervene to troubleshoot the problem. In many cases your technical support operator may have to update the MOA configuration file and then re-send the message. If the caller receives any other NAK code, the application should close the session and the technical

support operator should report the situation, with as many details as possible, to the appropriate ESG support personnel.

Once an ACK is received from the MDSwitch (step 4), the `MOA_send_msg` call is finished and the MOA returns the operation result to the application (step 5). If the message was acknowledged positively, the application should mark the message as successfully transmitted and change the message state to sent. If the message was rejected, the application should examine the rejection reason and try to recover (as described above).

The process of marking a message as sent or rejected is a conceptual one. There is no MOA call to perform this marking. It is suggested that your application assign a state to each message.

If the application terminates before `MOA_send_msg` completes, the application cannot determine whether the MDSwitch received the message or not. In this case, the application must re-send the message after a new session is established. The application should flag the message as “Possible Duplicate” to indicate a possible re-transmission. This flagging is literal; part of the information passed in a MOA call includes this “Possible Duplicate” flag. The exact syntax is described in the section titled “Control Structures” in Chapter 4, “Programming Interface.”

Receiving a Message

The following diagram shows the various steps that occur when an application receives a message. The paragraphs that follow the diagram explain the process in detail.

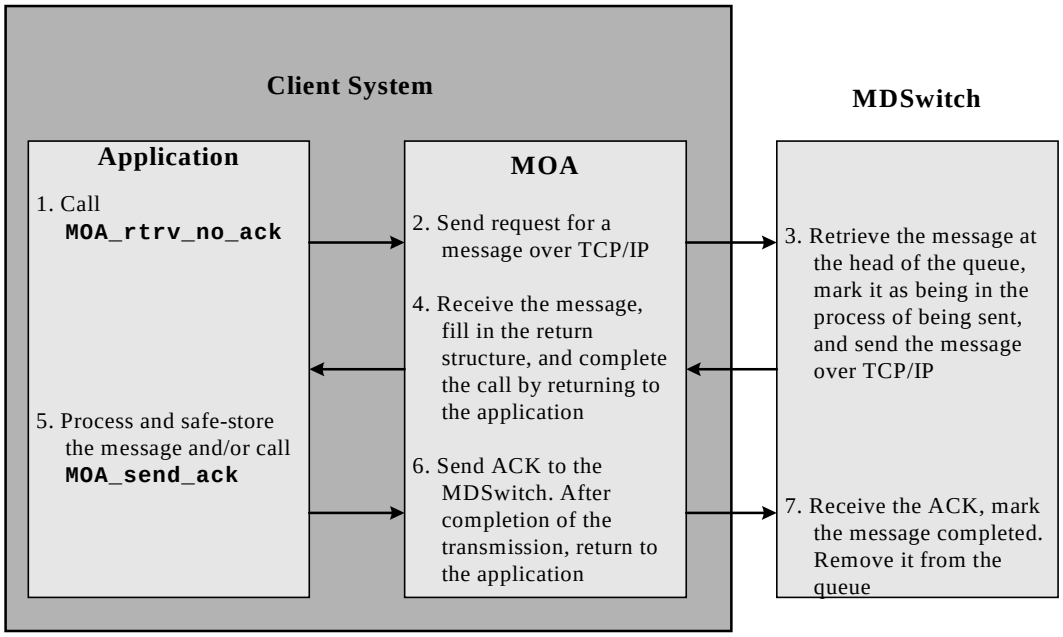


Figure 2.2

To get a message, your application calls `MOA_rtrv_no_ack` (step 1). The MOA forms the request packet and, without relinquishing control, transmits it across the network to the MDSwitch (step 2). The MDSwitch replies to the request either with a "data" or a "no data" indicator (step 3). The MOA receives the reply and propagates it up to the application layer through the `MOA_rtrv_no_ack` call return code (step 4). The application never receives unsolicited messages.

If no message is available, the application program must periodically call `MOA_rtrv_no_ack` until it receives a message or closes the session. To avoid overloading the MDSwitch with requests, an application interface should include a parameter for the interval between (unfulfilled) retrieval requests. The recommended value for this parameter is 1 to 4 seconds.

If a message is available, the MDSwitch marks it as "Pending Acknowledgment," sends it, and blocks delivery of all other messages on that message's logical stream. The MOA receives the message and, if it comes across a sequence number or format error, sends a NAK to the MDSwitch. Otherwise, it passes the message back to the application.

The application processes, and/or safe-stores, and acknowledges the message by calling `MOA_send_ack` (step 5). When the MDSwitch receives the ACK for the message, it removes it from the receiver's queue, and the operation is considered successfully completed (step 7).

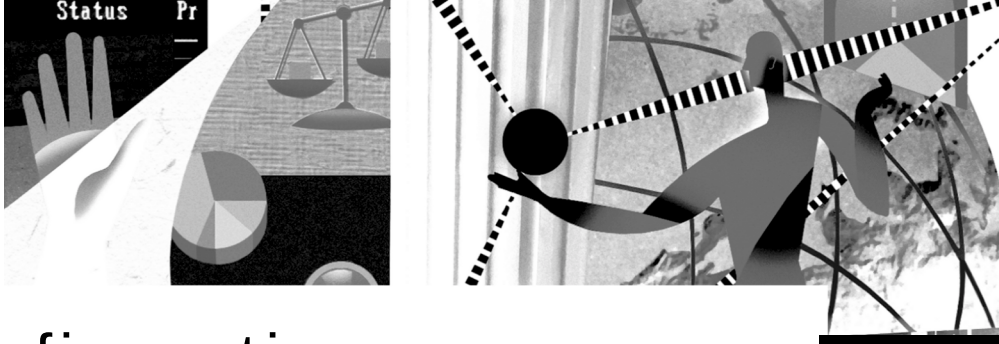
When the application cannot process the message, for example if there is no disk space available to store the message, the application must close the session by calling the `MOA_close` service.

If the MDSwitch does not receive an ACK within a configured time interval, it closes the session, flags the message as "Possible Duplicate," and leaves it on the destination queue. The next time the destination receives the same message, it establishes a session and requests a message.

Known Limitations

These are the known limitations of the MOA:

1. The maximum message length is 32K (no MT511 message exceeds this length).
2. The MOA only allows a window of one message. The application cannot request another message until the previous request completes (i.e. it receives an ACK). Your application should acknowledge back to the MDS upon message safe storage.
3. Message text may not contain the NUL (0x0) character.
4. After your application closes the session by calling the `MOA_close` service, wait 30 seconds for the port to close before calling the `MOA_open` service.
5. There can no more than 19 OGD acronyms associated with a comm port.
6. A two comserve design, one for inbound and one for outbound, processes messages much more efficiently than a one comserve design.



3: MOA Configuration

This chapter describes MOA configuration at a client site, and contains information needed for communications and internal MDS purposes used by all MOA services. It contains the following sections:

You must provide a configuration file for each communication stream. Thomson ESG provides the data required to build this file.

All the MOA services use a configuration file containing information needed for communications and internal MOA purposes. You must set up this file before ESG activates your client application that uses the MOA services. The file contains data in binary format. It must be present in the application's current directory, and its name must be <COMSERV_NAME>.DAT (all upper-case).

To create the binary configuration file, use any editor to create an ASCII text file. Next, use the `MOA_set` function to convert the ASCII text file to a binary configuration file.

Since the MOA services also write to the binary configuration file, once you have used a service the file’s contents will no longer be consistent with the ASCII text file. If you need to make changes, you must first use the `MOA_set` function to convert the binary configuration file back to ASCII.

ASCII File Structure

The ASCII text file contains lines of data in which each line has the following elements:

1. The name of the configuration parameter.
2. An equals sign (=).
3. A value for the parameter.

The following table describes the various code words and their attributes. This table uses the following abbreviations for the type: A for Alphabetic, N for Numeric, AN for alphanumeric, and X for hexadecimal.

