MERVA ESA Components

MERVA USE & Branch for Windows NT Application Programming

Version 4 Release 1

SH12-6336-02

MERVA ESA Components

MERVA USE & Branch for Windows NT Application Programming

Version 4 Release 1

SH12-6336-02

Note!

Before using this information and the product it supports, be sure to read the general information under "Appendix C. Notices" on page 151.

Third Edition, May 2001

This edition applies to

Version 4 Release 1 of IBM MERVA ESA Components (5648-B30)

and to all subsequent releases and modifications until otherwise indicated in new editions.

Changes to this edition are marked with a vertical bar.

Contents

About This Book

This book describes the Application Programming Interface (API) of the IBM licensed program Message Entry and Routing with Interfaces to Various Applications USE & Branch for Windows NT (hereafter abbreviated to MERVA or MERVA Workstation if it is necessary to be more specific).

Who Should Read This Book

This book is written for application programmers who need to access the services provided by the MERVA base component of MERVA. It contains examples of coding methods. It is assumed that you are familiar with the C or REXX programming language, and with the following MERVA services:

- Queuing and routing of messages in MERVA
- Allowing users to access API programs in MERVA

For a detailed description of these services, refer to the *MERVA USE & Branch for Windows NT User's Guide*.

How This Book Is Organized

"Chapter 1. Introduction to the API of MERVA" on page 1 contains introductory information about application programming and MERVA. It explains the API concepts and gives you an overview of the functions.

"Chapter 2. MERVA API Data Types" on page 7 describes to you the symbol definitions and data types that let you use the interface.

"Chapter 3. MERVA API Function Calls" on page 19 contains a detailed description of each function call. It describes the purpose of the function, processing conditions, prerequisites and syntax to use the function, and the function parameters.

"Chapter 4. How to Use, Build, and Load an API Program" on page 121 contains tips and techniques for the use of the MERVA API.

"Chapter 5. The REXX Function Package" on page 127 contains information on how to use the REXX programming language to write application programs with the REXX function package.

"Chapter 6. The SWIFT Link API" on page 131 contains a detailed description of the SWIFT Link API functions.

The appendixes describe return codes and message header checking.

Chapter 1. Introduction to the API of MERVA

With the MERVA API, you can write programs for additional message processing. You can, for example:

- Load messages in and unload messages from message queues
- Gather statistics on messages
- Edit messages
- Check the contents of messages

MERVA supports the following predefined message types:

- SWIFT messages
- Telex messages

For more information about these message types, refer to the *MERVA USE & Branch for Windows NT User's Guide*.

MERVA also supports the processing of user-defined message types. You can use MERVA as a queuing and routing system for general messages that are entered and removed by using the API. Note that the API and MERVA do not check these types of messages.

MERVA Instances

After you install the MERVA program code to your system, you have to configure MERVA by creating a MERVA instance. Note that MERVA can run only one instance at the same time.

The API program needs a running MERVA instance in multi-user mode and the environment variable **ENMD_IPC_DIR**.

For detailed information refer to "Chapter 4. How to Use, Build, and Load an API Program" on page 121 or to the *MERVA USE & Branch for Windows NT Installation and Customization Guide*.

The MERVA Message Routing Concept

Messages within a MERVA instance are stored in a message database. A message is stored in the database when you save a newly created message, add a message with the API, or when an incoming message is received from the SWIFT network or via MERVA Link. Within the message database, messages are assigned to logical entities called message queues.

In general, message queues store messages on a first-in, first-out basis until they are processed. The messages are then routed to the next message queue. The queue to which the message is forwarded is determined by the routing table for your installation. Each message queue belongs to a purpose group. A purpose group is a logical collection of one or more message queues.

Individual MERVA functions, such as Create SWIFT System Messages or Automatic Message Print, are associated with a purpose group and are responsible for processing all messages held in the queues of that purpose group.

The advantage of using purpose groups is that the system administrator can define routing conditions by using the MERVA Customization program. A routing condition tests the contents of message fields, for example, the amount field. It also routes the messages accordingly, possibly to different queues within the same purpose group. For more information about message routing, refer to the *MERVA USE & Branch for Windows NT User's Guide*.

MERVA includes a default set of queues and routing definitions. For information on how to customize MERVA, refer to the *MERVA USE & Branch for Windows NT Installation and Customization Guide*.

API Queues and Their Routing

API applications can only access queues that belong to the API purpose group. Therefore, messages in queues that belong to other purpose groups must be routed to queues in the API purpose group to be processed by an application program.

With this method, you can define the set of messages that can be accessed by an external application. You can also limit the changes that an application is allowed to make to a message. For example, if an application changes a part of the message that it is not allowed to be changed, the routing checks the field and routes the message to a queue for investigation. If the message is correct, it can then be routed to the queue of any purpose group in the MERVA instance.

With the MERVA Customization Program, you can define your own API queues and their corresponding routing. The following figure shows an API routing example that allows an API program to load and unload messages to and from the SWIFT ready-to-send queues.

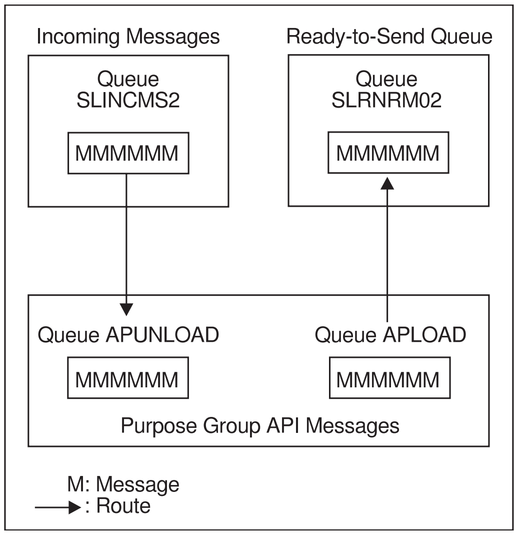

Figure 1. API Routing Example

The messages are routed from the APLOAD queue to a SWIFT ready-to-send queue, for example, the SLRNRM02 queue as shown in Figure 1. Incoming messages are routed from the SLINCMS2 queue to the APUNLOAD queue in this example.

API Concepts

The following section describes the API concepts. It also informs you about tracing, triggering, and message header checking.

The API Message Space

You can access a message in an API queue only via a call to an interface function. The interface allocates sufficient space to store a message and information about the message.It then allows the application access to that space. The API only supports one message at a time. If you want to work with more than one message, you must store these messages in your own program space.

The maximum size of a message is 28000 bytes.

Concurrent Access

You can use more than one API program at a time. Message processing, for example, can be shared by API programs. A program takes the message from a queue and routes it to its next queue. The next program takes it from this queue, processes it, and sends it to a third queue for further processing.

The routing procedure described in Figure 1 on page 2 could also be divided into two programs. One program handles load processing, the other handles unload processing.

Restricting Access to API Programs (API Queues)

Because MERVA works in an environment in which programs compete for resources, it needs to control its own resources. Its own resources are the message queues. An application program must therefore identify itself to MERVA before it is allowed to access the system's queues. The user must also be authorized to execute API programs. Additionally, queues must be assigned to the API access right. Otherwise, no queue and therefore no message can be accessed.

To identify an application program, use the function call **ENMSetAppl**. Otherwise, the program name is used as the default value.

To authorize a user to execute API programs, you must assign and approve the **API - with password** or **API - without password** access right to the user. To do this:

1. Select **Administration** and **Users** from the MERVA menu.
2. Select **Update User Rights** from the **Selected** menu.
3. Select **API - with password** or **API - without password** from the list of **Current Access Rights**.

To assign queues to the API access right, select the Users program. In this program:

1. Select **Update User Rights** from the **Selected** menu.
2. Select the needed access right from the list of **Current Access Rights**.
3. Select **Details**.
4. Select one or more queues.

Audit and Trace Information

The API trace-handling function **ENMTrace()** allows an application to trace its calls to the API. It also allows to add its own information to the API trace file of MERVA.

When the trace is set to **ON** by using the **ENMTrace()** function, the API writes an entry for each function call to that file. Each entry contains the name of the application, the name of the function called, and the contents of the parameters passed to the function as shown in the following example.

```
[    7]*232   19990101  12:00:04        174 ENMCAPI           12      12
ENMAdd                                             enmcapi.c     1234
Queue: APLOAD
```

If the trace is set to **ON**, an application can use the function **ENMWriteTrace()** to write its own entries to the API trace file.Each entry consists of the name of the application, the function name, and the information to be added as shown in the following example.

```
[    9]*232   19990101  12:00:04        174 ENMCAPI           12      12
ENMWriteTrace                                         enmcapi.c     1234
  This is an information message.
```

In addition, each call to a function that affects messages in queues, such as **ENMAdd()**, **ENMPut()**, **ENMRoutePut()**, and **ENMDelete()** results in an entry in the User Audit Log file.

For further information about the diagnosis log file, refer to the *MERVA USE & Branch for Windows NT Installation and Customization Guide*.

Triggering an Application from MERVA

Application programs can either continually pull message queues using a calling mechanism, the **ENMNextEntry** or **ENMQueryQueue** function, or a triggering service using alarms. This triggering service is an alternative provided by MERVA.

An alarm consists of the alarm name and the semaphore cleared when the alarm is activated.An alarm is associated with a queue and activated when a message enters that particular queue. A queue can have more than one alarm attached to it. How you define alarms to mark when a message enters a certain queue is described in the *MERVA USE & Branch for Windows NT Installation and Customization Guide*.

The following figure shows you an example of an application program that creates a semaphore and sets it. It then waits for the semaphore to be cleared.

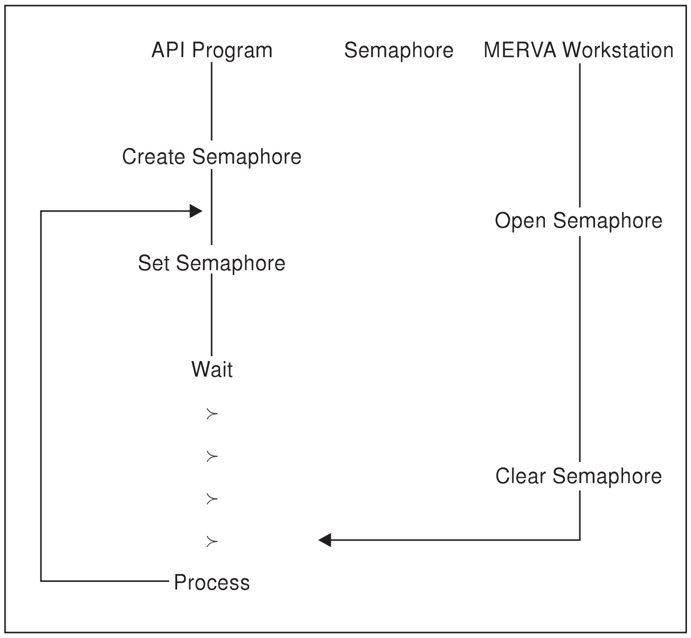

Figure 2. Clearing a Semaphore

If a message enters the specified queue, MERVA raises all alarms that are defined for that queue. One of the alarms clears the semaphore of the application program. The program processes the message, sets the semaphore, and waits for the semaphore to be cleared again.

Message Header Checking

The API includes functions to check whether the header of a message conforms to basic rules of the destination network. The header checking by the API is, however, not a comprehensive check whether the header is valid. A message that passes the API header checking can still be rejected by the network.

The rules for the S.W.I.F.T. and Telex networks are described in “Appendix B. Message Header Checking” on page 149.

Chapter 2. MERVA API Data Types

The API provides you with a set of symbol definitions and data types to facilitate use of the interface.

It also includes shortened forms of standard data types and pointers to them:

```
typedef unsigned char   UCHAR;
typedef unsigned short  USHORT;
typedef USHORT         *PUSHORT;
typedef UCHAR          *PUCHAR;
```

Note: All length definitions in the API header file contain the length of the string excluding the additional character that is needed to store the null terminator for strings. For example:

```
#define UIDlen    8
```

The API also includes definitions of all return codes.

Switch (SWITCH)

This data type indicates the logical states **ON** and **OFF**:

```
typedef enum {;OFF,
               ON
             }SWITCH;
```

The API also supplies a pointer to the SWITCH data type:

```
typedef SWITCH   *PSWITCH;
```

User ID (USERID)

This data type contains the identification of an application to access the MERVA instance. The API supplies the length definition, the data type, and a pointer to the data type:

```
#define UIDlen    8

typedef UCHAR          USERID[UIDlen+1];
typedef USERID        *PUSERID;
```

The user ID must conform to the rules of the MERVA Users program. Refer to the *MERVA USE & Branch for Windows NT User's Guide* for details.

Password (PASSWD)

This data type contains the authorization of an application to access the MERVA instance. The API supplies the length definition, the data type, and a pointer to the data type:

```
#define PWlen    8

typedef UCHAR          PASSWD[PWlen+1];
typedef PASSWD        *PPASSWD;
```

Function ID (FUNCID)

This data type is a reserved data type. Currently, only the function ID **API** is supported. The API supplies the length definition, the data type, and a pointer to the data type:

```
#define FUNClen    3

typedef UCHAR          FUNCID[FUNClen+1];
typedef APPLID        *PFUNCID;
```

Queue Name (QNAME)

This data type contains the name of a MERVA queue. The API supplies the length definition, the data type, and a pointer to the data type:

```
#define QNlen    8

typedef UCHAR          QNAME[QNlen+1];
typedef QNAME         *PQNAME;
```

The queue name must be defined for the API purpose group by using the MERVA Customization program. For more information refer to the *MERVA USE & Branch for Windows NT Installation and Customization Guide*.

Key Type (KEYTYPE)

This data type is an enumerated data type of valid key types to search for a message in a MERVA queue. The API supplies the data type and a pointer to the data type:

```
typedef enum
        {
          KEY_ISN,
          KEY_MRN
        }KEYTYPE;

typedef KEYTYPE     *PKEYTYPE;
```

KEY_ISN, the Input Sequence Number (ISN), is returned by the SWIFT network to uniquely identify each message sent to it. KEY_MRN, the Message Reference Number (MRN), is assigned by MERVA to uniquely identify each message within the system.

Search Key (KEY)

This data type is a union of key types. It is used when a message from a queue is retrieved. The API supplies the length definition, the data type, and a pointer to the data type:

```
#define ISNlen    6
#define MRNlen   16

typedef union
        {
           UCHAR ISN[ISNlen+1];
           UCHAR MRN[MRNlen+1];
        }KEY;

typedef KEY         *PKEY;
```

The keys must have the following format:

Key	Format
ISN	Six digits `Sample ISN:    123456`
MRN	Eight alphanumeric characters, followed by eight numeric digits. Refer to the *MERVA USE & Branch for Windows NT User's Guide* for more information. `Sample MRN:    AIXMERVA12345678`

Message (MMSG)

This data type contains the contents of the message. The API supplies the data type and a pointer to the data type:

```
#define MSG_LENGTH 28000

typedef PUCHAR           MMSG;
typedef MMSG            *PMMSG;
```

A message can be up to 28000 characters long. Note that the definition of **MSG_LENGTH** specifies the maximum length of the message but not the real length.

The following definitions guarantee compatibility with previous API programs for AIX(R):

```
#define MSG MMSG
#define PMSG PMMSG
```

Note that these definitions are only available if **<windows.h>** is not included in the API program.

Field Type (FIELDTYPE)

This data type is an enumerated data type of valid field names for data that is associated with a message of MERVA. The API supplies the data type and a pointer to the data type:

```
typedef enum
        {
          FLD_MRN,
          FLD_ISN,
          FLD_MSGNET,
          FLD_MSGLEN,
          FLD_MSGOK,
          FLD_MSGACK,
          FLD_MSGROUTE,
          FLD_MSGCMNT,
          FLD_TXHEAD,
          FLD_MSGUSER,
          FLD_MSGTRUSR,
          FLD_TXINFO,
          FLD_MSGMAC,
          FLD_MSGPAC
        }FIELDTYPE;

typedef FIELDTYPE       *PFIELDTYPE;
```

The fields serve the following purpose:

Field	Purpose
FLD_MRN	Message reference number; key for the message

FLD_ISN	Input SWIFT number; key for the message
FLD_MSGNET	Destination network for the message
FLD_MSGLEN	Message length
FLD_MSGOK	Message status
FLD_MSGACK	Response of the network to the message if a message has been sent to the network
FLD_MSGROUTE	Additional routing information
FLD_MSGCMNT	Comment
FLD_TXHEAD	Telex header data for the message. The contents of the telex header are listed in "Telex Header (TX_HEADER)" on page 11.
FLD_MSGUSER	Last user who changed the message
FLD_MSGTRUSR	Last user who processed the message, for example, authorized it
FLD_TXINFO	Telex transmission data of the message. The contents of the telex information field are listed in "Telex Information (TX_INFO)" on page 14.
FLD_MSGMAC	Contains the message authentication code (MAC) information
FLD_MSGPAC	Contains the proprietary authentication code (PAC) information

Message-Associated Field (FIELD)

This data type is a union of the data fields associated with a message. The field data type can be accessed via the API. The API supplies the length definition for the fields, the union of fields, and a pointer to the union data type:

```
#define MRNlen        16
#define ISNlen         6
#define OKlen          8
#define ACKlen       127
#define ROUTlen       19
#define CMTlen      1999
#define UIDlen         8
#define MSGMAClen    127
#define MSGPAClen    127
```

```
typedef union
        {
          UCHAR     mrn[MRNlen+1];
          UCHAR     isn[ISNlen+1];
          NETWORK   msgnet;
          USHORT    msglen;
          UCHAR     msgok[OKlen+1];
          UCHAR     msgack[ACKlen+1];
          UCHAR     msgroute[ROUTlen+1];
          UCHAR     msgcomment[CMTlen+1];
          TX_HEADER txhead;
          UCHAR     msguser[UIDlen+1];
          UCHAR     msgtrusr[UIDlen+1];
          TX_INFO   txinfo;
          UCHAR     msgmac[MSGMAClen+1];
          UCHAR     msgpac[MSGPAClen+1];
        }FIELD;
```

```
typedef FIELD           *PFIELD;
typedef PFIELD          *PPFIELD;
```

The fields can contain the following values:

Field	**Value**
mrn	Any valid message reference number. For a description of the format, refer to "Search Key (KEY)" on page 8.
isn	Any valid input sequence number. For a description of the format, refer to "Search Key (KEY)" on page 8.
msgnet	Any value listed in the enumerated data type NETWORK. For a description of the network, refer to "Network (NETWORK)".
msglen	An unsigned short value.
msgok	Any character string up to the length of this field. For values predefined by MERVA, refer to the *MERVA USE & Branch for Windows NT Installation and Customization Guide*.
msgack	Any character string up to the length of this field. For details refer to the *MERVA USE & Branch for Windows NT Installation and Customization Guide*.
msgroute	Any character string up to the length of this field.
msgcomment	Any character string. For details refer to the *MERVA USE & Branch for Windows NT Installation and Customization Guide*.
txhead	Telex header structure. For a description of the telex header, refer to "Telex Header (TX_HEADER)".
msguser	Any character string up to the length of this field.
msgtrusr	Any character string up to the length of this field.
txinfo	Telex info structure. For a description of the telex info, refer to "Telex Information (TX_INFO)" on page 14.
msgmac	Any character string up to the length of this field.
msgpac	Any character string up to the length of this field.

Network (NETWORK)

This data type is an enumerated data type of networks to which MERVA can be connected. The API supplies the following data type:

```
typedef enum
        {
           NET_SWIFT=0x02,
           NET_TELEX=0x03,
           NET_OWN=0x04
        }NETWORK;
```

Telex Header (TX_HEADER)

This data type contains data that is needed to send a message via the telex network. All elements of the structure are strings. The API supplies the length definition structure elements and for the structure itself:

```
#define TESTKEYlen     16
#define TEST_COMMlen   35
#define ADDRlen        35
#define DATElen         8
```

```
#define TO_IDlen       11
#define DIAL_UPlen     20
#define ANSW_BAlen     20
#define LINElen         2
#define TIMElen         4
#define REFlen         16
#define NOTElen        64


typedef struct
        {
           UCHAR testkey_cal     [1+1];
           UCHAR testkey_rc      [1+1];
           UCHAR testkey_val     [TESTKEYlen+1];
           UCHAR testkey_comment1[TEST_COMMlen+1];
           UCHAR testkey_comment2[TEST_COMMlen+1];
           UCHAR sender_addr0    [ADDRlen+1];
           UCHAR sender_addr1    [ADDRlen+1];
           UCHAR sender_addr2    [ADDRlen+1];
           UCHAR sender_addr3    [ADDRlen+1];
           UCHAR date            [DATElen+1];
           UCHAR to_id           [TO_IDlen+1];
           UCHAR receiver_addr0  [ADDRlen+1];
           UCHAR receiver_addr1  [ADDRlen+1];
           UCHAR receiver_addr2  [ADDRlen+1];
           UCHAR receiver_addr3  [ADDRlen+1];
           UCHAR line            [LINElen+1];
           UCHAR dial_up1        [DIAL_UPlen+1];
           UCHAR answ_back1      [ANSW_BAlen+1];
           UCHAR dial_up2        [DIAL_UPlen+1];
           UCHAR answ_back2      [ANSW_BAlen+1];
           UCHAR type            [1+1];
           UCHAR timed_time      [TIMElen+1];
           UCHAR timed_date      [DATElen+1];
           UCHAR ref_text        [REFlen+1];
           UCHAR note            [NOTElen+1];
        } TX_HEADER;
```

All entries to elements of the telex header structure must use the character set shown in the following figure. It consists of the internationally defined telex characters.
The fields of the telex header are as follows:

Letters	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Numbers	1234567890
Others	/.,:+-()=?’

Figure 3. Baudot Character Set

Field	Explanation
testkey_cal	One of the following characters: **Y** Route to testkey calculation (requires correct routing to be set up). **N** Do not route to testkey calculation.
testkey_rc	One of the following characters: (Blank). Testkey calculation has not yet been performed, or has failed. **C** The testkey was calculated by a testkey-processing program. **V** The testkey was verified by a testkey-processing program. **G** The testkey was verified, but the sequence number was not in sequence and therefore not correct. **M** The testkey was calculated or verified manually.
testkey_val	The testkey of the message.
testkey_comment1-2	A comment of two lines, each up to 35 characters long that can be added to the testkey.
sender_addr0-3	Four lines, each up to 35 characters long that contain the sender's address.
date	The date must be in the format **YYYYMMDD**, where **YYYY** is the year, **MM** the month, and **DD** the day.
to_id	The correspondent's identifier up to 11 characters long.
receiver_addr0-3	Four lines, each up to 35 characters long that contain the receiver's address.
line	The telex line number. Numbers only.
dial_up1	The first dial-up number to be used. Numbers, blanks, slashes (/) and dashes (-).
answ_back1	The expected answerback.
dial_up2	The alternative dial-up number to be used. Numbers, blanks, slashes (/) and dashes (-).
answ_back2	The expected alternative answerback.
type	One of the following characters: **N** Normal Messages. **U** Urgent Messages. **T** Timed Messages.
timed_time	The time must be in the format **HHMM**, where **HH** is the hour and **MM** the minutes.
timed_date	The date must be in the format **YYYYMMDD**, where **YYYY** is the year, **MM** the month, and **DD** the day.

ref_text	Message identifier, 16 characters.
note	A comment of up to 64 characters.

Telex Information (TX_INFO)

This data type contains data that describes the transmission process of received telexes and of sent telexes. All structure elements are strings.

```
#define ACK_INFOlen     1
#define TYPElen         1
#define REPORT_NRlen    5
#define DIAL_UPlen     20
#define RCV_AWBlen     20
#define STARTTIMElen   14
#define DURATIONlen     6
#define REASON_CODElen  8
#define TELEX_BOXlen    1
#define TELEX_LINElen   2
#define TERM_CODElen    1
#define EXCEPTIONSlen   1
#define POSS_DUPLlen    1
#define CARGOlen       50


typedef struct
        {
           UCHAR ack_info   [ACK_INFOlen+1];
           UCHAR type       [TYPElen+1];
           UCHAR report_nr  [REPORT_NRlen+1];
           UCHAR dial_up    [DIAL_UPlen+1];
           UCHAR rcv_awb    [RCV_AWBlen+1];
           UCHAR starttime  [STARTTIMElen+1];
           UCHAR duration   [DURATIONlen+1];
           UCHAR reason_code[REASON_CODElen+1];
           UCHAR telex_box  [TELEX_BOXlen+1];
           UCHAR telex_line [TELEX_LINElen+1];
           UCHAR term_code  [TERM_CODElen+1];
           UCHAR exceptions [EXCEPTIONSlen+1];
           UCHAR poss_dupl  [POSS_DUPLlen+1];
           UCHAR cargo      [CARGOlen+1];
        } TX_INFO;
```

The field values are defined by the telex provider but the standard routing of MERVA assumes that the **ack_info** field contains one of the following values:

0	Positive telex transmission acknowledgment
8	Negative telex transmission acknowledgment

If the telex provider does not use these values, you must change the standard routing.

For more detailed information about standard routing, refer to the *MERVA USE & Branch for Windows NT Installation and Customization Guide*.

The following list shows you the fields of the telex information and their explanations. Note that the explanations do not describe the field contents because telex information is created only by the telex provider. For a description of the field contents, refer to the documentation of the telex provider.

Field	Explanation
ack_info	The acknowledgment information indicates whether the telex was positively acknowledged by the telex network. Suggested values are: **0** Positive acknowledgment **8** Negative acknowledgment
type	The type field indicates the direction of the telex, for example, outgoing or received telex.
report_nr	The report sequence number.
dial_up	The dial-up number for transmission.
rcv_awb	The received answerback.
starttime	The starting time of transmission.
duration	The transmission duration.
reason_code	The error message code.
telex_box	The telex box for transmission.
telex_line	The telex line number.
term_code	The box termination code.
exceptions	The exception code indicates a possible error in the telex.
poss_dupl	Possible duplicate indicator.
cargo	Cargo field that the application can use, for example, for the document number.

MERVA recognizes a message as a Telex message if one of the telex header or telex information fields are not empty.

Trace Data (TRACEDATA)

This data type contains information that the application adds to the API trace file. The API supplies the length definition, the data type, and a pointer to the data type:

```
#define DATAlen  240

typedef UCHAR             TRACEDATA[DATAlen+1];
typedef TRACEDATA        *PTRACEDATA;
```

Purpose Group (GROUP)

This data type is an enumerated data type of all purpose groups defined in MERVA. The API supplies the data type and a pointer to the data type:

```
      typedef enum
      {
         /* Message Processing SWIFT Message */
         GROUP_INCOMPLT              = 101,
         GROUP_VERIFY                = 102,
         GROUP_AUTH1                 = 103,
         GROUP_EDIT                  = 104,
         GROUP_AUTH2                 = 105,

         /* SWIFT Link */
         GROUP_SEND                  = 201,
         GROUP_MANL_AUTH             = 202,

         /* MERVA Link */
         GROUP_MLINK_RECEIVE         = 301,
         GROUP_MLINK_TO_SEND         = 302,
         GROUP_MLINK_WAIT_ACK        = 303,
         GROUP_MLINK_CONTROL         = 304,

         /* MERVA Base */
         GROUP_PURGE                 = 401,
         GROUP_DELETE                = 402,
         GROUP_ERROR                 = 403,
         GROUP_UNLOAD                = 404,
         GROUP_PRINT                 = 405,
         GROUP_API                   = 406,
         GROUP_USE_FROM_SWIFT        = 407,
         GROUP_USE_NAKED             = 408,
         GROUP_USE_MT960_WITHOUT_PRE  = 409,
         GROUP_USE_INCOMING_COMMAND   = 410,

         /* Message Processing Telex */
         GROUP_TELEX_CALC_TESTKEY     = 501,
         GROUP_TELEX_NON_ACK          = 502,
         GROUP_TELEX_VERIFY_TESTKEY   = 504
      }GROUP;

typedef GROUP     *PGROUP;
```

Message Console Identifer (CON_MSG_ID)

This data type is an enumerated data type of values to determine the error level. It also defines whether error messages are written only to the diagnosis log or also to the MERVA message console.

```
      typedef enum
      {
         CON_ID_NONE   = '.',
         CON_ID_INFO   = 'I',   /* Information           */
         CON_ID_ERROR  = 'E',   /* Recoverable error     */
         CON_ID_FATAL  = 'F'    /* Non-recoverable error */
      }CON_MSG_ID;
```

Intervention (INTERVENTION)

This data type is an ennumerated data type of values that define whether operator intervention is required for the corresponding error message.

```
typedef enum
{
   INT_NOT_REQ = '.',
   INT_REQ     = 'R'
}INTERVENTION;
```

Right (RIGHTS)

This data type enumerates valid rights to be checked. It is an enumerated data type added to the header file **enmcapi.h**:

```
typedef enum
{
   USER_R1,
   USER_R2,
   USER_R3,
   USER_R4,
   USER_R5,
   USER_R6,
   USER_R7,
   USER_R8,
   USER_R9
}RIGHTS
```

Pointer to Function (PFUNC)

Each function contains a pointer definition. The function type is defined as **PFUNC<function name>**. To use, for example, **ENMAdd**, the following definition is valid:

```
USHORT ENMAdd(QNAME QueueID);          /* prototype of ENMAdd */
typedef USHORT (* PFUNCENMAdd)(QNAME); /* definition of pointer
                                          to function ENMAdd  *
```

You can use this definition to load the library dynamically. For a detailed description refer to "Chapter 4. How to Use, Build, and Load an API Program" on page 121.