Parameter	Type	Length	Description	Recommended Initial Value
comserv	AN	8	The MDS ComServ name with which the client application communicates.	ESG will supply information for each stream.
timeout*	N	4	Time-out period for TCP/IP commands in seconds.	60
in_socket*	N	4	Input socket ID used for MOA internal purposes.	0
out_socket*	N	4	Output socket ID used for MOA internal purposes.	0
session_nr	N	4	Current MDS session number.	0
seq_nr	N	4	Current MOA outgoing message sequence number.	0
acked_seq_nr	N	4	Last acknowledged outgoing message sequence number.	0
expected_seq_nr	N	4	Next expected incoming message sequence number.	0
input_port	N	4	The TCP/IP input port number (output port number in the MDSwitch).	ESG will supply information for each stream.
output_port	N	4	The TCP/IP output port number (input port number in the MDSwitch).	ESG will supply information for each stream.
stage*	N	2	Internal MOA value.	1
ip_address	N	20	The IP Address of the MDS.	ESG will supply information for each stream.
sep_len	N	1	The line separator length (for example, for CRLF, `sep_len = 2`).	2
separator	X	10	Line separator in ASCII hexadecimal code (for example, for CRLF, separator = `0d0a`).	0d0a
insok_activ*	N	1	Indicator used for MOA internal purposes.	0
outsok_activ*	N	1	Indicator used for MOA internal purposes.	0

Parameter	Type	Length	Description	Recommended Initial Value
display_flag	A	1	Y: MOA services will write debug/trace information to a file. N: MOA services will not write debug information.	Y for testing. N for production.
reconnect_period	N	4	The number of seconds to wait between attempts to open a session.	20
connect_tries	N	4	The number of attempts to make to open a session.	5
authen_active*	AN	N/A	Indicator used for MOA internal purposes.	Empty.
binary_data*	AN	N/A	Indicator used for MOA internal purposes.	Empty.
for_future_use*	AN	N/A	Indicator used for MOA internal purposes.	Empty.

Notes!1. You do not need to include the fields marked with asterisk (*) in the initial data file. However, they will appear after binary data-to-ASCII text conversion.

2. The system ignores empty lines.

3. The system ignores blanks before, between, or after elements.

4. The system does not permit embedded blanks in value elements of type N or X.

5. The system allows comments. A line starting with the character # identifies a comment. The comment exists only in the ASCII file and disappears during conversion to the binary configuration file.

6. There is no significance to the order in which the parameters appear.

Here is an example of a configuration file:

```
comserv = SUB00001
timeout = 60
in_socket = 0
out_socket = 0
session_nr = 0
seq_nr = 0
acked_seq_nr = 0
expected_seq_nr = 0
input_port = 1
output_port = 2
stage = 0
ip_address = 164.179.103.35
sep_len = 2
separator = 0d0a0000000000000000
insok_activ = 0
outsok_activ = 0
display_flag = N
reconnect_period = 20
connect_tries = 5
authen_active =
binary_data =
for_future_use =
```

MOA_set Program

The `MOA_set` conversion program supports two modes of conversion:

- ASCII text file to binary configuration file
- Binary configuration file to ASCII text file

You must name the binary configuration file <COMSERV NAME>.DAT (all upper-case). The ASCII text file may have any name; we recommend that you name it <COMSERV NAME>.TXT

When running `MOA_set`, you must supply three arguments as follows:

Argument Name	ASCII to Binary Conversion	Binary to ASCII Conversion
`Conversion type:`	a-b	b-a
`Input file path and name:`	ASCII file	binary file
`Output file path and name:`	binary file	ASCII file

Examples:

1. ASCII to binary conversion:

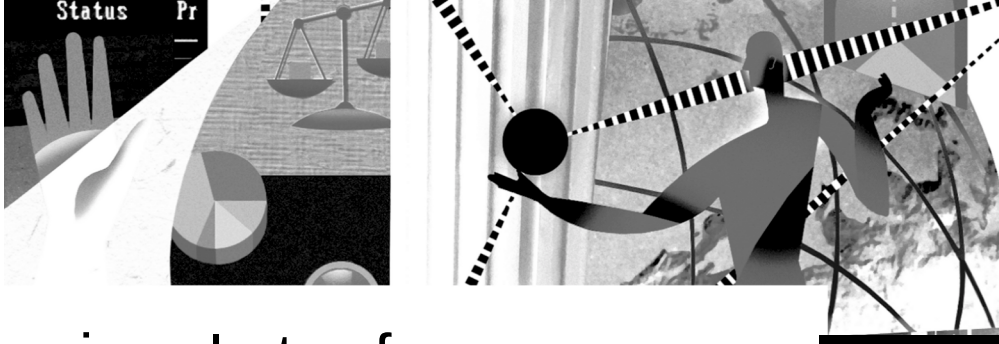
```
MOA_set a-b COMSERV1.TXT COMSERV1.DAT
```

The program displays either “a-b conversion ended successfully” or a self-explanatory error message to the standard output.

2. Binary to ASCII conversion:

```
MOA_set b-a COMSERV1.DAT COMSERV1.TXT
```

The program displays either “b-a conversion ended successfully” or a self-explanatory error message to the standard output.



4: Programming Interface

This chapter describes MOA services as they are used in the MDS. It contains the following sections:

The MOA consists of a number of services it uses to communicate with the MDSwitch. To call these services, the application includes specific parameters such as address information and length of message. Two of these parameters are control structures (`moa_param_i` and `moa_param_o`). The following sections describe the MOA parameters in detail.

MOA Parameters

The MOA services all use a set of common parameters (defined in the **MOA_call.h** and **MOA_bas.h** header files). The following table describes each of these parameters:

Parameter	Type	Description
moa_msg	Char*	A message buffer maintained by the application program. When sending, this points to the outgoing message. When receiving, this points to a buffer to hold the incoming message. The size of `moa_msg` should be large enough to handle the maximum message length received. When sending, you must format the message according to the format appearing in the input parameters structure. When receiving, the message will be formatted according to the format in the output parameter structure.
moa_msg_len	Long	When sending, this field contains the length of the message pointed to by `moa_msg`, not including any 0-termination character. When receiving, this field is the length of `moa_msg`, not including any 0-termination character (note that MOA neither allows nor adds 0-termination characters). The maximum value of this field is 32,000 (`MOA_MAX_BUFF_LEN` as defined in **MOA_bas.h**). MOA does not alter this field.
moa_key	MOA_KEY*	A structure maintained by the application program. This field, reserved for future use, will authenticate messages. You must initialize this structure to all blanks.
moa_work_area	MOA_WORK_AREA*	A structure maintained by the application program for internal use by the MOA. The application program must not change the contents of this structure in any way, although it must initialize it to 0x0 (i.e. `(memset (moa_work_area, '\0', sizeof (MOA_WORK_AREA))`).
moa_param_i	MOA_PARAM_I*	The Input Message Parameters structure maintained by the application program. This structure passes values to the MOA services. See the section titled "Input Message Parameters Structure (moa_param_i)" in this chapter for more information on the fields of this structure and their requirements. You must initialize this structure to all blanks.
moa_param_o	MOA_PARAM_O*	The Output Message Parameters structure maintained by the application program. The MOA uses this structure to pass information back to the application program. See the section titled "Output Message Parameters Structure (moa_param_o)" in this chapter for more information on the fields of this structure and their requirements. You must initialize this structure to 0x0 (i.e. `memset (moa_param_o, '\0', sizeof (MOA_PARAM_O))`).

Control Structures

4

The following sections describe the configuration file, the `moa_param_i` input message parameter control structure, and the `moa_param_o` output message parameter control structure.

MOA Configuration

MOA services use a configuration file containing information needed for communications and internal MOA purposes. This file must be ready for use before activating a client application that uses MOA services. The file contains data in binary format. It must be present in the application's current directory, and its name must be <COMSERV_NAME>.DAT (all upper case).

To create the binary configuration file, create an ASCII text file using any editor then use the `MOA_set` program to convert the ASCII text file to a binary configuration file.

Since the MOA services also write to the binary configuration file, once an application has used a service the file’s contents will no longer be consistent with the ASCII text file. If you need to make changes, use the `MOA_set` program to convert the binary configuration file back to ASCII.