Chapter 3. MERVA API Function Calls

This chapter describes in alphabetical order the MERVA API functions. The description of each function is divided into the following parts:

Purpose	A brief description of the function's purpose.
Format	The syntax of the function, its name in mixed case, and the number and order of parameters.
Parameters	A description of each parameter of the function.
Processing (optional)	Conditions that might occur while this function is processed.
Restrictions (optional)	Prerequisites for functions and the validity of data returned by the function or passed to the function.
Example	An example of how to call the function in C language.

Functional Overview

The following API functions are valid:

Instance connection or disconnection

ENMSetAppl()	Set the application name.
ENMAttach()	Start to work on messages.
ENMDetach()	Stop to work on messages.

Message creation

ENMCreate()	Get an empty message.

Message retrieval

ENMKeyRead()	Search for a message in a given queue by using a specified key.
ENMKeyNext()	Search for the next message in a queue by using a specified key.
ENMFirstEntry()	Get the oldest message from the message buffer.
ENMLastEntry()	Get the latest message from the message buffer.
ENMNextEntry()	Get the next message from the message buffer.
ENMPreviousEntry()	Get the previous message from the message buffer.

Note: With these calls, you can lock the message. A locked message can only be changed by the user who locked it.

Message routing

ENMAdd()	Put a new message in a queue.
ENMRouteAdd()	Add the created message to its destination queue.
ENMFree()	Return the original message without any changes to its original queue.
ENMPut()	Return a message to its original queue.
ENMRoutePut()	Put a message in the next queue, as defined by the routing tables.

Accessing information associated with messages

ENMReadField()	Get the contents of one field with information associated with the actual message.
ENMWriteField()	Set a field of associated information. **Note:** To access associated information, a message must be stored in the message space of the API.

Message removal

ENMClear()	Erase the contents of a new message.
ENMDelete()	Delete a message.

Message checking

ENMCheck()	Check whether a message conforms to the rules defined in the message process tables on your system.
ENMCheckSwiftMsg()	Check whether a message conforms to the rules established by S.W.I.F.T., and whether the length of the message is appropriate for the message type.

API trace handling

ENMTrace()	Switch the trace on or off.
ENMWriteTrace()	Add a line to the API trace file.
ENMWriteLog()	Write diagnosis and console log entries.

Status inquiry

ENMQueryQueue()	Get the number of messages currently in a queue. **ENMQueryQueue** can manage up to 65535 messages. The highest number of messages is 2 147 483 647. To get the correct number of messages, use **ENMQueryQueueEx**.
ENMQueryQueueEx()	Get the number of messages currently in a queue. **ENMQueryQueueEx** can manage up to 2 147 483 647 messages. This is the highest number of messages.
ENMWhereIs()	Get the purpose group that a specified message last entered.

API triggering service

ENMCreateSem()	Create a semaphore.
ENMOpenSem()	Open a semaphore.
ENMCloseSem()	Close and delete a semaphore.
ENMSetSem()	Set a semaphore unconditionally.
ENMClearSem()	Clear a semaphore unconditionally.
ENMWaitSemList()	Wait for a list of semaphores.

API User services

ENMCheckUserRight()	Check user rights.

Functions to ensure compatibility between MERVA and MERVA Connection/NT

ENMSetProfile()	In MERVA Connection/NT, this function specifies the name of the profile you want to use. In the local API, this function does nothing.
ENMStartRAPI()	In MERVA Connection/NT, this function establishes the connection to a MERVA server. In the local API, this function calls **ENMSetAppl()**.
ENMRestartRAPI()	In MERVA Connection/NT, this function reconnects to a MERVA server. In the local API, this function calls **ENMSetAppl()**.
ENMEndRAPI()	This function disconnects from the MERVA server if MERVA Connection/NT is called. In the local API, this function does nothing.
ENMSetSecurity()	In MERVA Connection/NT, this function defines the conversation security information. In the local API, this function does nothing.
ENMSetTestEnv()	Enables or disables the test environment in MERVA Connection/NT. In the local API, this function does nothing.

ENMGetReason()	Returns a reason code for internal errors in MERVA Connection/NT. In the local API, this function returns **NO_ERROR (0)**.

ENMAdd—Add Created Message to Queue

Purpose

The **ENMAdd** function adds a created message to a queue of the API purpose group.

Format

```
USHORT ENMAdd(QNAME QueueID)
```

Parameters

QueueID (QNAME) - input
This is the name of a queue known to MERVA. Only queues of the API purpose group can be addressed by the API.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

2 (ERR_SYSTEM)
An error in the MERVA instance occurred.

5 (ERR_NOT_ATTACHED)
The application is not attached to the MERVA instance.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

101 (ERR_NO_QUEUE_NAME)
The specified queue name is empty or too long.

102 (ERR_INVALID_QUEUE_NAME)
The named queue does not belong to the API purpose group or the user has no right to use the named queue.

115 (ERR_SWIFT_HEAD)
The header of the message does not match the rules for SWIFT headers.

The checking level for the header of a SWIFT message is described in “Appendix B. Message Header Checking” on page 149.

116 (ERR_TELEX_HEAD)
The header of the message does not match the rules for telex headers.

The checked fields of the telex header are described in “Telex Header (TX_HEADER)” on page 11.

117 (ERR_NETWORK)
There is no destination network specified for the network.

202 (ERR_NO_MSG_CREATED)
No message has been previously created by the application.

Processing

The message is checked before it is added to a queue. Depending on the value of **FLD_MSGNET**, the message is checked for conformance to S.W.I.F.T or Telex rules.

Restrictions

You must call **ENMCreate** before you call **ENMAdd** because the application can only add a previously created message. The message can be added only once.

For details on how to specify message lengths, refer to the **ENMWriteField** information in "Processing" on page 113.

Example

The following example shows you how to use the **ENMAdd** call to add a new message to an API queue.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;
   CHAR   msgTxt[ 200 ];
   MMSG   msg;
   FIELD  fldAssociated;

   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      if( ENMCreate( &msg ) == NO_ERROR )
      {
         fldAssociated.msgnet = NET_SWIFT;
         rc = ENMWriteField( FLD_MSGNET, &fldAssociated );
         if( rc == NO_ERROR )
         {
            /* create message example type 399 */
            strcpy( msgTxt,
                    "{1:F01VNDPBET2AXXX0000000299}{2:I399VNDPBET2AXXXN}"
                    "{3:{108:399-14}}{4:\r\n:20:399-14\r\n:79:REPLACE MT 101");
            memcpy( msg, msgTxt, strlen(msgTxt) );
            if( ENMAdd( "API_OUT" ) == NO_ERROR )
            {
               printf("New Message added to Queue API_OUT\n");
            }
         }
         else
         {
            printf("Error in ENMWriteField, rc %d\n", rc );
            /* delete the created message */
            rc = ENMClear();
         }
      }
      /* now do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
          printf( "Error in detach %d\n", rc );
      }
      else
      {
          printf("Program detached...\n");
      }
   }
   else
```

```
      {
         printf( "Error attaching to MERVA, return code %d\n", rc );
      }
   }
```

ENMAttach—Attach to MERVA Instance

Purpose

The **ENMAttach** function connects the application program to the MERVA instance. It prepares the interface data structures to pass messages from MERVA to the application and from the application to MERVA.

Format

```
USHORT ENMAttach(USERID UserID, PASSWD Password,
                 FUNCID FunctionID)
```

Parameters

UserID (USERID) - input
The API tries to log on to the MERVA instance with this user ID (USERID). The user ID is defined and approved to start an API application program with the MERVA Users program.

The user ID is ignored or can be **NULL** if the input parameter **FunctionID** is identical to **MEN** because the MERVA logon user ID is used in this case.

Password (PASSWD) - input
This parameter is optional. If it is supplied, the API tries to log on to the MERVA instance with this password. The password is defined with the MERVA Users program. It is only necessary if the user has the access right **API - with password**.

Can be **NULL**. The password is **ignored** if the input parameter **FunctionID** is identical to **MEN** because the MERVA logon password is used in this case.

FunctionID (FUNCID) - input
The function ID for the application program must be specified. The following values are valid:

API
Application program is started from the command line.

MEN
Application program is started from the MERVA Menu window. For more information about API programs in the MERVA menu refer to “Adding an API Program to the MERVA Menu window” on page 123.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

2 (ERR_SYSTEM)
An error occurred in the MERVA instance.

3 (ERR_ATTACH_FAILED)
An application with the same name is already attached.

6 (ERR_OUT_OF_MEMORY)
The application program interface could not allocate the message space.

7 (ERR_WRITE_TRACE)
: An error occurred while writing to the trace file.

9 (ERR_NO_FREE_SLOT)
: All slots to attach to MERVA Base services are currently in use.

10 (ERR_SIGNON_FAILED)
: The application cannot sign on to the MERVA Control Process (daemon). For more information refer to "Appendix A. Return Codes" on page 141.

22 (ERR_NO_API_QUEUE)
: There is no predefined queue that belongs to the API purpose group.

23 (ERR_NO_API_QUEUE_ASSIGNED)
: There is no API queue assigned to the available API access right.

106 (ERR_INVALID_ID)
: The application name is not correct. The name must start with one of the approved prefixes.

109 (ERR_NO_PASSWD)
: The length of the specified password is not correct.

110 (ERR_NO_AUTHORIZATION)
: An authorization problem occurred during the attempt to attach to MERVA. Therefore the application program cannot get a valid API queue.

118 (ERR_NO_USERID)
: The user ID passed to the function is not correct.

119 (ERR_NO_FUNCID)
: The function ID passed to the function is not correct.

207 (ERR_CRC_CHECK)
: The application detected a CRC error on the Control database.

214 (ERR_USERID_NOT_FOUND)
: The supplied user ID is not a defined MERVA user ID.

216 (ERR_RIGHTS_NOT_APPROVED)
: The API access rights assigned to the user are not approved.

217 (ERR_NO_RIGHTS)
: The user is not authorized to execute an API application program, or has the access right **API - with password** but a password is not specified.

406 (ERR_WRONG_PASSWD)
: The specified password does not match your MERVA password.

407 (ERR_USERID_REVOKED)
: The specified user ID is revoked by the MERVA instance.

410 (ERR_NO_PASSWD_SET)
: No initial password is defined for this user in the user maintenance task.

414 (ERR_GET_PSW)
: Reading the user's locally defined password information fails because the MERVA instance does not have root user authority.

415 (ERR_WRONG_AIX_PSW)
: The specified password does not match your Windows NT password.

416 (ERR_NOTIFY_FAILED)
: Cannot get MERVA logon user ID and password.

Processing

The function uses the supplied values of user ID, password, and function ID to check whether the user is authorized to run this API application program.

Note: If every user is allowed to run the API application, integrate the user ID and function ID in the application. Do not specify a password and set the access right to **API - without password**. This allows the user to start API programs only together with MERVA.

Regarding Windows NT signals, consider the following:

- If you use semaphore calls, such as **ENMWaitSemList**, the signal **SIGALRM** is used internally. Then, the time value set in an alarm() call is reset to zero. Be careful when you use this alarm in your own program.
- If the API program is added to the MERVA Menu window, the API must react to the signal **SIGTERM**. When you close this window:
  - All user applications are stopped.
  - The user is logged off from MERVA.

  The logout step sends a **SIGTERM** signal to all running applications. Each user application should stop processing when it receives a **SIGTERM** signal.

Restrictions

Each API program that communicates with MERVA is identified by its name set in the API call **ENMSetAppl()**. The name must be unique. If this call is missing, the program name is used as identifier. For example, only one program with the application ID PGM1 can be attached to MERVA at one time. To prevent a conflict with names used by MERVA, do not specify the application name with one of the following 3-character prefixes:

- **enm**
- **ENM**
- **enn**
- **ENN**
- **eka**
- **EKA**
- **enl**
- **ENL**

Note: Before you can use a user ID for an API program, you have to define it in the MERVA Users program. Even if the user ID has the access right **API - without password**, you have to define a valid password for this user ID.

Example

The following example shows you an **ENMAttach** call to the MERVA instance that uses the application ID **PGM1**, the user ID **SAMPLE**, and the password **SAMPLE1**.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "enmcapi.h"

main()
{
   USHORT rc = 0;

   rc = ENMSetAppl("PGM1");
   printf( "\nENMSetAppl processed, rc = %d\n", rc );

   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");

      /* ... do some processing here ... */

      /* then do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
          printf( "Error in detach %d\n", rc );
      }
   }
   else
   {

      switch ( rc )
      {
         case ERR_USERID_NOT_FOUND:
              printf("Sorry, you are no MERVA user\n");
              break;
         case ERR_RIGHTS_NOT_APPROVED:
              printf("Sorry, your API access right is not approved.\n");
              break;
         case ERR_NO_RIGHTS:
              printf("Sorry, you have no API access right.\n");
              break;
         case ERR_WRONG_PASSWORD:
              printf("Sorry, your password is not valid.\n");
              break;
         case ERR_USERID_REVOKED:
              printf("Sorry, your user ID is revoked.\n");
              break;
         case ERR_NO_PASSWD_SET:
              printf("Sorry, you have no password defined for this user.\n");
              break;
         case ERR_NO_AUTHORIZATION:
              printf("Sorry, other authorization problems occurred.\n");
              break;
         default:
              printf( "Error attaching to MERVA, return code %d\n", rc );
              break;
      }
   }
}
```

The following example shows how to use **ENMAttach** if the sample program is started from the MERVA Menu window.

```
#include <signal.h>
#include "enmcapi.h"

USHORT PGM_Init( int argc );
VOID sig_term();
VOID PGM_Terminate();

main(int argc, CHAR *argv[])
{
   USHORT usRc;
   BOOL   bRight;
   CHAR   achLogbuf[200];
   /* must be done, because program will be started from MENU window */
   if (signal(SIGTERM, (void(*)())sig_term) == SIG_ERR)
      ENMWriteLog("Unable to register signal handler",
                   CON_ID_NONE, INT_NOT_REQ);
   else
   {
      usRc = PGM_Init(argc);
      if (usRc == NO_ERROR)
      {
          /* do some processing here */

         PGM_Terminate();
      }
      else
      {
        sprintf(achLogbuf, "Error calling PGM_Init rc=%d", usRc);
        ENMWriteLog(achLogbuf, CON_ID_NONE, INT_NOT_REQ);
      } /* endif */

   } /* endif */

}

USHORT PGM_Init( int argc )
{
  USHORT usRc = NO_ERROR;
  UCHAR  achLogbuf[200];

  usRc = ENMSetAppl( "PGM1" );
  if (usRc == NO_ERROR)
  {
     /* attach to MERVA that program can be started from MENU window */
     usRc = ENMAttach("", "", "MEN");
     if (usRc == NO_ERROR)
     {
        ENMWriteLog("Program successfully attached to MERVA",
                     CON_ID_NONE, INT_NOT_REQ);
     }
     else
     {
        sprintf(achLogbuf, "Error attaching to MERVA rc=%d", usRc);
        ENMWriteLog(achLogbuf, CON_ID_NONE, INT_NOT_REQ);
     }
  }
  else
  {
     sprintf(achLogbuf, "Error calling ENMSetAppl rc=%d", usRc);
     ENMWriteLog(achLogbuf, CON_ID_NONE, INT_NOT_REQ);
  } /* endif */
  return(usRc);
}


VOID PGM_Terminate()
{
  USHORT usRc;
  UCHAR  achLogbuf[200];

  usRc = ENMDetach();
  if ( usRc != NO_ERROR)
  {
     sprintf(achLogbuf, "Error in detach rc=%d", usRc);
     ENMWriteLog(achLogbuf, CON_ID_NONE, INT_NOT_REQ);
  }
}

VOID sig_term()
```

```
{
   PGM_Terminate();
   exit(0);
}
```

ENMCheck—Checking a Message

Purpose

| The **ENMCheck** function checks whether a message conforms to the rules defined
| in the message process tables on your system. If a checking error is found, it is
| written to the MERVA diagnosis log.

Format

```
USHORT ENMCheck(PUSHORT pusCheckErr)
```

Parameters

pusCheckErr (PUSHORT) - output
This variable specifies whether a checking error was found in the message. Possible checking errors are:

304 (NO_CHECK_ERROR)
No checking error was found, the message is syntactically and semantically correct.

305 (ERR_MSG_SYNTAX)
A syntactical error was found in the message.

306 (ERR_MSG_SEMANTIC)
A semantic error was found in the message.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
Function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not been started.

2 (ERR_SYSTEM)
Installation error. The necessary library could not be found.

5 (ERR_NOT_ATTACHED)
The application is not attached to the MERVA instance.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

202 (ERR_NO_MSG_CREATED)
No message has been retrieved by the application program.

303 (ERR_CHECK_MSG)
A message processing error occurred while checking the message.

| **421** (ERR_NO_DATA)
| Message contains no data.

Example

The following example shows you how to check with **ENMCheck** whether the message is in correct format before it is written to the MERVA message database.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```
#include "enmcapi.h"

USHORT PGM_Init( int argc );
VOID PGM_Terminate();
USHORT load_message();

main(int argc, CHAR *argv[])
{
  USHORT usRc;
  MMSG    Message;             /* Actual storage of the message         */
  CHAR   MsgTxt[200];
  FIELD  fldAssociated;        /* field for network identifier          */
  USHORT usCheckErr;

  usRc = ENMSetAppl( "PGM1" );
  if (usRc == NO_ERROR)
  {
     usRc = ENMAttach("SAMPLE", "SAMPLE1", "API");
     if (usRc == NO_ERROR)
     {
        printf("Program successfully attached to MERVA\n");

        usRc = ENMCreate( &Message );
        if ( usRc == NO_ERROR )
        {
           /* Set the destination network to 'SWIFT network'. */
           memset(&fldAssociated,; '\0', sizeof(FIELD));
           fldAssociated.msgnet = NET_SWIFT;
           usRc = ENMWriteField( FLD_MSGNET, &fldAssociated );
           if ( usRc == NO_ERROR )
           {
              /* create message example type 999 */
              strcpy( MsgTxt,
                      "{1:F01VNDPBET2AXXX0000000000}"
                      "{2:I999IBMADEFFXXXXN}{4:\r\n:20:T20\r\n:79:hallo\r\n-}");
              memcpy( Message, MsgTxt, strlen(MsgTxt));

              /* check whether message is correct */
              usRc = ENMCheck(&usCheckErr);
              if (usRc == NO_ERROR)
              {
                 if (usCheckErr == NO_CHECK_ERROR)
                 {
                    printf("Message syntactically and semantically correct\n");

                    /* add message to database */
                    usRc = ENMAdd( "API_OUT");
                    if ( usRc != NO_ERROR )
                    {
                       printf( "Could not add message to queue, usRc = %d\n", usRc );
                    }
                 }
                 else if (usCheckErr == ERR_MSG_SYNTAX)
                 {
                    printf("Message syntactically not correct\n");
                    usRc = ERR_MSG_SYNTAX;
                 }
                 else if (usCheckErr == ERR_MSG_SEMANTIC)
                 {
                    printf("Message semantically not correct\n");
                    usRc = ERR_MSG_SEMANTIC;
                 }

              else
                 printf("Error calling function ENMCheck, return code = %d\n", usRc);
           }
           else
              printf( "Could not set destination network, usRc = %d\n", usRc );
           ENMClear();
        }
        else
           printf( "\nCould not create a message, rc = %d\n", usRc );

        /* detach from merva */
        usRc = ENMDetach();
        if ( usRc != NO_ERROR)
           printf("Error calling ENMDetach return code = %d\n", usRc);
     }
     else
        printf("Error attaching to MERVA rc=%d\n", usRc);
  }
  else
     printf("Error calling ENMSetAppl rc=%d\n", usRc);
}
```

ENMCheckSwiftMsg—Checking a SWIFT Message

Purpose

The **ENMCheckSwiftMsg** function checks whether a message conforms to the rules established by S.W.I.F.T. It also checks whether the length of the message is appropriate for the message type. If a checking error is found, it is written to the MERVA diagnosis log.

Format

```
USHORT ENMCheckSwiftMsg(PUSHORT pusCheckErr)
```

Parameters

pusCheckErr (PUSHORT) - output
This variable specifies whether a checking error was found in the message. Possible checking errors are:

304 (NO_CHECK_ERROR)
No checking error was found, the message is syntactically and semantically correct. Also the length of the message is correct for the message's message type.

305 (ERR_MSG_SYNTAX)
A syntactical error was found in the message.

306 (ERR_MSG_SEMANTIC)
A semantic error was found in the message.

419 (ERR_MSG_INVALID_LENGTH)
The message is too long for the corresponding message type.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
Function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not been started.

2 (ERR_SYSTEM)
Installation error. The necessary library could not be found.

5 (ERR_NOT_ATTACHED)
The application is not attached to the MERVA instance.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

202 (ERR_NO_MSG_CREATED)
No message has been retrieved by the application program.

303 (ERR_CHECK_MSG)
A message processing error occurred while checking the message.

420 (ERR_NO_MSG_TYPE)
Message buffer contains no message type information.

421 (ERR_NO_DATA)
Message contains no data.

Example

The following example shows you how to check with **ENMCheckSwiftMsg** whether the message is in correct format before it is written to the MERVA message database.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "enmcapi.h"

USHORT PGM_Init( int argc );
VOID PGM_Terminate();
USHORT load_message();

main(int argc, CHAR *argv[])
{
  USHORT usRc;
  MMSG    Message;             /* Actual storage of the message         */
  CHAR   MsgTxt[200];
  FIELD  fldAssociated;        /* field for network identifier          */
  USHORT usCheckErr;

  usRc = ENMSetAppl( "PGM1" );
  if (usRc == NO_ERROR)
  {
     usRc = ENMAttach("SAMPLE", "SAMPLE1", "API");
     if (usRc == NO_ERROR)
     {
        printf("Program successfully attached to MERVA\n");

        usRc = ENMCreate( &Message );
        if ( usRc == NO_ERROR )
        {
           /* Set the destination network to 'SWIFT network'. */
           memset(&fldAssociated,; '\0', sizeof(FIELD));
           fldAssociated.msgnet = NET_SWIFT;
           usRc = ENMWriteField( FLD_MSGNET, &fldAssociated );
           if ( usRc == NO_ERROR )
           {
              /* create message example type 999 */
              strcpy( MsgTxt,
                      "{1:F01VNDPBET2AXXX0000000000}"
                      "{2:I999IBMADEFFXXXXN}{4:\r\n:20:T20\r\n:79:hallo\r\n-}");
              memcpy( Message, MsgTxt, strlen(MsgTxt));

              /* check whether message is correct */
              usRc = ENMCheckSwiftMsg(&usCheckErr);
              if (usRc == NO_ERROR)
              {
                 if (usCheckErr == NO_CHECK_ERROR)
                 {
                    printf("Message syntactically and semantically correct\n");

                    /* add message to database */
                    usRc = ENMAdd( "API_OUT");
                    if ( usRc != NO_ERROR )
                    {
                       printf( "Could not add message to queue, usRc = %d\n", usRc );
                    }
                 }
                 else if (usCheckErr == ERR_MSG_SYNTAX)
                 {
                    printf("Message syntactically not correct\n");
                    usRc = ERR_MSG_SYNTAX;
                 }
                 else if (usCheckErr == ERR_MSG_SEMANTIC)
                 {
                    printf("Message semantically not correct\n");
                    usRc = ERR_MSG_SEMANTIC;
                 }
                 else if (usCheckErr == ERR_MSG_INVALID_LENGTH)
                 {
                    printf("Message too long\n");
                    usRc = ERR_MSG_INVALID_LENGTH;
                 }
              else
                 printf("Error calling function ENMCheckSwiftMsg, return code = %d\n", usRc);
           }
           else
              printf( "Could not set destination network, usRc = %d\n", usRc );
           ENMClear();
        }
        else
           printf( "\nCould not create a message, rc = %d\n", usRc );

        /* detach from merva */
        usRc = ENMDetach();
        if ( usRc != NO_ERROR)
           printf("Error calling ENMDetach return code = %d\n", usRc);
     }
     else
        printf("Error attaching to MERVA rc=%d\n", usRc);
  }
  else
     printf("Error calling ENMSetAppl rc=%d\n", usRc);
}
```

ENMCheckUserRight—Checking User Rights

Purpose

The **ENMCheckUserRight** function checks whether the specified user right is granted to the user starting the API program from the MERVA Menu window. For more information about API programs in the MERVA menu refer to “Adding an API Program to the MERVA Menu window” on page 123.

Format

```
USHORT ENMCheckUserRight(RIGHTS Right, PBOOL bRight)
```

Parameters

Right (RIGHTS) - input
: User right to be checked.

bRight (PBOOL) - input
: Values are:

  TRUE
  : The specified right is granted to the user.

  FALSE
  : The specified right is not granted to the user.

rc (USHORT) - return
: Values are:

  0 (NO_ERROR)
  : The function completed successfully.

  417 (ERR_NO_NOTIFY)
  : **ENMAttach** was done with wrong function ID.

  418 (ERR_CKRIGHT_FAILED)
  : **ENMCheckUserRight** failed due to an internal error.

Example

The following example shows you how to check with **ENMCheckUserRight** whether user right 1 is granted to the user who is logged on to MERVA.

```
#include <signal.h>
#include "enmcapi.h"

USHORT PGM_Init( int argc );
VOID sig_term();
VOID PGM_Terminate();

main(int argc, CHAR *argv[])
{
   USHORT usRc;
   BOOL   bRight;
   CHAR   achLogbuf[200];

   /* must be done, because program will be started from MENU window */
   if (signal(SIGTERM, (void(*)())sig_term) == SIG_ERR)
      ENMWriteLog("Unable to register signal handler",
                  CON_ID_NONE, INT_NOT_REQ);
   else
   {
      usRc = PGM_Init(argc);
      if (usRc == NO_ERROR)
      {
         /* check whether user right 1 is granted to the user */
         usRc = ENMCheckUserRight(USER_R1, &bRight);
         if (usRc == NO_ERROR)
         {
             if (bRight == TRUE)
             {
                ENMWriteLog("Logged on user has user right 1",
                              CON_ID_NONE, INT_NOT_REQ);

                /* do some processing here */;

             }
             else
             {
                ENMWriteLog("Logged on user does not have user right 1",
                             CON_ID_NONE, INT_NOT_REQ);
             } /* endif */
         }
         else
         {
            sprintf(achLogbuf, "Error calling ENMCheckUserRight rc=%d", usRc);
            ENMWriteLog(achLogbuf, CON_ID_NONE, INT_NOT_REQ);
         } /* endif */
         PGM_Terminate();
      }
      else
      {
        sprintf(achLogbuf, "Error calling PGM_Init rc=%d", usRc);
        ENMWriteLog(achLogbuf, CON_ID_NONE, INT_NOT_REQ);
      } /* endif */

   } /* endif */
}

USHORT PGM_Init( int argc )
{
  USHORT usRc = NO_ERROR;
  UCHAR  achLogbuf[200];

  usRc = ENMSetAppl( "PGM1" );
  if (usRc == NO_ERROR)
  {
     /* attach to MERVA that program can be started from MENU window */
     usRc = ENMAttach("", "", "MEN");
     if (usRc == NO_ERROR)
     {
        ENMWriteLog("Program successfully attached to MERVA",
                     CON_ID_NONE, INT_NOT_REQ);
     }
     else
     {
        sprintf(achLogbuf, "Error attaching to MERVA rc=%d", usRc);
        ENMWriteLog(achLogbuf, CON_ID_NONE, INT_NOT_REQ);
     }
  }
  else
  {
     sprintf(achLogbuf, "Error calling ENMSetAppl rc=%d", usRc);
     ENMWriteLog(achLogbuf, CON_ID_NONE, INT_NOT_REQ);
  } /* endif */
  return(usRc);
}

VOID PGM_Terminate()
{
  USHORT usRc;
  UCHAR  achLogbuf[200];
  usRc = ENMDetach();
  if ( usRc != NO_ERROR)
  {
     sprintf(achLogbuf, "Error in detach rc=%d", usRc);
     ENMWriteLog(achLogbuf, CON_ID_NONE, INT_NOT_REQ);
  }
}

VOID sig_term()
{
   PGM_Terminate();
   exit(0);
}
```

ENMClear—Free Created Message

Purpose

The **ENMClear** function frees the API message space of a message that is created with the **ENMCreate** function. For details of the **ENMCreate** function refer to "ENMCreate—Create New Message" on page 47.