ASCII File Structure

The ASCII text file contains lines of data in which each line is comprised of the following elements:

1. The name of the configuration parameter.
2. An equals sign ('=').
3. A value for the parameter.

Input Message Parameters Structure (moa_param_i)

Your application program maintains the `moa_param_i` structure. The application supplies it to the MOA in every service call. This structure contains various parameters used by the MOA to construct the minimum MDS headers required for each request type. See the section titled "Messaging Open API (MOA) Services" in this chapter for more information on mandatory fields for specific MOA service calls. To override the default values for a given parameter, the application must supply the optional parameters. The MOA never alters the contents of this structure.

Note! The client application program must initialize the `moa_param_i` structure to all blanks.

The following table lists the fields that comprise the `moa_param_i` structure and describes how the MDS uses them. The following table uses these abbreviations in the columns representing the services (`open/close`, `send_msg`, `rtrv_no_ack`, and `send_ack`): M for mandatory, O for optional, and N for not used. Unused (by any service) fields are gray. All field values are of type `char`.

Name	Length	Description	open/close	send_msg	rtrv_no_ack	send_ack
comserv[1]	8	MDS ComServ name assigned to application.	M	M	M	M
msg_fmt	4	Format type of message contained in message text field. Must be set to `MOA_FMT_MDS`.	N	M	N	N
owner	8	Not used.	N	N	N	N
sender	32	Sender of message.	N	M[2]	N	N
receiver	37	Receiver of message.	N	M[3]	N	N
rfk	16	Message Reference Key (RFK) to be assigned to message by MDS.	N	M[4]	N	N
trn	21	Message Transaction Reference Number (TRN) to be assigned to message by MDS. This is unique for every trade.	N	M[5]	N	N
next_act	8	Last activity performed on message in application. This might affect message flow within MDS.	N	M[6]	N	N
pdm	1	Possible Duplicate Message indicator. Set to `MOA_POS_DUP_Y` when application may have sent a message more than once.	N	O[7]	N	N
int_prio	6	Not used.	N	N	N	N
net_prio	6	Not used.	N	N	N	N

Name	Length	Description	open/close	send_msg	rtrv_no_ack	send_ack
rec_prio	6	Not used.	N	N	N	N
val_date	8	Not used.	N	N	N	N
cur	3	Not used.	N	N	N	N
amount	15	Not used.	N	N	N	N
net_dels	8	Not used.	N	N	N	N
net_coms	8	Not used.	N	N	N	N
net_mt	4	Must be set to `mdsa`.	N	M	N	N
int_mt	6	Not used.	N	N	N	N
user_1	32	Indicates type of message initiator .	N	O [8]	N	N
act_1	8	Specifies activity performed by initiator indicated in field `user_1`.	N	O [8]	N	N
user_2	32	Not used.	N	N	N	N
act_2	8	Not used.	N	N	N	N
user_3	32	Not used.	N	N	N	N
act_3	8	Not used.	N	N	N	N

Notes![1] The `comserv` field is mandatory for all MOA calls. To use the MDS, an application must open a session explicitly by calling `MOA_open` or implicitly by calling one of the other services. The comserv supplied in the `moa_param_i` structure and a session number, assigned by MOA, identify the session. The comserv used must be previously configured; see the earlier section titled “MOA Configuration” in this chapter for more information. All subsequent calls on that session must include the same comserv name.

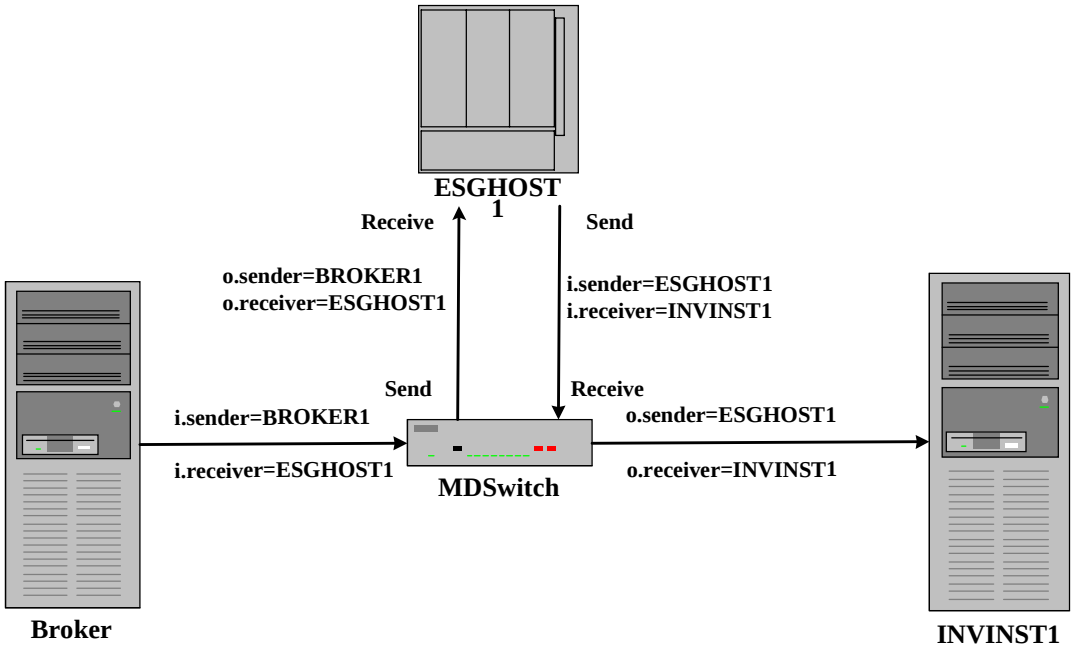
[2] ESG will provide the `sender` code that identifies the message sender. The MDS must know the `sender` code. Note the use of sender and receiver codes in the example below. OASYS clients should put the identity of the message originator into this field. For example, if a broker acts on behalf of corresponding brokers, it should put the initiating broker’s code into this field.

[3] The `receiver` code identifies the host to which the sender sends a message. ESG will provide this value. This field can be up to 37 characters in length.

For example, the broker BROKER1 sends an MT511 *CN New* message that includes, among many others, the following fields:

```
:23A:CN          /sub message type
:23B:01          /sub message type function (“new”)
:80a1:BROKER1    /sender of message
:80a3:INVINST1   /receiver of message
```

[4] The following diagram shows how the message passes through the MDS. Note that *i* represents `moa_param_i` and o represents `moa_param_o`.



The `user_1` field must be set to `OASYS` by all clients that send MT511 formatted messages. Messages that the current OASYS workstations initiate must set the value of that field to `OLD`. If the `user_1` field is set, then the field `act_1` must be set to `INITIAT`.

The default value of the `user_1` field is `OASYS`. If that field contains blanks (spaces), the MDS treats it as if it were `OASYS`.

[5] The `rfk` and the `trn` fields, mandatory fields to be placed within the `MOA_param_i`, are constructed from the following SSAB tags:

SSAB TAG	Size	SSAB Message Field
TFH1	3	Message Type = “511”
23A	2	Sub-message type
23B	2	Sub-message function
TF01	1	Advice Indicator
23M	3	Status/Response code

If any of these fields are missing from the message, the `rfk` field must be padded with blank spaces to accommodate the missing fields.

[6] The `trn` field should be constructed from the following SSAB tags. If field 20B is defined then the `trn` will be composed of the following:

Size	SSAB Message Field	Value and Comments
2	Type of Transaction Reference	`01` (to indicate sender's reference) `03` (to indicate common reference)
16	Transaction Reference Sender's Reference	Variable length, terminated with "." Variable length, terminated with "."
2	Transaction Version Number	20C field (version of trade ≥ `01`.

If field 20B is missing, as in the case of a broker or an institution sending a new trade, then the `trn` field should contain `01` (to indicate sender's reference) followed by the content of field TF14 (party reference).

If any of these fields are missing from the message, the `trn` field must be padded with blank spaces to accommodate the missing fields.

[7] This field must be set to `MOA_act_auth` (defined in **MOA_bas.h**).

[8] For setting and checking this field, use `MOA_pos_dup_y` or `MOA_pos_dup_n`, as defined in **MOA_bas.h**.