Format

```
USHORT ENMClear()
```

Parameters

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

5 (ERR_NOT_ATTACHED)
The application is not attached to the MERVA instance.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

202 (ERR_NO_MSG_CREATED)
No message was previously created by the application.

Example

The following example shows you how to use **ENMClear** to clear the memory areas used by the **ENMCreate** call to detach from the MERVA instance:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;
   FIELD  Field;
   PFIELD pField = &Field;
   MMSG   msgBuffer;

   rc = ENMSetAppl("PGM1");

   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      rc = ENMCreate( &msgBuffer );
      if( rc == NO_ERROR )
      {
         ENMClear();
      }
      else
      {
         printf( "Error in ENMCreate, rc = %d\n", rc );
      }
      /* now do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
         printf( "Error in detach %d\n", rc );
      }
      else
      {
         printf("Program detached...\n");
      }
   }
   else
   {
     printf( "Error attaching to MERVA, return code %d\n", rc );
   }
}
```

ENMClearSem—Clear a Semaphore

Purpose

The **ENMClearSem** function clears a semaphore unconditionally. If processes are blocked on the semaphore, they are restarted.

Format

```
USHORT ENMClearSem(ULONG SemHandle)
```

Parameters

SemHandle (ULONG) - input
Semaphore handle created by **ENMCreateSem** or **ENMOpenSem**.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

| **1** (ERR_SYSTEM_NOT_UP)
| The MERVA instance has not been started.

6 (ERR_OUT_OF_MEMORY)
The system does not have enough memory to complete the function.

| **7** (ERR_WRITE_TRACE)
| An error ocurred while writing to the trace file.

255 (ERR_SEMAPHORE_FAILED)
The semaphore call failed with an internal error.

Example

The following example shows you how to use **ENMClearSem** to unblock a process that is waiting for a signal.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

#define  TRIGGER  "apisem"
main()
{
   USHORT  lRc = 0;
   ULONG   SemHandle;

   lRc = ENMOpenSem(&SemHandle, TRIGGER);
   if( lRc == NO_ERROR )
   {
      printf("Semaphore successfully opened by MERVA\n");

      /* ... do some processing here ... */

      /* if another process waits until this step is ready */
      /* clear now the semaphore                           */
      lRc = ENMClearSem( SemHandle );
      if( lRc == NO_ERROR )
      {
         printf("Semaphore successfully cleared\n");
      }
      else
      {
        printf( "Error in ENMClearSem, rc = %d\n", lRc );
      }
      /* close the semaphore, it will be automatically */
      /* deleted by the last ENMCloseSem call          */
      lRc = ENMCloseSem(SemHandle);
   }
   else
   {
      printf( "Error opening a semaphore by MERVA, return code %d\n", lRc );
   }
}
```

ENMCloseSem—Close a Semaphore

Purpose

The **ENMCloseSem** function closes a semaphore that is obtained with an **ENMCreateSem** or **ENMOpenSem** call.

Format

```
USHORT ENMCloseSem(ULONG SemHandle)
```

Parameters

SemHandle (ULONG) - input
Semaphore handle created by **ENMCreateSem** or **ENMOpenSem**.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

| **1** (ERR_SYSTEM_NOT_UP)
| The MERVA instance has not been started.

6 (ERR_OUT_OF_MEMORY)
The system does not have enough memory to complete the function.

| **7** (ERR_WRITE_TRACE)
| An error ocurred while writing to the trace file.

31 (ERR_SEMAPHORE_NO_AUTHORITY)
The user is not authorized to delete the semaphore.

255 (ERR_SEMAPHORE_FAILED)
The semaphore call failed with an internal error.

Processing

The function closes a semaphore and decrements an internal counter by 1. On the opposite, the **ENMCreateSem** call increases this counter by 1, and each **ENMOpenSem** call also increases the counter by 1. If the counter reaches 0, the semaphore is automatically removed from the system. For each **ENMCreateSem** and **ENMOpenSem** call, you must call **ENMCloseSem**. This ensures that no semaphore remains in the system after the application ends.

Example

The following example shows you how to use an **ENMCloseSem** call to delete the created semaphore from the system.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

#define  TRIGGER  "apisem"
main()
{
   USHORT lRc = 0;
   ULONG  SemHandle;

   lRc = ENMCreateSem(&SemHandle, TRIGGER);
   if( lRc == NO_ERROR )
```

```
      {
         printf("Semaphore successfully created by MERVA\n");

         /* ... do some processing here ... */

         lRc = ENMCloseSem( SemHandle );
         if( lRc == NO_ERROR )
         {
            printf("Semaphore successfully deleted\n");
         }
         else
         {
           printf( "Error in ENMCloseSem, rc = %d\n", lRc );
         }
      }
      else
      {
         printf( "Error creating a semaphore by MERVA, return code %d\n", lRc );
      }
   }
```

ENMCreate—Create New Message

Purpose

The **ENMCreate** function prepares a new, empty message for the application.

Format

```
USHORT ENMCreate(PMMSG Message)
```

Parameters

Message (PMMSG) - output
Reference to empty message.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

2 (ERR_SYSTEM)
An error in the MERVA instance occurred.

5 (ERR_NOT_ATTACHED)
The application is not attached to the MERVA instance.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

204 (ERR_MSG_INUSE)
The message is locked by the application, or a created message already exists in the message space.

Processing

The function requests a message reference number (MRN) from MERVA to identify the new message. The MRN can be referenced by an **ENMReadField** call. This message is further processed with an **ENMAdd** or **ENMRoutePut** call.

A created message is locked immediately. Use an **ENMClear** call to free the API message space. **ENMFree** can only unlock a message that is in the message database.

Example

The following example shows you how to use an **ENMCreate** call to create a new message. The MRN of the new message is printed, then an **ENMDetach** is attempted. This causes the error ERR_MSG_INUSE (204) because the message is still in use.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;
   FIELD  Field;
   PFIELD pField = &Field;
   MMSG   msgBuffer;

   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      rc = ENMCreate( &msgBuffer );
      if( rc == NO_ERROR )
      {
         printf("New Message created\n");
         rc = ENMReadField( FLD_MRN, (PPFIELD)&pField );
         if( rc == NO_ERROR )
         {
             printf( "The MRN field contains: %s\n", pField );
         }
         else
         {
             printf("Error in read field, rc = %d\n", rc );
         }
      }
      else
      {
         printf( "Error in ENMCreate, rc = %d\n", rc );
      }
   }
   else
   {
      printf( "Error attaching to MERVA, return code %d\n", rc );
   }
   rc = ENMClear();
   /* now do the detach from MERVA */
   rc = ENMDetach();
   if( rc != NO_ERROR )
   {
     printf( "Error in detach %d\n", rc );
   }
   else
   {
     printf("Program detached...\n");
   }
}
```

ENMCreateSem—Create a Semaphore

Purpose

The **ENMCreateSem** function creates a semaphore. The semaphore is used by several API programs to wait for MERVA alarms.

Format

```
USHORT ENMCreateSem(PULONG pSemHandle, PCHAR pszSemName)
```

Parameters

pSemHandle (PULONG) - output
Address of the semaphore handle.

pszSemName (PCHAR) - input
Pointer to a null-terminated string containing the name of the semaphore to be created.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not been started.

6 (ERR_OUT_OF_MEMORY)
The system does not have enough memory to complete the function.

| **7** (ERR_WRITE_TRACE)
| An error ocurred while writing to the trace file.

100 (ERR_TOO_MANY_SEMAPHORES)
The count of semaphores available on the system exceeds the maximum value.

123 (ERR_INVALID_SEMAPHORE_NAME)
The semaphore name is an invalid Windows NT file name.

183 (ERR_SEMAPHORE_ALREADY_EXISTS)
The semaphore already exists.

255 (ERR_SEMAPHORE_FAILED)
The semaphore call failed with an internal error.

Example

The following example shows you how to use an **ENMCreateSem** call to create a semaphore. The semaphore is set and the program waits until it gets a clear signal.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

#define  TRIGGER  "apisem"
main()
{
   USHORT  usIndex = 0;
   USHORT  lRc = 0;
   ULONG   SemHandle;
   lRc = ENMSetAppl("PGM1");

   lRc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( lRc == NO_ERROR )
   {

      lRc = ENMCreateSem(&SemHandle, TRIGGER);
      if( lRc == NO_ERROR )
      {
         printf("Semaphore successfully created by MERVA\n");

         lRc = ENMSetSem(SemHandle);

         /* Wait indefinitely until the semaphore will be cleared. */
         /* For example, if a message reaches an API queue and the */
         /* defined alarm will be raised.                          */
         lRc = ENMWaitSemList(&usIndex, -1, SemHandle, 0);

         /* ... do some processing with the received message ... */

         lRc = ENMCloseSem( SemHandle );
         if( lRc == NO_ERROR )
         {
            printf("Semaphore successfully deleted\n");
         }
         else
         {
           printf( "Error in ENMCloseSem, rc = %d\n", lRc );
         }
      }
      else
      {
         printf( "Error creating a semaphore by MERVA, return code %d\n", lRc );
      }
      lRc = ENMDetach();
   }
   else
   {
      printf( "Error attaching to MERVA, return code %d\n", lRc );
   }
}
```

ENMDelete—Delete Current Message from Queue

Purpose

The **ENMDelete** function deletes the currently locked message.

Format

```
USHORT ENMDelete()
```

Parameters

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

2 (ERR_SYSTEM)
An error in the MERVA instance occurred.

5 (ERR_NOT_ATTACHED)
The application is not attached to the MERVA instance.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

201 (ERR_NO_MSG_LOCKED)
No message has been locked by the application.

301 (ERR_MSG_LOCKED)
Another application program has also locked this message.

302 (ERR_MSG_NOT_FOUND)
The message could not be found in the queue.

Example

The following example shows you how to read all messages in the API_OUT queue and uses the **ENMDelete** call to delete the messages from this queue. The number of deleted messages is displayed.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;
   MMSG   msg;
   USHORT usLen;
   FIELD  Field;
   PFIELD pField = &Field;
   USHORT i;

   rc = ENMSetAppl("PGM1");
   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      /* we do not lock the message */
      do
      {
         /* Read next with lock */
         rc = ENMNextEntry( "API_OUT", ON, &msg, &usLen );
         if( rc == NO_ERROR )
         {
            ENMDelete();
            i++;
         }
      } while( rc == NO_ERROR );
      printf("%d Messages deleted in queue API_OUT\n", i);
      if( rc != ERR_MSG_NOT_FOUND )
      {
         printf( "Error in ENMNextEntry, rc = %d\n", rc );
      }
      /* now do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
         printf( "Error in detach %d\n", rc );
      }
      else
      {
         printf("Program detached...\n");
      }
   }
   else
   {
      printf( "Error attaching to MERVA, return code %d\n", rc );
   }
}
```

ENMDetach—Detach from MERVA Instance

Purpose

The **ENMDetach** function disconnects the application from the MERVA instance. An application with the same name can now attach to the MERVA instance.

Format

```
USHORT ENMDetach()
```

Parameters

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

4 (ERR_DETACH_FAILED)
The detach failed due to an internal error.

5 (ERR_NOT_ATTACHED)
The application is not attached to the MERVA instance.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

204 (ERR_MSG_INUSE)
The message is locked by the application, or a created message already exists in the message space.

Example

The following example shows you an **ENMDetach** call to disconnect a connection between the API program and the MERVA instance.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "enmcapi.h"

main()
{
   USHORT rc = 0;

   rc = ENMSetAppl("PGM1");
   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      /* ... do some processing here ... */

      /* then do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
        printf( "Error in detach %d\n", rc );
      }
   }
   else
```

```
    {
       printf( "Error attaching to MERVA, return code %d\n", rc );
    }
}
```

ENMEndRAPI—Disconnect from the MERVA System

Purpose

In MERVA Connection/NT, the **ENMEndRAPI** function stops the conversation with the remote MERVA system. To ensure compatibility between MERVA API programs and MERVA Connection/NT programs, this function is also provided for the local MERVA system, however for the local system it is only a dummy function that always returns 0 and otherwise does nothing.

Format

```
USHORT ENMEndRAPI()
```

Parameters

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

2 (ERR_SYSTEM)
An internal error occurred. The API receives further information by calling the function **ENMGetReason** (see "ENMGetReason—Get Reason Code for Internal Error" on page 61).

Example

The following example shows you an **ENMEndRAPI** call.

```
#include "enmcapi.h" /* or include "enmrapi.h" for MERVA Connection/NT */

               ...
               ENMSetProfile("enm6ri.prf");
               ENMStartRAPI("SAMPLE");
               /* ... do some API calls */
               if (ENMEndRAPI()!=0)
                   /* do error handling */
               ...
```

ENMFirstEntry—Read First Message of Queue

Purpose

The **ENMFirstEntry** function returns the oldest message in the queue.

Format

```
USHORT ENMFirstEntry(QNAME QueueID, SWITCH Lock,
                     PMMSG Message, PUSHORT MessageLength)
```

Parameters

QueueID (QNAME) - input
: This is the name of a queue known to MERVA. Only queues of the API group can be addressed by the API.

Lock (SWITCH) - input
: The variable defines whether the read message is locked for update.

Message (PMMSG) - output
: Message containing the information retrieved from the message.

MessageLength (PUSHORT) - output
: Short integer variable containing the length of the found message.

rc (USHORT) - return
: Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

2 (ERR_SYSTEM)
An error in the MERVA instance occurred.

5 (ERR_NOT_ATTACHED)
The application is not attached to the MERVA instance.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

101 (ERR_NO_QUEUE_NAME)
The specified queue name is empty or too long.

102 (ERR_INVALID_QUEUE_NAME)
The named queue does not belong to the API purpose group, or the user is not allowed to use the named queue.

107 (ERR_NOT_SWITCH)
A value other than ON or OFF has been passed in a variable that has a SWITCH data type.

204 (ERR_MSG_INUSE)
The message is either locked by the application, or a created message already exists in the message space.

302 (ERR_MSG_NOT_FOUND)
No matching message could be found.

Processing

With this call, the messages in a queue are sorted according to the time at which they entered the queue. With the **ENMxxxEntry** calls, the system maintains a position pointer for each queue. An application can switch between queues and resume at the point from where it switched to another queue.

If the **ENMFirstEntry** call is used with the lock set to **ON**, the first unlocked message will be returned.

If the message is read with the lock set to **ON**, no other program can change the message. Use **ENMFree** to unlock the message.

Example

The following example shows you how to use the **ENMFirstEntry** call to read the first message in the API_OUT queue and display the MRN of this message.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;
   MMSG   msg;
   USHORT usLen;
   FIELD  Field;
   PFIELD pField = &Field;

   rc = ENMSetAppl("PGM1");
   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      /* we do not lock the message */
      rc = ENMFirstEntry( "API_OUT", OFF, &msg, &usLen );
      if( rc == NO_ERROR )
      {
         rc = ENMReadField( FLD_MRN, (PPFIELD)&pField );
         if( rc == NO_ERROR )
         {
            printf( "The MRN of the first is: %s\n", pField );
         }
         else
         {
            printf("Error in ENMReadField, rc = %d\n", rc );
         }
      }
      else
      {
         printf( "Error in ENMFirstEntry, rc = %d\n", rc );
      }
      /* now do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
        printf( "Error in detach %d\n", rc );
      }
      else
      {
        printf("Program detached...\n");
      }
   }
   else
```

```
    {
       printf( "Error attaching to MERVA, return code %d\n", rc );
    }
}
```

ENMFree—Unlock Message

Purpose

The **ENMFree** call unlocks a previously locked message. The message can be locked by another application.

Format

```
USHORT ENMFree()
```

Parameters

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

2 (ERR_SYSTEM)
An error in the MERVA instance occurred.

5 (ERR_NOT_ATTACHED)
The application is not attached to the MERVA instance.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

201 (ERR_NO_MSG_LOCKED)
No message has been locked by the application.

301 (ERR_MSG_LOCKED)
Another application program has also locked this message.

302 (ERR_MSG_NOT_FOUND)
The message could not be found in the queue.

Restrictions

An **ENMxxxEntry** call with the lock set to **ON** must have been issued already. An application can only free a message that it has previously locked.

Example

The following example shows you how to use **ENMFree** to remove the lock on a message.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;
   MMSG   msg;
   USHORT usLen;
   FIELD  Field;
   PFIELD pField = &Field;

   rc = ENMSetAppl("PGM1");
   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      /* we lock the message */
      rc = ENMNextEntry( "API_OUT", ON, &msg, &usLen );
      if( rc == NO_ERROR )
      {
         printf("Message locked\n");
         rc = ENMReadField( FLD_MRN, (PPFIELD)&pField );
         if( rc == NO_ERROR )
         {
            printf( "The MRN field contains: %s\n", pField );
         }
         else
         {
            printf("Error in ENMReadField, rc = %d\n", rc );
         }
         /* now we decide to allow others to work on this message */
         /* again, so set it free                                  */
         rc = ENMFree();
         if( rc == NO_ERROR)
         {
            printf("Message freed again\n");
         }
      }
      /* now do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
        printf( "Error in detach %d\n", rc );
      }
      else
      {
        printf("Program detached...\n");
      }
   }
   else
   {
      printf( "Error attaching to MERVA, return code %d\n", rc );
   }
}
```

ENMGetReason—Get Reason Code for Internal Error

Purpose

In MERVA Connection/NT, the **ENMGetReason** function returns the reason code for an internal error. To ensure compatibility between MERVA API programs and MERVA Connection/NT programs, this function is also provided for the local MERVA system, however for the local system it is only a dummy function that always returns 0 and otherwise does nothing.

Format

```
USHORT ENMGetReason()
```

Parameters

rc (USHORT) - return
Possible values are:

0	No Connection/NT error occurred. Either a previous API call was successful or an internal error in the MERVA API (not in the remote program) occurred.
2xxx	Reason codes from **2000** to **2999** indicate communication problems.
2110	The APPC conversation cannot be established or is cancelled.
2120	The Communications Side Information object is not found.
2130	The connection to the Remote MERVA API Server program failed.
2140	Deallocation failed because the conversation has already been stopped.
2150	The conversation was interrupted while the program tried to receive data.
2200	An empty data buffer was received.
28xx	xx is a return code of the TCP/IP service programs.
29xx	xx is a return code of the CPI-C call.
2999	A general communication problem occurred. For details refer to the diagnosis log.
3xxx	An internal semaphore error occurred. **xxx** is the error number provided by Windows NT.
7006	The Remote MERVA API Server failed while the program tried to allocate memory.
7012	The Remote MERVA API Server does not accept further API calls due to a previous error.
7013	The Remote MERVA API Server received a negative return code from user exit **ENM4ExitDecrypt**.
7014	The Remote MERVA API Server received a negative return code from user exit **ENM4ExitEncrypt**.
7015	The Remote MERVA API Server received a negative return code from user exit **ENM4ExitMacVerify** or **ENM4ExitMacGen**.
7016	The Remote MERVA API Server received an incorrect API request.

7018	The Remote MERVA API Server received an error while the program converted ASCII to EBCDIC. For details refer to the diagnosis log of MERVA.
7019	The Remote MERVA API Server received an error while the program accessed the message integrity control file.
7030	Internal message space was not created.
8002	The Remote MERVA API Client cannot open the programmer's log file that is specified in the profile.
8003	The Remote MERVA API Client cannot close the programmer's log file that is specified in the profile.
8004	The Remote MERVA API Client cannot open the diagnosis log file that is specified in the profile.
8005	The Remote MERVA API Client cannot close the diagnosis log file that is specified in the profile.
8006	The Remote MERVA API Client could not allocate memory.
8007	The Remote MERVA API Client cannot write to the diagnosis log file that is specified in the profile.
8008	The Remote MERVA API Client cannot write to the programmer's log file that is specified in the profile.
8010	The Remote MERVA API Client failed because the profile name in **ENMSetProfile** is incorrect or not specified.
8011	The Remote MERVA API Client failed because the profile specified in **ENMSetProfile** does not exist.
8013	The Remote MERVA API Client received a negative return code from user exit **ENM4ExitDecrypt**.
8014	The Remote MERVA API Client received a negative return code from user exit **ENM4ExitEncrypt**.
8015	The Remote MERVA API Client received a negative return code from user exit **ENM4ExitMacVerify**.
8016	The Remote MERVA API Client received a negative return code from user exit **ENM4ExitMacGen**.
8017	The conversation has not been started with **ENMStartRAPI** or with **ENMStartAPPC**.
8019	The Remote MERVA API Client could not access the message integrity control file.
8020	The Remote MERVA API Client could not load the file **ENMRATP.DLL**.
8021	The profile does not contain information about the partner system.

Example

The following example shows you an **ENMGetReason** call.

```
#include "enmcapi.h" /* or include "enmrapi.h" for MERVA Connection/NT */

             USHORT rc     = 0;
             USHORT reason = 0;

             ...
```

```
rc = ENMFree();
if (rc)
{
  reason = ENMGetReason();
  if (reason)
    printf("Internal error, reason code %d",reason);
  else
    printf("Internal error, unknown reason");
}
...
```

ENMKeyNext—Read Next Message with Key

Purpose

The **ENMKeyNext** function searches for the next message that matches the conditions set by a previous **ENMKeyRead** call. It searches for the identical key value, for example, the same MRN in the queues.

Format

```
USHORT ENMKeyNext(SWITCH Lock, PMMSG Message, PUSHORT MessageLength)
```

Parameters

Lock (SWITCH) - input
: The variable defines whether the message read is locked for update.

Message (PMMSG) - output
: Message containing the information retrieved from the message.

MessageLength (PUSHORT) - output
: Short integer variable containing the length of the message found.

rc (USHORT) - return
: Values are:

  0 (NO_ERROR)
  : The function completed successfully.

  1 (ERR_SYSTEM_NOT_UP)
  : The MERVA instance has not yet been started.

  2 (ERR_SYSTEM)
  : An error occurred in the MERVA instance.

  5 (ERR_NOT_ATTACHED)
  : The application is not attached to the MERVA instance.

  7 (ERR_WRITE_TRACE)
  : An error occurred while writing to the trace file.

  103 (ERR_NO_KEY)
  : The specified key is empty.

  107 (ERR_NOT_SWITCH)
  : A value other than ON or OFF is passed in a variable that has a SWITCH data type.

  204 (ERR_MSG_INUSE)
  : The message is either locked by the application, or a created message already exists in the message space.

  302 (ERR_MSG_NOT_FOUND)
  : No matching message could be found.

Processing

If the message is read with the lock set to **ON**, no other program can change the message. Use **ENMFree** to unlock the message.

If you call **ENMKeyNext** with the lock set to **ON**, the next unlocked message in the queue that matches the key is returned.

Restrictions

The function requires that search conditions are defined. Search conditions can only be defined with a call to the **ENMKeyRead** function.

Example

The following example program shows you how to use **ENMKeyNext** to search for the next message matching the conditions set in a previous ENMKeyRead call (KEY_MRN). The first 50 bytes of a message matching the set conditions are printed.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;
   UCHAR  mrnin[ MRNlen + 1 ];
   KEY    key;
   MMSG   msg;
   USHORT usLen;
   FIELD  Field;
   PFIELD pField = &Field;

   rc = ENMSetAppl("PGM1");
   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      printf("Please enter the MRN from which the read begins: ");
      scanf( "%s", mrnin );
      strcpy( key.MRN, mrnin );
      /* we do not lock the message */
      rc = ENMKeyRead( "API_OUT", KEY_MRN, key, OFF, &msg, &usLen );
      if( rc == NO_ERROR )
      {
         printf("MRN: %s \nFirst 50 bytes %50.50s\n----\n", mrnin, msg );

         rc = ENMKeyNext( OFF, &msg, &usLen );

         if( rc == NO_ERROR )
         {
            printf("Second instance of MRN %s found\n", mrnin );
            printf("First 50 bytes %50.50s\n", msg );
         }
         else
         {
            if( rc == ERR_MSG_NOT_FOUND )
            {
               printf("No second instance of same MRN found\n");
            }
            else
            {
               printf( "Error in KeyNext, rc %d\n", rc );
            }
         }
      }
      else
      {
         printf( "Error in ENMKeyRead, rc = %d\n", rc );
      }
      /* now do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
         printf( "Error in detach %d\n", rc );
      }
      else
      {
         printf("Program detached...\n");
      }
   }
   else
   {
      printf( "Error attaching to MERVA, return code %d\n", rc );
   }
}
```

ENMKeyRead—Read Message from Queue by Key

Purpose

The **ENMKeyRead** function searches the named queue for the first message with the specified key.

Format

```
USHORT ENMKeyRead(QNAME QueueID, KEYTYPE KeyType, KEY Key,
                  SWITCH Lock, PMMSG Message,
                  PUSHORT MessageLength)
```

Parameters

QueueID (QNAME) - input
: The name of a queue known to MERVA. Only queues of the API group can be addressed by the API.

KeyType (KEYTYPE) - input
: Defines the type of key to search for. The supported types are defined in the enumerated data type.

Key (KEY) - input
: The identification of a message in a queue. If a key has a special format, the API checks for this format.

Lock (SWITCH) - input
: Defines whether the message read is locked for update.

Message (PMMSG) - output
: Message containing the information retrieved from the message.

MessageLength (PUSHORT) - output
: Short integer variable containing the length of the message found.

rc (USHORT) - return
: Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

2 (ERR_SYSTEM)
An error in the MERVA instance occurred.

5 (ERR_NOT_ATTACHED)
The application is not attached to the MERVA instance.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

101 (ERR_NO_QUEUE_NAME)
The specified queue name is empty or too long.

102 (ERR_INVALID_QUEUE_NAME)
The named queue does not belong to the API purpose group or the user has no right to use the named queue.

103 (ERR_NO_KEY)
The specified key is empty or too long.

104 (ERR_INVALID_MRN)
The specified MRN has an invalid format.

105 (ERR_INVALID_ISN)
The specified ISN has an invalid format.

107 (ERR_NOT_SWITCH)
A value other than ON or OFF is passed in a variable with the SWITCH data type.

108 (ERR_INVALID_KEYTYPE)
The specified key type is not in the range of the KEYTYPE enumerated data type.

204 (ERR_MSG_INUSE)
The message is locked by the application, or a created message already exists in the message space.

302 (ERR_MSG_NOT_FOUND)
No matching message could be found.

Processing

The search starts at the first element of the queue. Therefore, the function returns the same message if neither the key nor the queue change.

Valid key types are defined through the KEYTYPE enumerated data type. If the key follows a specified format, the MERVA API rejects keys that do not match this format.

If the message is read with the lock set to **ON**, no other program can change the message. Use **ENMFree** to unlock the message.

Restrictions

A call to this function erases all information about a key search with a prior call.