Output Message Parameters Structure (moa_param_o)

The application program maintains the `moa_param_o` structure. The application supplies it to the MOA in every service call. The MOA updates this structure, and it contains various parameters retrieved from the MDS headers following execution of the relevant service.

Note! Fields in this structure for which contents are not relevant to the specific call will contain blanks.

The following table lists the fields that comprise the `moa_param_o` structure. Fields that the MDS does not use are gray.

Name	Type	Description
msg_len	Long	On send, actual length of sent message that, in case of successful completion, is same as `msg_len` argument passed to `MOA_send_msg` call. On retrieve, actual length of received message, not including space for a 0 terminator.
comserv	char[8]	MDS ComServ name to which client application connects.
sess_nr	char[4]	Session number assigned to current session of comserv.
seq_nr	char[6]	On send, sequence number assigned to sent message. On retrieve, sequence number of received message.
nak_code	char[8]	On send, error code of received NAK. See Appendix B, "MOA Return/NAK Codes," for a complete list of MOA error and NAK codes. On retrieve, not used.
fail_rsn	Long	Reason that requested service call failed. Not supported in this release.
pde	char[1]	On send, not used. On retrieve, Possible Duplicate Emission. This equals `MOA_POS_DUP_Y` when MDSwitch has sent a message more than once.
mint_dir	char[1]	O
msg_fmt	char[4]	MDS
mt_fmt	char[4]	mdsa
owner	char[8]	Not used.
sender	char[32]	Sender of message as entered by sender in `moa_param_i.sender`.
receiver	char[37]	Receiver of message as entered by sender in `MOA_param_i.receiver`.
rfk	char[16]	Message Reference Key (RFK) within MDS as entered by sender in `moa_param_i_rfk.`
trn	char[21]	Message Transaction Reference Number (TRN) within MDS as entered by sender in `moa_param_i_trn`.
stage	char[8]	`In-Xmt`
next_act	char[8]	Xmt
pdm	char[1]	Possible Duplicate Message. Equals `MOA_pos_dup_y` when MDS flags a message it received as Possible Duplicate or, because of a communications failure, when MDS may have sent the message previously.
int_prio	char[6]	Message internal priority within MDS.
net_prio	char[6]	Network priority within MDS.
rec_prio	char[6]	Not used.

Name	Type	Description
val_date	char[6]	Not used.
cur	char[3]	Not used.
amount	char[15]	Not used.
net_dels	char[8]	Not used.
net_coms	char[8]	Not used.
net_sess	char[4]	Not used.
net_seq	char[6]	Not used.
net_date	char[6]	Not used.
net_time	char[6]	Not used.
net_rslt	char[3]	Not used.
inp_coms	char[8]	Name of ComServ through which message entered MDS. It is value entered by sender in `MOA_PARAM_I.COMSRV`.
inp_sess	char[4]	Session number on which MDS received message from sender.
inp_seq	char[6]	Sequence number of message when MDS received it from sender.
nak_text	char[55]	Not used.
testw	char[16]	Not used.
tst_date	char[6]	Not used.
tst_p	char[16]	Not used.
tst_amnt	char[15]	Not used.
tst_cur	char[3]	Not used.
tst_rslt	char[8]	Not used.
tst_wrap	char[5]	Not used.
tst_seq	char[3]	Not used.
tst_seq2	char[3]	Not used.

4

Messaging Open API (MOA) Services

This section provides detailed specifications for each MOA service call and includes the following information:

1. Service name and description.
2. Specific service parameters to be passed by the application.
3. Mandatory fields in the input parameters structure.
4. Results when the service is successfully executed.
5. Possible return codes and suggested error handling.

MOA_set

Usage:

`MOA_set`

Arguments:

`a-b` — Conversion type, ASCII to binary conversion.

`b-a` — Conversion type, binary to ASCII conversion.

`COMSERV1.TXT` — ASCII file, either input or output file path and name.

`COMSERV1.DAT` — binary file, either input or output file path and name.

Return Values:

On success, the program displays, to the standard output, either:

"a-b conversion ended successfully," or a self-explanatory error message.

"b-a conversion ended successfully," or a self-explanatory error message.

Description:

The `MOA_set` conversion program supports conversion of an ASCII text file to a binary configuration file or conversion of a binary configuration file to an ASCII text file.

The binary configuration file created must be named <COMSERV NAME>.DAT (all upper-case). The ASCII text file may have any name; it is recommended that it be named <COMSERV NAME>.TXT

When running `MOA_set`, the arguments must be supplied as follows:

Argument name	ASCII to Binary Conversion	Binary to ASCII Conversion
Conversion type:	a-b	b-a
Input file path and name:	ASCII file	binary file
Output file path and name:	binary file	ASCII file

Restrictions:

None.

Related Functions:

All of the MOA services use the binary configuration file generated by this program.

References:

None.

Example:

ASCII to binary conversion:

```
MOA_set a-b COMSERV1.TXT COMSERV1.DAT
```

Binary to ASCII conversion:

```
MOA_set b-a COMSERV1.DAT COMSERV1.TXT
```

MOA_open

Usage:

```
MOA_RT_CODE MOA_open(MOA_PARAM_I* moa_param_i,MOA_PARAM_O*
       moa_param_o,
       MOA_WORK_AREA* moa_work_area)
```

Arguments:

`moa_param_i` — a pointer to a structure, allocated by the caller, containing input parameters.

`moa_param_o` — a pointer to a structure, allocated by the caller, containing output parameters.

`moa_work_area` — a pointer to an area, allocated by the caller, for internal use by MOA.

Return Values:

The system returns the following values:

Return Value	Description
MOA_SUCCESS	The service has been successfully executed.
MOA_inv_params	Invalid parameters supplied. Application program error. Debug and re-issue request.
MOA_comm_problem	A communication problem occurred. Sleep for two seconds, then call `MOA_open` again. After the third call to `MOA_open`, check the lines so that the environment will be ready to process when restarted.
MOA_BAD_Seq_num	Open request failed due to incorrect session number. Call `MOA_open` again. After the third call to `MOA_open`, check the configuration file to see why session number went out of synchronization. After investigating, update the configuration file.
MOA_open_error	An unspecified protocol error occurred. Contact your integration specialist.
MOA_no_config_file	A configuration file has not been set up for the ComServ. Make sure that a configuration file was created for the ComServ used in the call. Check that you have spelled the ComServ name correctly.
MOA_bad_config_file	The format of the configuration file is bad. Check the configuration file, correct, and re-issue the request.

Description:

This service opens a session between the client application and the MDSwitch. If the session is already open, the service returns a successful result. MOA uses the `connect_tries` and `reconnect_period` parameters (defined in the configuration file) to execute a number of attempts to make the TCP/IP connection. If the application cannot establish a session, after the indicated number of retries, an appropriate return code is sent back to the application.

Restrictions:

The following field must be filled in `moa_param_i` during input:

Name	Type	Description
comserv	char[8]	The MDS ComServ name with which the client application wishes to establish a session. This ComServ must be configured as described in the earlier section titled "MOA Configuration."

Related Functions:

All of the other MOA services use the connection established by this function.

References:

None

Example:

```
#include <MOA_call.h>
#include <string.h>
MOA_RT_CODE moa_rtc;
MOA_PARAM_I moa_param_i;
MOA_PARAM_O moa_param_o;
MOA_WORK_AREA moa_work_area;
...
/* initialize the moa_param_i structure to all blanks*/
memset(&moa_param_i, ' ', sizeof (MOA_PARAM_I));
/*move the COMSERV name into the moa_param_i structure*/
memcpy(&moa_param_i.comserv, "COMSERV1",
sizeof(MOA_param_i.comserv));
...
/*call MOA_open*/
MOA_rtc = moa_open(&moa_param_i, &moa_param_o, &moa_work_area);
/*check the return code*/
if (moa_rtc != MOA_SUCCESS)
{
 error handling
}
```

MOA_close

Usage:

```
MOA_RT_CODE MOA_close(MOA_PARAM_I* moa_param_i,MOA_PARAM_O*
      moa_param_o, MOA_WORK_AREA* moa_work_area)
```

Arguments:

`moa_param_i` - a pointer to a structure, allocated by the caller, containing input parameters.