Example

The following example program requests an MRN and uses this MRN in the **ENMKeyRead** call (KEY_MRN). The length, the first 20 bytes, and the network identifier of the message are printed if the call is successful. For a description of the network identifiers, refer to "Network (NETWORK)" on page 11.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;
   UCHAR  mrnin[ MRNlen + 1 ];
   KEY    key;
   MMSG   msg;
   USHORT usLen;
   FIELD  Field;
   PFIELD pField = &Field;

   rc = ENMSetAppl("PGM1");
   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      printf("Please enter the MRN from which the read begins: ");
      scanf( "%s", mrnin );
      strcpy( key.MRN, mrnin );
      /* we do not lock the message */
      rc = ENMKeyRead( "API_OUT", KEY_MRN, key, OFF, &msg, &usLen );
      if( rc == NO_ERROR )
      {
         printf("Message length %d\n", usLen );
         printf("First 20 bytes %20.20s\n", msg );
         rc = ENMReadField( FLD_MSGNET, (PPFIELD)&pField );
         if( rc == NO_ERROR )
         {
            printf("Network Identifier is: %d\n", pField->msgnet);
         }
         else
         {
            printf("Error in ReadField, rc = %d\n", rc );
         }
      }
      else
      {
         printf( "Error in ENMKeyRead, rc = %d\n", rc );
      }
      /* now do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
         printf( "Error in detach %d\n", rc );
      }
      else
      {
         printf("Program detached...\n");
      }
   }
   else
   {
      printf( "Error attaching to MERVA, return code %d\n", rc );
   }
}
```

ENMLastEntry—Read Last Message of Queue

Purpose

The **ENMLastEntry** function returns the most recent message in the named queue.

Format

```
USHORT ENMLastEntry(QNAME QueueID, SWITCH Lock, PMMSG Message,
                    PUSHORT MessageLength)
```

Parameters

QueueID (QNAME) - input
This is the name of a queue known to MERVA. Only queues of the API purpose group can be addressed by the API.

Lock (SWITCH) - input
The variable defines whether the message read is locked for update.

Message (PMMSG) - output
Message contains information about the message found.

MessageLength (PUSHORT) - output
Short integer variable containing the length of the found message.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

2 (ERR_SYSTEM)
An error in the MERVA instance occurred.

5 (ERR_NOT_ATTACHED)
The application is not attached to the MERVA instance.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

101 (ERR_NO_QUEUE_NAME)
The specified queue name is empty or too long.

102 (ERR_INVALID_QUEUE_NAME)
The named queue does not belong to the API purpose group, or the user has no right to use the named queue.

107 (ERR_NOT_SWITCH)
A value other than ON or OFF has been passed in a variable that has a SWITCH data type.

204 (ERR_MSG_INUSE)
The message is locked by the application, or a created message already exists in the message space.

302 (ERR_MSG_NOT_FOUND)
No matching message could be found.

Processing

With this call, the messages in a queue are sorted according to the time at which they entered the queue. With the **ENM*xxx*Entry** calls, the system maintains a position pointer for each queue. An application can switch between queues and resume at the point from where it switched to the other queue.

If the message is read with the lock set to **ON**, no other program can change the message. Use **ENMFree** to unlock the message.

If you call **ENMLastEntry** with the lock set to **ON**, the latest unlocked message in the queue is returned.

Example

The following example shows you how to use the **ENMLastEntry** call to read the last message in the API_OUT queue and display its MRN. It also attempts to read the next message. This should result in the return code ERR_MSG_NOT_FOUND (302).

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;
   MMSG   msg;
   USHORT usLen;
   FIELD  Field;
   PFIELD pField = &Field;

   rc = ENMSetAppl("PGM1");
   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      /* we do not lock the message */
      rc = ENMLastEntry( "API_OUT", OFF, &msg, &usLen );
      if( rc == NO_ERROR )
      {
         rc = ENMReadField( FLD_MRN, (PPFIELD)&pField );
         if( rc == NO_ERROR )
         {
            printf( "MRN of last message in queue: %s\n", pField );
         }
         else
         {
            printf("Error in ENMReadField, rc = %d\n", rc );
         }
         rc = ENMNextEntry( "API_OUT", OFF, &msg, &usLen );
         printf("Try reading next,return code %d (should be 302)\n", rc );
      }
      else
      {
         printf( "Error in ENMLastEntry, rc = %d\n", rc );
      }
      /* now do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
         printf( "Error in detach %d\n", rc );
      }
      else
```

```
        {
           printf("Program detached...\n");
        }
     }
     else
     {
        printf( "Error attaching to MERVA, return code %d\n", rc );
     }
  }
```

ENMNextEntry—Read Next Message in Queue

Purpose

The **ENMNextEntry** function returns the next message from the current position in the queue of messages sorted by time. If the application accesses the queue for the first time, the function returns the oldest message.

Format

```
USHORT ENMNextEntry(QNAME QueueID, SWITCH Lock, PMMSG Message,
                    PUSHORT MessageLength)
```

Parameters

QueueID (QNAME) - input
This is the name of a queue known to MERVA. Only queues of the API purpose group can be addressed by the API.

Lock (SWITCH) - input
This variable defines whether the message read should be locked against update.

Message (PMMSG) - output
Message containing the information retrieved from the message found.

MessageLength (PUSHORT) - output
Short integer variable containing the length of the message found.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

2 (ERR_SYSTEM)
An error in the MERVA instance occurred.

5 (ERR_NOT_ATTACHED)
The application is not attached to the MERVA instance.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

101 (ERR_NO_QUEUE_NAME)
The specified queue name is empty or too long.

102 (ERR_INVALID_QUEUE_NAME)
The named queue does not belong to the API purpose group, or the user has no right to use the named queue.

107 (ERR_NOT_SWITCH)
A value other than ON or OFF has been passed to a variable that has a SWITCH data type.

204 (ERR_MSG_INUSE)
The message is locked by the application, or a created message already exists in the message space.

302 (ERR_MSG_NOT_FOUND)
No matching message could be found.

Processing

With this call, the messages of a queue are sorted according to the time at which they entered the queue. With the **ENMxxxEntry** calls, the system maintains a position pointer for each queue. An application can switch between queues and resume at the point from where it switched to the other queue.

If the **ENMNextEntry** call is used with the lock set to **ON**, then the next unlocked message will be returned.

If the message is read with the lock set to **ON**, no other program can change the message. Use **ENMFree** to unlock the message.

Example

The following example shows you how to use the **ENMNextEntry** call to locate and print all the MRNs found in an API queue.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;
   MMSG   msg;
   USHORT usLen;
   FIELD  Field;
   PFIELD pField = &Field;

   rc = ENMSetAppl("PGM1");
   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      /* we do not lock the message */
      do
      {
         rc = ENMNextEntry( "API_OUT", OFF, &msg, &usLen );
         if( rc == NO_ERROR )
         {
            ENMReadField( FLD_MRN, (PPFIELD)&pField );
            printf( "The MRN field contains: %s\n", pField );
         }
      } while( rc == NO_ERROR );
      if( rc != ERR_MSG_NOT_FOUND )
      {
         printf( "Error in ENMNextEntry, rc = %d\n", rc );
      }
      /* now do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
         printf( "Error in detach %d\n", rc );
      }
      else
      {
         printf("Program detached...\n");
      }
   }
   else
   {
      printf( "Error attaching to MERVA, return code %d\n", rc );
   }
}
```

ENMOpenSem—Open a Semaphore

Purpose

The **ENMOpenSem** function opens an existing semaphore created by another process with an **ENMCreateSem** call.

Format

```
USHORT ENMOpenSem(PULONG pSemHandle, PCHAR pszSemName)
```

Parameters

pSemHandle (PULONG) - output
Address of the handle of the opened semaphore.

pszSemName (PCHAR) - input
Pointer to a null-terminated string containing the name of the semaphore to be opened.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not been started.

6 (ERR_OUT_OF_MEMORY)
The system does not have enough memory to complete the function.

| **7** (ERR_WRITE_TRACE)
| An error ocurred while writing to the trace file.

100 (ERR_TOO_MANY_SEMAPHORES)
The count of semaphores available on the system exceeds the maximum value.

123 (ERR_INVALID_SEMAPHORE_NAME)
The semaphore name is an invalid Windows NT file name.

187 (ERR_SEMAPHORE_NOT_EXISTS)
The semaphore to be opened does not exist.

255 (ERR_SEMAPHORE_FAILED)
The semaphore call failed with an internal error.

Processing

The function opens an existing semaphore and increments an internal counter by 1. On the opposite, the **ENMCloseSem** call decrements this counter by 1. If the counter reaches 0, the semaphore is automatically removed from the system. As counterpart to each **ENMCreateSem** and **ENMOpenSem** you must code an **ENMCloseSem** so that no semaphores remain in the system after the application has ended.

Example

The following example shows you how to use an **ENMOpenSem** call to get the handle of an existing semaphore.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

#define  TRIGGER  "apisem"
main()
{
   USHORT lRc = 0;
   ULONG  SemHandle;

   lRc = ENMOpenSem(&SemHandle, TRIGGER);
   if( lRc == NO_ERROR )
   {
      printf("Semaphore successfully opened by MERVA\n");

      /* ... do some processing here ... */

      /* for example, clear this semaphore to        */
      /* signal another process that your processing */
      /* is done.                                    */
      lRc = ENMClearSem( SemHandle );

      lRc = ENMCloseSem( SemHandle );
      if( lRc == NO_ERROR )
      {
         printf("Semaphore successfully closed\n");
      }
      else
      {
        printf( "Error in ENMCloseSem, rc = %d\n", lRc );
      }
   }
   else
   {
      if ( lRc == ERR_SEMAPHORE_NOT_EXISTS )
      {
         printf("Semaphore does not exist.\n");
      }
      else
      {
         printf( "Error opening a semaphore by MERVA, return code %d\n", lRc );
      }

   }
}
```

ENMPreviousEntry—Read Previous Queue Message

Purpose

The **ENMPreviousEntry** function returns the previous message from the current position in the queue of messages sorted by time. If the application accesses the queue for the first time, the function returns the latest message.

Format

```
USHORT ENMPreviousEntry(QNAME QueueID, SWITCH Lock, PMMSG Message,
                        PUSHORT MessageLength)
```

Parameters

QueueID (QNAME) - input
: This is the name of a queue known to MERVA. Only queues of the API purpose group can be addressed by the API.

Lock (SWITCH) - input
: The variable defines whether the message read should be locked against update.

Message (PMMSG) - output
: Message containing the information retrieved from the message found.

MessageLength (PUSHORT) - output
: Short integer variable containing the length of the message found.

rc (USHORT) - return
: Values are:

  0 (NO_ERROR)
  : The function completed successfully.

  1 (ERR_SYSTEM_NOT_UP)
  : The MERVA instance has not yet been started.

  2 (ERR_SYSTEM)
  : An error in the MERVA instance occurred.

  5 (ERR_NOT_ATTACHED)
  : The application is not attached to the MERVA instance.

  7 (ERR_WRITE_TRACE)
  : An error occurred while writing to the trace file.

  101 (ERR_NO_QUEUE_NAME)
  : The specified queue name is empty or too long.

  102 (ERR_INVALID_QUEUE_NAME)
  : The named queue does not belong to the API purpose group, or the user has no right to use the named queue.

  107 (ERR_NOT_SWITCH)
  : A value other than ON or OFF has been passed to a variable that has a SWITCH data type.

  204 (ERR_MSG_INUSE)
  : The message is locked by the application, or a created message already exists in the message space.

  302 (ERR_MSG_NOT_FOUND)
  : No matching message could be found.

Processing

With this call the messages of a queue are sorted according to the time at which they entered the queue. The system keeps a position pointer for each queue with the **ENMxxxEntry** calls. An application can switch between several queues and resume at the point where it switched to another queue.

If the message is read with the lock set to **ON**, no other program can change the message. Use **ENMFree** to unlock the message.

If you call **ENMPreviousEntry** with the lock set to **ON**, the previous unlocked message in the queue is returned.

Example

The following example shows you how to:

- Read the first message in the API_OUT queue.
- Display the MRN of this message.
- Read the next message.
- Print the MRN.
- Use the ENMPreviousEntry call to read the first message again.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;
   MMSG   msg;
   USHORT usLen;
   FIELD  Field;
   PFIELD pField = &Field;

   rc = ENMSetAppl("PGM1");
   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      /* we do not lock the message */
      rc = ENMFirstEntry( "API_OUT", OFF, &msg, &usLen );
      if( rc == NO_ERROR )
      {
         rc = ENMReadField( FLD_MRN, (PPFIELD)&pField );
         printf( "The MRN of the first is: %s\n", pField );
         rc = ENMNextEntry( "API_OUT", OFF, &msg, &usLen );
         if( rc == NO_ERROR )
         {
            rc = ENMReadField( FLD_MRN, (PPFIELD)&pField );
            printf( "The MRN of the next is:  %s\n", pField );

            rc = ENMPreviousEntry( "API_OUT", OFF, &msg, &usLen );

            if( rc == NO_ERROR )
            {
               rc = ENMReadField( FLD_MRN, (PPFIELD)&pField );
               printf( "The MRN of the prev. is: %s\n", pField );
            }
         }
      }
      else
      {
         printf( "Error in ENMFirstEntry, rc = %d\n", rc );
      }
      /* now do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
         printf( "Error in detach %d\n", rc );
      }
      else
      {
         printf("Program detached...\n");
      }
   }
   else
   {
      printf( "Error attaching to MERVA, return code %d\n", rc );
   }
}
```

ENMPut—Return Message to Queue and Unlock

Purpose

The **ENMPut** function returns a message to the queue from where it was retrieved. It stays in the same position in the queue. The message is unlocked after the operation.

Format

```
USHORT ENMPut()
```

Parameters

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

2 (ERR_SYSTEM)
An error in the MERVA instance occurred.

5 (ERR_NOT_ATTACHED)
The application is not attached to the MERVA instance.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

115 (ERR_SWIFT_HEAD)
The header of the message does not match the rules for SWIFT headers.

The level of checking done for the header of a SWIFT message is described in “Appendix B. Message Header Checking” on page 149.

116 (ERR_TELEX_HEAD)
The header of the message does not match the rules for telex headers.

The checked fields of the telex header are described in “Telex Header (TX_HEADER)” on page 11.

117 (ERR_NETWORK)
There is no destination network specified for the network.

201 (ERR_NO_MSG_LOCKED)
No message has been locked by the application.

302 (ERR_MSG_NOT_FOUND)
The message could not be found in the queue.

Processing

The message is checked before it is added to a queue. Depending on the value of **FLD_MSGNET**, the message is checked for conformance to S.W.I.F.T or Telex rules.

If processing completes successfully, the message is unlocked. In case of error, use **ENMFree** to unlock the message.

Restrictions

A retrieve call with the lock set to **ON** must have been previously issued, because an application can only return a message it has previously locked.

For detailed information about how to specify message lengths, refer to "Processing" on page 113.

Example

The following example shows you how to use an **ENMPut** call to modify an existing message in a queue without further routing. The message is retrieved from the API_OUT queue and the **MSG_ACK** field is filled with the string **API ACK Message**..

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"


main()
{
   USHORT rc = 0;
   MMSG   msg;
   USHORT usLen;
   FIELD  Field;

   rc = ENMSetAppl("PGM1");
   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      /* we lock the message */
      rc = ENMNextEntry( "API_OUT", ON, &msg, &usLen );
      if( rc == NO_ERROR )
      {
         strcpy( Field.msgack, "API ACK Message" );
         rc = ENMWriteField( FLD_MSGACK, &Field );
         if( rc != NO_ERROR )
         {
            printf("Error in ENMWritefield %d\n",rc );
         }
         else
         {
           /* now put it back */
           rc = ENMPut();
           if( rc == NO_ERROR)
           {
            printf("Message modified and put back to queue\n");
            printf("Use the Retrieve Message by Queue facility\n");
            printf("to view MSGACK field in message\n");
           }
         }
         if ( rc != NO_ERROR )
         {
            rc = ENMFree();
         }
      }
      /* now do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
          printf( "Error in detach %d\n", rc );
      }
      else
```

```
        {
            printf("Program detached...\n");
        }
    }
    else
    {
        printf( "Error attaching to MERVA, return code %d\n", rc );
    }
}
```

ENMQueryQueue—Get Status of Queue

Purpose

The **ENMQueryQueue** function returns the number of messages in the named queue at the time of the call. The maximum count of messages is 65535. Even if the named queue contains more than 65535 messages, the number 65535 is returned. To get the correct number of messages, use **ENMQueryQueueEx**.