`moa_param_o` - a pointer to a structure, allocated by the caller, containing output parameters.

`moa_work_area` - a pointer to an area, allocated by the caller, for internal use by MOA.

Return Values:

The system returns the following values:

Return Value	Description
MOA_SUCCESS	The service has been successfully executed.
MOA_inv_params	Invalid parameters supplied. Application program error. Debug and re-issue request.
MOA_comm_problem	A communication problem has occurred. Check the lines so that the environment will be ready to process when restarted.
MOA_open_error	Open request failed due to incorrect session number. Check the configuration file to see why session number went out of synchronization. After investigating, update the configuration file.
MOA_no_config_file	A configuration file has not been set up for the ComServ. Make sure that a configuration file was created for the ComServ used in the call. Check that you have spelled the ComServ name correctly.
MOA_bad_config_file	The format of the configuration file is bad. Check the configuration file, correct, and re-issue the request.

The session is over and no further communication may take place until another session opens. The following fields are updated in `moa_param_o`:

Name	Type	Description
comserv	char[8]	The MDS ComServ name with which the client application communicated.
sess_nr	char[4]	The session number of the recently closed session.

Description:

This service closes an active session between the client application/MOA and the MDSwitch. If the session already ended, this service returns a successful result.

Restrictions:

The following field must be filled in `MOA_param_i` during input:

Name	Type	Description
comserv	char[8]	The MDS ComServ name with which the client application is in session. This is the name used to open the session. See the description of `MOA_open`.

Related Functions:

All of the other MOA services use the connection terminated by this function call.

References:

None

Example:

```
#include <MOA_call.h>
#include <string.h>
MOA_RT_CODE moa_rtc;
MOA_PARAM_I moa_param_i;
MOA_PARAM_O moa_param_o;
MOA_WORK_AREA moa_work_area;
...
/* initialize the moa_param_i structure to all blanks*/
memset(&moa_param_i, ' ', sizeof(MOA_PARAM_I));
/*move the COMSERV name into the MOA_param_i structure*/
memcpy(&moa_param_i.comserv, "COMSERV1",
sizeof(moa_param_i.comserv));
...
MOA_rtc = MOA_close(&moa_param_i, &moa_param_o, &moa_work_area);
if (moa_rtc != MOA_SUCCESS)
{
 error handling
}
else
{
 session closed successfully
}
```

MOA_send_msg

Usage:

```
MOA_RT_CODE MOA_send_msg(char* moa_msg, MOA_MSG_LEN moa_msg_len,
      MOA_PARAM_I* moa_param_i, MOA_PARAM_O* moa_param_o, MOA_KEY*
      moa_key,
      MOA_WORK_AREA* moa_work_area)
```

Arguments:

`moa_msg` - a pointer to an area containing the message to be transmitted `MOA_msg_len` - length of the message to be transmitted, not including any 0 terminator.

`moa_param_i` - a pointer to a structure, allocated by the caller, containing input parameters.

`moa_param_o` - a pointer to a structure, allocated by the caller, containing output parameters.

`moa_key` - a pointer to an area containing the authentication key.

`moa_work_area` - a pointer to an area, allocated by the caller, for internal use by MOA.

Return Values:

The system returns the following values:

Return Value	Description
MOA_SUCCESS	The service successfully executed. Session number and sequence number of the sent message are stored in `moa_param_o`. Mark the message as successfully transmitted and remove it from the queue.
MOA_inv_params	Invalid parameters supplied. Application program error. Debug and re-issue request.
MOA_open_error	Open session failed.
MOA_msg_too_long	The messages exceeds the maximum allowed message length (32000 characters). `msg_len` is greater than the 32000 limit. The MOA has not sent the message.
MOA_msg_NAK	The message has been NAK'd.
MOA_comm_problem	A communication problem has occurred. Check the lines so that the environment will be ready to process when restarted.
MOA_open_error	Open request failed due to incorrect session number. Check the configuration file to see why session number went out of synchronization. After investigating, update the configuration file.
MOA_no_config_file	A configuration file has not been set up for the ComServ. Make sure that a configuration file was created for the ComServ used in the call. Also, check that you have spelled the ComServ name correctly.
MOA_bad_config_file	The format of the configuration file is bad. Check the configuration file, correct, and re-issue the request.

Description:

This service requests transmission of a message, on the session with the specified ComServ, to the MDSwitch. If there is no active opened session when calling this service, the service will attempt to open it before sending the message.

After the message is transmitted to the MDSwitch, a positive or negative acknowledgment (ACK or NAK) is returned. A positive acknowledgment from the MDSwitch indicates that it safe-stored the message and put it on the destination queue. The MDSwitch will deliver the

message to the destination when that destination establishes a session with the MDSwitch and is ready to accept the message. A negative acknowledgment indicates that the MDSwitch could not process the message and did not place it into the destination queue.

The following fields are updated in `moa_param_o`:

Name	Type	Description
`msg_len`	Long	The actual length of the sent message that, in case of successful completion, will be the same as the `moa_msg_len` parameters provided in the call.
`comserv`	char[8]	The MDS ComServ name with which the client application has a session.
`sess_nr`	char[4]	The current session number.
`seq_nr`	char[6]	The sequence number assigned to the sent message.
`nak_code`	char[8]	The error code when the application receives a NAK from the MDSwitch.

Restrictions:

`moa_param_i` must have the information as described in the earlier section in this chapter titled “Input Message Parameters Structure (moa_param_i).” The application must supply all mandatory fields.

Related Functions:

`MOA_open`

`MOA_rtrv_no_ack`

References:

None

Example:

```
#include <MOA_call.h>
#include <string.h>
MOA_RT_CODE moa_rtc;
MOA_PARAM_I moa_param_i;
MOA_PARAM_O moa_param_o;
MOA_WORK_AREA moa_work_area;
MOA_KEY moa_key;
MOA_MSG_LEN moa_msg_len;
char moa_msg[MOA_MAX_BUFF_LEN];
...
/* Initialize the MOA_param_i structure to all blanks*/
memset(&moa_param_i, ' ', sizeof(MOA_PARAM_I));
/*Move the COMSERV name into the MOA_param_i structure*/
memcpy(&moa_param_i.comserv, "COMSERV1",
sizeof(moa_param_i.comserv));
/* Store the message format type in the MOA_param_i structure. The
message format for OASYS messages is "MDS " */
memcpy(&moa_param_i.msg_fmt, MOA_FMT_MDS,
sizeof(moa_param_i.msg_fmt));
...
/* Prepare message in moa_message and its length in moa_msg_len */
...
moa_rtc = MOA_send_msg(moa_msg,
moa_msg_len,&moa_param_i,&moa_param_o,
                        &moa_key, &moa_work_area);
```


```
if (moa_rtc != MOA_SUCCESS)
{
    error handling
}
else
{
  ... message was successfully sent, continue processing
}
```

MOA_rtrv_no_ack

Usage:

```
MOA_RT_CODE MOA_rtrv_no_ack(char* moa_msg, MOA_msg_LEN moa_msg_len,
       MOA_PARAM_I* moa_param_i, MOA_PARAM_O* moa_param_o,
       MOA_key* moa_key,MOA_WORK_AREA* moa_work_area)
```

Arguments:

`moa_msg` - a pointer to an area, allocated by the caller, into which the retrieved message will be copied.

`moa_msg_len` - the length of the message area into which the message will be placed.

`moa_param_i` - a pointer to a structure, allocated by the caller, containing input parameters.

`moa_param_o` - a pointer to a structure, allocated by the caller, containing output parameters.

`moa_key` - a pointer to an area containing the authentication key.

`moa_work_area` - a pointer to an area, allocated by the caller, for internal use by MOA.

Return Values:

The system returns the following values:

Return Value	Description
MOA_SUCCESS	The service has been successfully executed. A message was received from the MDSwitch and is stored in moa_msg buffer with the associated information stored in the `moa_param_o` structure. Process the message and/or safe-store it, and when done call `MOA_send_ack`.
MOA_inv_params	Invalid parameters supplied. Application program error. Debug and re-issue request.
MOA_open_error	Open session failed.[1]
MOA_no_msg	Your application did not retrieve a message. No message is available at this time. The queue is empty. Wait for some time and re-issue the request.[2]
MOA_msg_NAK	Received message NAK'd by MOA. The message had a bad sequence number. Close the session. Check sequence numbers and open a new session.
MOA_msg_too_long	The message retrieved exceeds the maximum allowed message length. Increase the buffer in which the retrieved message is stored. Try the operation again.
MOA_comm_problem	A communication problem has occurred.[1]
MOA_no_config_file	A configuration file has not been set up for the ComServ. Make sure that a configuration file was created for the ComServ used in the call. Fix it and retry the operation.
MOA_bad_config_file	The format of the configuration file is bad. Check the configuration file, correct, and re-issue the request.

Notes! [1]There is either a communication problem or the MDSwitch is not responding. Check the connection and try to open a session. If the communication path is correct, then call ESG to check the interface. A protocol violation may reflect as this error. An example of this is when a second `MOA_rtrv_no_ack` goes out without an `MOA_send_ack` between it and the first one.

[2] The time that an application should wait between retries is application dependent. If your application issues requests too often, these requests may use too many system resources. The recommended value is 1 to 4 seconds.

If the MDSwitch did not receive an ACK for the last retrieved message (`MOA_send_ack`), the message will be re-sent to the application on the next established session. The MDSwitch marks the message as "Possible Duplicate." The application must be able to handle such "Possible Duplicate" messages.

Description:

This service requests retrieval of a message from the specified ComServ's queue on the MDSwitch. If the communication link is not yet established when calling this service, the service will attempt to open it before sending the request. This service checks the received message. If the received sequence number does not match the one expected by the MOA, or the MOA encounters formatting errors, it sends a NAK back to the MDSwitch and notifies the application with the appropriate return code. If, however, the sequence number matches and MOA mapped the incoming headers into the `moa_param_o` structure, then MOA returns to the application with the message. After the application processes and/or safe-stores the message, it must call `MOA_send_ack`. If the application cannot accept the message, it must close the session.

If you receive a message from the MDS, or the received message is copied to the `moa_msg` area, the `moa_param_o` structure will contain the information described in the earlier section in this chapter titled "Output Message Parameters Structure (moa_param_o)."

Restrictions:

The following field must be filled in `moa_param_i` during initial input:

Name	Type	Description
comserv	char[8]	The MDS ComServ name with which the client application wishes to establish a session. This ComServ must be configured as described in the earlier section in this chapter titled "MOA Configuration."

Related Functions:

`MOA_open`

`MOA_send`

`MOA_send_ack`

References:

None

Example:

```
#include <MOA_call.h>
#include <string.h>
MOA_RT_CODE moa_rtc;
MOA_PARAM_I moa_param_i;
MOA_PARAM_O moa_param_o;
MOA_WORK_AREA moa_work_area;
MOA_KEY moa_key;
MOA_MSG_LEN moa_msg_len;
char moa_msg[MOA_MAX_BUFF_LEN];
...
```


```
/* Initialize the moa_param_i structure to all blanks*/
memset(&moa_param_i, ' ', sizeof(MOA_PARAM_I));
/* Move the COMSERV name into the MOA_param_i structure */
memcpy(&moa_param_i.comserv, "COMSERV1",
sizeof(moa_param_i.comserv));
moa_msg_len = MOA_MAX_BUFF_LEN;
moa_rtc = MOA_rtrv_no_ack(moa_msg,
moa_msg_len,&moa_param_i,&moa_param_o,
   &moa_key, &moa_work_area);
if (moa_rtc != MOA_SUCCESS)
{
 error handling 1
}
else
{
if (application found some error, or the application is not ready
to process the message now, e.g., there is no disk space available)
then    {
moa_rtc = MOA_close(&moa_param_i, &moa_param_o, &moa_work_area);
if (moa_rtc != MOA_SUCCESS)
{
 error handling
}
else
{
 session closed successfully
}
}
else  /* no error was found by application */
{
/* Application should have safe-stored message by this point */
moa_rtc =
MOA_send_ack(&moa_param_i,&moa_param_o,&moa_key,&moa_work_area);
if (moa_rtc != MOA_SUCCESS)
{
 error handling 2
}
}
.
```

MOA_send_ack

Usage:

```
MOA_RT_CODE MOA_send_ack(MOA_PARAM_I* moa_param_i,MOA_PARAM_O*
       moa_param_o,
       MOA_KEY* moa_key,MOA_WORK_AREA* moa_work_area)
```

Arguments:

`moa_param_i` - a pointer to a structure, allocated by the caller, containing input parameters.

`moa_param_o` - a pointer to a structure, allocated by the caller, containing output parameters.

`moa_key` - a pointer to an area containing the authentication key.

`moa_work_area` - a pointer to an area, allocated by the caller, for internal use by MOA.

Return Values:

The system returns the following values:

Return Value	Description
MOA_SUCCESS	The service has successfully executed.
MOA_no_memory	Not enough memory for acknowledgement message.
MOA_comm_problem	A communication problem has occurred. Check communication path. Fix problem and open another session.
MOA_no_config_file	A configuration file has not been set up for the ComServ. Make sure that a configuration file was created for the ComServ used in the call. Also, check that you have spelled the ComServ name correctly.
MOA_bad_config_file	The format of the configuration file is bad. Check the configuration file, correct, and re-issue the request.

Note! If the MDSwitch does not receive the ACK for the last retrieved message, the message will be re-sent to the application on the next established session. The MDSwitch marks the message as “Possible Duplicate.” The application must be able to handle such “Possible Duplicate” messages.

Description:

This service sends a positive acknowledgment (ACK) message to the MDSwitch, referring to the last received message. Your application should only call this service after a successful call to `MOA_rtrv_no_ack`.

When the ACK message is successfully passed to the MDS, the following fields are updated in `moa_param_o`.

Name	Type	Description
comserv	char[8]	The MDS ComServ name with which the client application communicated.
sess_nr	char[4]	The session number assigned to the current session.

Restrictions:

The following field must be filled in `moa_param_i` during initial input:

Name	Type	Description
`comserv`	char[8]	The MDS ComServ name with whom the client application wishes to establish a session. This ComServ must be configured as described in the earlier section in this chapter titled "MOA Configuration."

Related Functions:

`MOA_rtrv_no_ack`

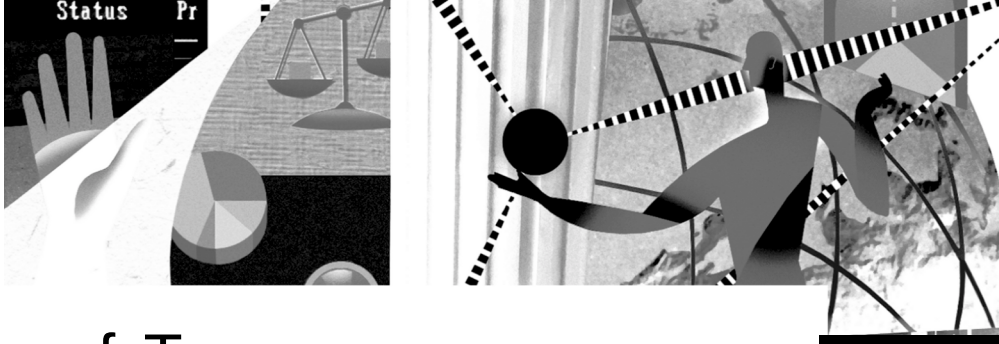
References:

None

Example:

```
#include <MOA_call.h>
#include <string.h>
MOA_RT_CODE moa_rtc;
MOA_PARAM_I moa_param_i;
MOA_PARAM_O moa_param_o;
MOA_WORK_AREA moa_work_area;
MOA_KEY moa_key;
MOA_MSG_LEN moa_msg_len;
char moa_msg[MOA_MAX_BUFF_LEN];
...
/* Initialize the moa_param_i structure to all blanks*/
memset(&moa_param_i, ' ', sizeof(MOA_PARAM_I));
/* Move the COMSERV name into the MOA_param_i structure */
memcpy(&moa_param_i.comserv, "COMSERV1",
sizeof(moa_param_i.comserv));
moa_msg_len = MOA_MAX_BUFF_LEN;
moa_rtc = MOA_rtrv_no_ack(moa_msg,
moa_msg_len,&moa_param_i,&moa_param_o,
   &moa_key, &moa_work_area);
if (moa_rtc != MOA_SUCCESS)
{
 error handling 1
}
else
{
if (application found some error, or the application is not ready
to process the message now, e.g., there is no disk space available)
then    {
moa_rtc = MOA_close(&moa_param_i, &moa_param_o, &moa_work_area);
if (moa_rtc != MOA_SUCCESS)
{
 error handling
}
else
{
 session closed successfully
}
}
```

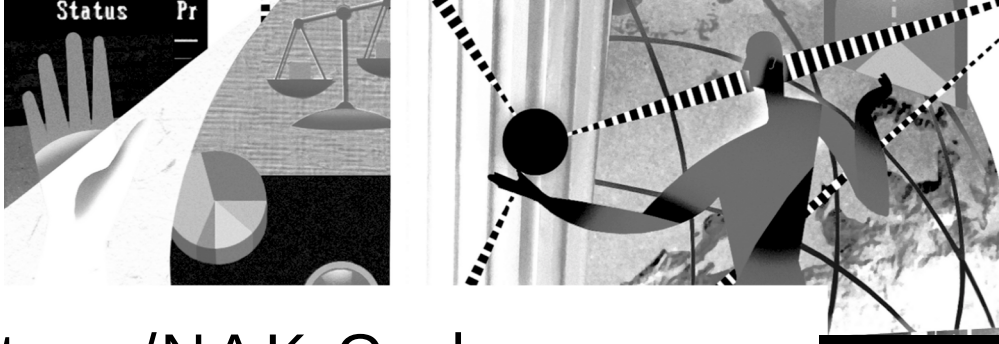
```
else  /* no error was found by application */
{
/* Application should have safe-stored message by this point */
moa_rtc =
MOA_send_ack(&moa_param_i,&moa_param_o,&moa_key,&moa_work_area);
if (moa_rtc != MOA_SUCCESS)
{
 error handling 2
}
}
.
```



A: Glossary of Terms

This appendix defines the technical terms used in this document.

Address	A client identifier.
Alias	An alternate name for an address.
Client	Any system that requires the services of the MDS. Clients send messages to other clients. Clients receive messages from other clients. Each client is known by a unique address.
Client Address	An address assigned to a client.
Communication Link	Synonym of “connection.”
ComServ	Communication Service. A client application can communicate with one or more ComServs. Each relation of a client program to a ComServ is called a stream. Sessions are established on streams.
Logical Stream	Messages received on a single queue with an identical Processing Stream Identifier define one logical stream. Messages of one logical stream are delivered to the destination in the order they were received by the MDS.
MDS	Message Delivery System. MDS is composed of the MDSwitch, the MDS/API, and the underlying transport network. See the illustration under “Scope of the Message Delivery System” in Chapter 1, “Message Delivery System Overview,” for a view of the MDS boundary.
MDSwitch	Message Delivery Switch. The part of the MDS which provides for message storage and queue management. See the illustration under “Scope of the Message Delivery System” in Chapter 1, “Message Delivery System Overview,” for more information.
MOA	Messaging Open API to the MDSwitch. MOA is a component of the MDS. See the illustration under “Scope of the Message Delivery System” in Chapter 1, “Message Delivery System Overview,” for more information.
Network	A physical medium, such as an ethernet LAN, a leased telephone line, a dial-up telephone connection, or an X.25 virtual circuit, which enables communication between a client and the MDSwitch.
Processing Stream Identifier	Information, associated with a message, which defines the order in which messages received by the MDSwitch on a single session are to be delivered to their destination. Messages with the same Processing Stream Identifier are delivered to the destination one at a time, in the order they were sent into the MDSwitch.
Queue	A queue is a facility to store messages in a given order until they are retrieved. A queue is associated with a destination. A message is addressed to a destination, and until that destination retrieves the message, it is stored on the queue associated with the destination. A message is put on the queue following all previously stored messages of the same priority. Messages are ordered by priority and, within priority, by order received.
Session	A session is an application-to-MDSwitch logical connection which enables message exchange. A session is initiated by a request to open it. At that time, the client application identifies itself to the MDSwitch and the session sequence number and message sequence number are synchronized. The session sequence number is per stream, where a stream identifies the client address and the MDS ComServ.
Stream	A logical communications connection over which a client sends messages to the switch for delivery to other clients and receives messages sent to it by other clients. A stream is defined by the client application-to-MDSwitch ComServ connection.



B: MOA Return/NAK Codes

This chapter includes information about MOA error codes and NAK codes. It contains the following sections:

B

MOA Return Codes

When returning to the application program, MOA provides a return code. If the required operation was executed successfully, the MOA returns `0` (`MOA_SUCCESS`). If an error occurred, one of the other return codes will indicate the type of error that occurred.

Note! In some cases, the return code is supplemented with a `FAILURE_REASON` field. This field supplies the specific reason regarding the error condition that occurred.

The following table lists the MOA return codes in alphabetical order:

Return Code	Number
MOA_APPL_ERROR	13
MOA_BAD_CONFIG_FILE	8
MOA_BAD_MSG_FORMAT	10
MOA_BAD_SEQ_NUM	9
MOA_COMM_PROBLEM	5
MOA_INV_PARAMS	1
MOA_MAC_FAIL	15
MOA_MAC_REDUNDANT	16
MOA_MSG_NAK	2
MOA_MSG_TOO_LONG	4
MOA_NO_CONFIG_FILE	7
MOA_NO_MAC	14
MOA_NO_MEMORY	6
MOA_NO_MSG	3
MOA_OPEN_ERROR	11
MOA_RECV_TIMEOUT	12
MOA_REPLY_TOO_LONG	17
MOA_SUCCESS	0

During MOA operations, MOA returns the return code number. The MOA return codes are listed on the following pages in numerical order, along with an explanation of the return code, possible supplemental failure reasons (where relevant), and action to be taken (where relevant).

B

Return Code 0 (MOA_SUCCESS)

The service has been successfully executed.

Return Code 1 (MOA_INV_PARAMS)

An invalid parameter was supplied to the service accessed, or there is a problem in the configuration file. The failure reason indicates which parameter is invalid.

Supplemental Failure Reasons

Reason Name	Number	Explanation	Action Needed
MOA_COMSERV	1	Invalid ComServ name was supplied in a non-first MOA service call.	Check the application program for possible corruption of the `moa_work_area`. Check that the correct ComServ name was supplied in the input parameters.
MOA_SENDER	4	Mandatory sender field is not present in input parameters structure.	Correct the application program supplying this parameter.
MOA_RECEIVER	5	Mandatory receiver field is not present in input parameters structure.	Correct the application program supplying this parameter.
MOA_NET_COMSERV	17	Mandatory network Comserv field is not present in the input parameters structure (SIC/SECOM only)	Correct the application program supplying this parameter.
MOA_NET_MT	18	Mandatory message type field is not present in input parameters structure.	Correct the application program supplying this parameter.
MOA_USER_ACT	21	When using the user/activity option, a user is supplied without a corresponding activity.	Correct the application program supplying the user/ activity pair.
MOA_INV_AUTHEN_KEY	22	An invalid authentication key was supplied	Correct the application program supplying this parameter.
MOA_BIN_AND_AUTH_INVALID	23	Forbidden combination in the configuration file: `authen_activated = y` and `binary_data = y`	Change the parameter in the configuration file.

Return Code 2 (MOA_MSG_NAK)

The message has been NAKed (`MOA_send_msg` or `MOA_send_stp_msg` call) or the retrieved message was NAKed by MOA due to incorrect sequence number or authentication error (`MOA_retrieve_msg` or `MOA_retrieve_long_msg` or `MOA_rtrv_no_ack` or `MOA_rtrv_long_no_ack` call). The NAK code explains the reason for the error (see the section titled “NAK Codes” for more information.

Return Code 3 (MOA_NO_MSG)

No message was retrieved.

Return Code 4 (MOA_MSG_TOO_LONG)

The retrieved message exceeds the length of the message text area. (`MOA_retrieve_msg` or `MOA_rtrv_no_ack` call).

Return Code 5 (MOA_COMM_PROBLEM)

An error occurred when executing a communication related function. Check communication (TCP/IP) definition. Check application program for possible corruption of `moa_work_area`. Check that the communication link, in MINT, is open.

Supplemental Failure Reasons

Reason Name	Number	Explanation
MOA_SOCKET_ALLOC_ERROR	1	Error occurred while allocating a socket in opening the communication link.
MOA_SOCKET_CONNECT_ERROR	2	Error occurred when connecting socket to port when opening the communication link.
MOA_SOCKET_SHUTDOWN_ERROR	3	Error occurred when executing socket shutdown.
MOA_SOCKET_CLOSE_ERROR	4	Error occurred when executing socket net_close.
MOA_LOGIN_SEND_ERROR	5	Error occurred when sending a login message to MINT.
MOA_LOGOUT_SEND_ERROR	6	Error occurred when sending a logout message to MINT.
MOA_MSG_SEND_ERROR	7	Error occurred when sending a message to MINT.
MOA_REQUEST_SEND_ERROR	8	Error occurred when sending a retrieve request to MINT.
MOA_ACK_SEND_ERROR	9	Error occurred when sending a message acknowledgement to MINT.
MOA_LOGIN_RECEIVE_ERROR	10	Error occurred when attempting to receive login response from MINT.
MOA_LOGOUT_RECEIVE_ERROR	11	Error occurred when attempting to receive logout response from MINT.
MOA_ACK_RECEIVE_ERROR	12	Error occurred when attempting to receive a message acknowledgement from MINT.
MOA_RTRV_RECEIVE_ERROR	13	Error occurred when attempting to receive a message from MINT.
MOA_INVALID_DATA	20	Invalid data is present in a response received from MINT.

Return Code 6 (MOA_NO_MEMORY)

Not enough memory for allocations.

Return Code 7 (MOA_NO_CONFIG_FILE)

A configuration file has not been setup for the ComServ.

Return Code 8 (MOA_BAD_CONFIG_FILE)

An error occurred when accessing the ComServ's configuration file. The format of the configuration file is erroneous.

Supplemental Failure Reasons

Reason Name	Number	Explanation	Action Needed
MOA_READ_ERROR	1	Read error occurred when attempting to read the configuration file.	Check the ComServ's configuration file for data corruption. If the file is corrupted, activate the `MOA_set` program (with the a-b option) taking care to update the session and sequence numbers with the correct values (taken from MINT).
MOA_WRITE_ERROR	2	Write error occurred when attempting to update the configuration file.	Check your application program for possible corruption of the `moa_work_area`. If this is the case, take actions described above.

Return Code 9 (MOA_BAD_SEQ_NUM)

A retrieved message’s sequence number was not as expected.

Return Code 10 (MOA_BAD_MSG_FORMAT)

A retrieved message contained a format error.

Supplemental Failure Reasons

Reason Name	Number	Explanation	Action Needed
MOA_INVALID_SEP	1	Invalid line separator is received in header of a retrieved message.	Check application program for possible of `moa_work_area`. Check that the separator parameter in the ComServ’s configuration file is the same as that defined in MINT.

Return Code 11 (MOA_OPEN_ERROR)

Negative response for a login request was received from MINT, due to incorrect session number or unknown status in the login response was received from MINT.

B

Return Code 12 (MOA_RECV_TIMEOUT)

A response was not received from MINT within the timeout period stipulated in the ComServ's configuration file (`MOA_open` or `MOA_close` call), or an ACK/NAK message was not received within the timeout period (`MOA_send_msg` call), or a message was not retrieved within the timeout period (`MOA_retrieve_msg` or `MOA_retrieve_long_msg` or `MOA_rtrv_no_ack` or `MOA_rtrv_long_no_ack` call).

Check the MINT log for reporting of possible errors. Check application program for possible corruption of `moa_work_area`. Check communication (TCP/IP) definition. Check that the communication link, in MINT, is open.

Supplemental Failure Reasons

Reason Name	Number	Explanation
MOA_LOGIN_TIMEOUT	1	Login response not received from MINT.
MOA_LOGOUT_TIMEOUT	2	Logout response not received from MINT.
MOA_ACK_TIMEOUT	3	Acknowledgement message (for a sent message) not received from MINT.
MOA_RTRV_TIMEOUT	4	Message not received from MINT in response to a retrieve request.

Return Code 13 (MOA_APPL_ERROR)

The receiving application could not store or handle the received message, and therefore called the service `MOA_send_nak`.

Return Code 14 (MOA_NO_MAC)

Used internally by MOA routines. This return code is not returned to the calling application.

Return Code 15 (MOA_MAC_FAIL)

Used internally by MOA routines. This return code is not returned to the calling application.

Return Code 16 (MOA_MAC_REDUNDANT)

Used internally by MOA routines. This return code is not returned to the calling application.

Return Code 17 (MOA_REPLY_TOO_LONG)

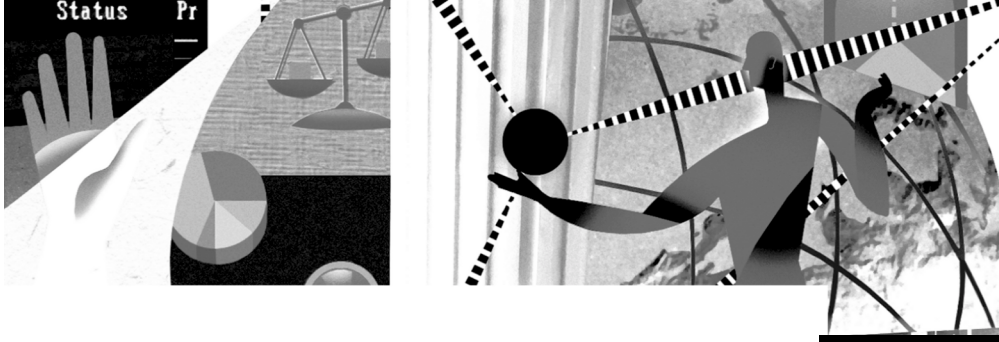
The reply message exceeds the maximum allowed message length [32000 characters] (`MOA_stp_send_reply` call) or the reply message exceeds the length of the reply message text area (`MOA_stp_send_msg` call).

NAK Codes

B

The following is a list of NAK codes which may be returned by MOA. These codes are returned as strings and so must be compared with `strcmp`.

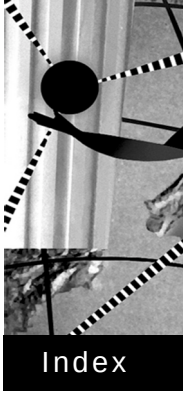
NAK Code	Meaning
MOA_NAK_APPL	NAK sent by the application due to error detected in retrieved messages.
MOA_NAK_SEQUENCE	Bad sequence number. Sequencing error.
MOA_NAK_FORMAT	Incorrect message format.
MOA_NAK_NO_RCVR	No receiver was specified when mandatory for a format type.
MOA_NAK_SRVR	Straight Through Processing: Destination NAKed query message.
MOA_NAK_NO_SRVR	Straight Through Processing: Destination not available to receive query message.
MOA_NAK_INV_RCVR	Receiver unknown.
MOA_NAK_NO_ROUTE	The receiver cannot be reached or is disabled. Cannot determine routing for message.
MOA_NAK_MAC_FAIL	Authentication error.
MOA_NAK_NO_MSG	No message present in call. (i.e., message length was 0 or message pointer was NULL.)
MOA_NAK_NO_MAC	MAC not present when expected.
MOA_NAK_MAC_REDUNDANT	MAC present when not expected.
MOA_NAK_INV_RQST	Invalid request. Straight Through Processing functionality is not allowed.



Index

K

M



N

O

P

R

S

T

U

V