Format

```
USHORT ENMQueryQueue(QNAME QueueID, PUSHORT MessageCount)
```

Parameters

QueueID (QNAME) - input
The QueueID is the name of the queue that the application is enquiring about.

MessageCount (PUSHORT) - output
Short integer variable containing the number of messages in the queried queue at the time of the call.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

2 (ERR_SYSTEM)
An error in the MERVA instance occurred.

5 (ERR_NOT_ATTACHED)
The application is not attached to the MERVA instance.

6 (ERR_OUT_OF_MEMORY)
The API could not allocate the memory required for processing.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

101 (ERR_NO_QUEUE_NAME)
The specified queue name is empty or too long.

102 (ERR_INVALID_QUEUE_NAME)
The specified queue name is not valid (does not exist).

Processing

The **ENMQueryQueue** function can be used to determine the number of messages in any MERVA queue and is not restricted to API queues. The **ENMQueryQueue** call is independent of any other message function calls, such as **ENMCreate**, **ENMWriteField**, or **ENMFirstEntry**.

Restrictions

The number is only valid for the time of the call because more than one application can work on a queue.

Example

The following program uses the **ENMQueryQueue** call to determine the number of messages in the API_OUT queue.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;
   USHORT usMessagecount;
   QNAME  qnMyQueue;

   rc = ENMSetAppl("PGM1");
   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      strcpy( qnMyQueue, "API_OUT" );

      rc = ENMQueryQueue( qnMyQueue, &usMessagecount );

      if( rc == NO_ERROR )
      {
          printf("There are %d messages in Queue %s\n",
                  usMessagecount, qnMyQueue );
      }
      else
      {
          printf( "Error in ENMQueryQueue, return code %d\n", rc );
      }
      /* now do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
          printf( "Error in detach %d\n", rc );
      }
   }
   else
   {
      printf( "Error attaching to MERVA, return code %d\n", rc );
   }
}
```

ENMQueryQueueEx—Get Status of Queue

Purpose

The **ENMQueryQueueEx** function returns the number of messages in the named queue at the time of the call. The maximum count of messages is 2 147 483 647.

Format

```
USHORT ENMQueryQueueEx(QNAME QueueID, PLONG MessageCount)
```

Parameters

QueueID (QNAME) - input
The QueueID is the name of the queue that the application is enquiring about.

MessageCount (PLONG) - output
Short integer variable containing the number of messages in the queried queue at the time of the call.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

2 (ERR_SYSTEM)
An error in the MERVA instance occurred.

5 (ERR_NOT_ATTACHED)
The application is not attached to the MERVA instance.

6 (ERR_OUT_OF_MEMORY)
The API could not allocate the memory required for processing.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

101 (ERR_NO_QUEUE_NAME)
The specified queue name is empty or too long.

102 (ERR_INVALID_QUEUE_NAME)
The specified queue name is not valid (does not exist).

Processing

The **ENMQueryQueueEx** function can be used to determine the number of messages in any MERVA queue and is not restricted to API queues. The **ENMQueryQueueEx** call is independent of any other message function calls, such as **ENMCreate**, **ENMWriteField**, or **ENMFirstEntry**.

Restrictions

The number is only valid for the time of the call because more than one application can work on a queue.

Example

The following program uses the **ENMQueryQueueEx** call to determine the number of messages in the API_OUT queue.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;
   LONG  lMessagecount;
   QNAME qnMyQueue;

   rc = ENMSetAppl("PGM1");
   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      strcpy( qnMyQueue, "API_OUT" );

      rc = ENMQueryQueueEx( qnMyQueue, &lMessagecount );

      if( rc == NO_ERROR )
      {
          printf("There are %d messages in Queue %s\n",
                 lMessagecount, qnMyQueue );
      }
      else
      {
          printf( "Error in ENMQueryQueueEx, return code %d\n", rc );
      }
      /* now do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
          printf( "Error in detach %d\n", rc );
      }
   }
   else
   {
      printf( "Error attaching to MERVA, return code %d\n", rc );
   }
}
```

ENMReadField—Read Field Associated with Message

Purpose

The **ENMReadField** function returns information associated with a message.

Format

```
USHORT ENMReadField(FIELDTYPE FieldType, PPFIELD Field)
```

Parameters

FieldType (FIELDTYPE) - input
: The field type contains the name of the field the application wants to read.

Field (PPFIELD) - output
: The field union contains the contents of the field the application wants to read.

rc (USHORT) - return
: Values are:

0 (NO_ERROR)
: The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
: The MERVA instance has not yet been started.

2 (ERR_SYSTEM)
: An error in the MERVA instance occurred.

5 (ERR_NOT_ATTACHED)
: The application is not attached to the MERVA instance.

7 (ERR_WRITE_TRACE)
: An error occurred while writing to the trace file.

112 (ERR_INVALID_FIELDTYPE)
: The specified field type is not in the range of the FIELDTYPE enumerated data type.

203 (ERR_NO_MSG)
: No message has been retrieved by the application.

Restrictions

The information can be read only if a message has been previously retrieved because the fields are associated with a message. After the return of a message to MERVA, information can no longer be read.

Example

The following example shows you how to use an **ENMCreate** call to create a new message, and then use the **ENMReadField** function to display the MRN field of this newly created message.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;
   FIELD  Field;
   PFIELD pField = &Field;
   MMSG   msgBuffer;

   rc = ENMSetAppl("PGM1");
   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      rc = ENMCreate( &msgBuffer );
      if( rc == NO_ERROR )
      {
         printf("New Message created\n");
         rc = ENMReadField( FLD_MRN, (PPFIELD)&pField );
         if( rc == NO_ERROR )
         {
             printf( "The MRN field contains: %s\n", pField );
         }
         else
         {
             printf("Error in read field, rc = %d\n", rc );
         }
         rc = ENMClear();
      }
      else
      {
         printf( "Error in ENMCreate, rc = %d\n", rc );
      }
      /* now do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
          printf( "Error in detach %d\n", rc );
      }
      else
      {
          printf("Program detached...\n");
      }
   }
   else
   {
      printf( "Error attaching to MERVA, return code %d\n", rc );
   }
}
```

ENMRestartRAPI—Reconnect Remote API Program to Another MERVA System

Purpose

This function reconnects to the remote MERVA API server. If **ENMStartRAPI** is not called before this function, this function has the same effect as **ENMStartRAPI**. After this function is called, the program must end with the call **ENMEndRAPI**.

The resynchronization is provided for the following API calls:

- ENMAdd
- ENMDelete
- ENMPut
- ENMRouteAdd
- ENMRoutePut

For details refer to the section of the *MERVA Connection/NT* manual that describes resynchronization.

To ensure compatibility between MERVA API programs and MERVA Connection/NT programs, this function is also provided for the local MERVA system, however for the local system it internally calls **ENMSetAppl**.

Format

```
USHORT ENMRestartRAPI(PUCHAR pucApplicationName)
```

Parameters

pucApplicationName (PUCHAR) - input
Pointer to a null-terminated string of up to eight characters. This name is registered by the Remote MERVA API Server if using MERVA Connection/NT.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

2 (ERR_SYSTEM)
If ENMStartRAPI is called in:

The remote MERVA system
An internal error occurred. The API receives further information by calling the function **ENMGetReason** (see "ENMGetReason—Get Reason Code for Internal Error" on page 61).

The local MERVA system
An error occurred while calling **ENMSetAppl**.

Example

The following example shows you how to use an **ENMRestartRAPI** call.

```
#include "enmcapi.h" /* or include "enmrapi.h" for MERVA Connection/NT */

               ...
               ENMSetProfile("enm6ri.prf");
               ENMStartRAPI("SAMPLE");
               /* ... do some API calls */
               /* ... there was an error */
               rc = ENMRestartRAPI("SAMPLE");
               /* ... do some API calls */
               ENMEndRAPI();
               ...
```

ENMRouteAdd—Route and Add a Created Message

Purpose

The **ENMRouteAdd** function adds the created message to the destination queue defined by the routing conditions for the supplied source queue.

Format

```
USHORT ENMRouteAdd(QNAME QueueID)
```

Parameters

QueueID (QNAME) - input
This is the name of a queue known to MERVA. Only queues of the API purpose group can be addressed by the API.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

2 (ERR_SYSTEM)
An error in the MERVA instance occurred.

5 (ERR_NOT_ATTACHED)
The application is not attached to the MERVA instance.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

8 (ERR_ROUTING)
The router could not identify a destination for the message.

101 (ERR_NO_QUEUE_NAME)
The specified queue name is empty or too long.

102 (ERR_INVALID_QUEUE_NAME)
The named queue does not belong to the API purpose group, or the user has no right to use the named queue.

115 (ERR_SWIFT_HEAD)
The header of the message does not match the rules for SWIFT headers.

The level of checking done for the header of a SWIFT message is described in "Appendix B. Message Header Checking" on page 149.

116 (ERR_TELEX_HEAD)
The header of the message does not match the rules for telex headers.

The checked fields of the telex header are described in "Telex Header (TX_HEADER)" on page 11.

117 (ERR_NETWORK)
There is no destination network specified for the network.

202 (ERR_NO_MSG_CREATED)
No message has been previously created by the application.

Processing

Depending on the value of **FLD_MSGNET**, the message is checked for correctness before it is added to a queue. Checking is not performed when **FLD_MSGNET** contains **NET_OWN**. For more information about message checking, refer to "Appendix B. Message Header Checking" on page 149.

A destination queue is calculated from the routing conditions.

Restrictions

The **ENMCreate** call must have been issued previously because the application can only add a message it has previously created. The message can be added only once.

For detailed information about how to specify message lengths, refer to "Processing" on page 113.

Example

The following example shows you how to use **ENMRouteAdd** to add a new message to an API queue, and route the message from this queue to other queues defined in the routing set up for this queue.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;
   CHAR   msgTxt[ 200 ];
   MMSG   msgBuffer;
   FIELD  fldAssociated;

   rc = ENMSetAppl("PGM1");
   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      if( ENMCreate( &msg ) == NO_ERROR )
      {
         fldAssociated.msgnet = NET_SWIFT;
         rc = ENMWriteField( FLD_MSGNET, &fldAssociated );
         if( rc == NO_ERROR )
         {
            /* create message example type 399 */
            strcpy( msgTxt,
                    "{1:F01VNDPBET2AXXX0000000299}{2:I399VNDPBET2AXXXN}"
                    "{3:{108:399-14}}{4:\r\n:20:399-14\r\n:79:REPLACE MT 101\r\n-}");
            memcpy( msg, msgTxt, strlen(msgTxt) );

            rc = ENMRouteAdd( "API_IN" );
            if( rc == NO_ERROR )
            {
               printf("Message added to Queue API_IN and routed\n");
            }
         }
         else
         {
            printf("Error in ENMWriteField, rc %d\n", rc );
         }
         rc = ENMClear();
      }
      /* now do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
          printf( "Error in detach %d\n", rc );
      }
      else
      {
          printf("Program detached...\n");
      }
   }
   else
   {
      printf( "Error attaching to MERVA, return code %d\n", rc );
   }
}
```

ENMRoutePut—Route Message to Queue

Purpose

The **ENMRoutePut** function sends a message to one or more queues of the MERVA instance. The destination queues are defined by message routing. A message from an API queue can be routed to any queue of any purpose group defined through the MERVA Customization program. The message is removed from the source queue.

Format

```
USHORT ENMRoutePut()
```

Parameters

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

2 (ERR_SYSTEM)
An error in the MERVA instance occurred.

5 (ERR_NOT_ATTACHED)
The application is not attached to the MERVA instance.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

8 (ERR_ROUTING)
The router could not identify a destination for the message.

115 (ERR_SWIFT_HEAD)
The header of the message does not match the rules for SWIFT headers.

The level of checking done for the header of a SWIFT message is described in “Appendix B. Message Header Checking” on page 149.

116 (ERR_TELEX_HEAD)
The header of the message does not match the rules for telex headers.

The checked fields of the telex header are described in “Telex Header (TX_HEADER)” on page 11.

117 (ERR_NETWORK)
There is no destination network specified for the message.

201 (ERR_NO_MSG_LOCKED)
No message has been locked by the application.

301 (ERR_MSG_LOCKED)
Another application program has also locked this message.

302 (ERR_MSG_NOT_FOUND)
The message could not be found in the queue.

Processing

The message is checked before it is added to a queue. Depending on the value of **FLD_MSGNET**, the message is checked for conformance to S.W.I.F.T or Telex rules.

If processing completes successfully, the message is unlocked. In case of error, use **ENMFree** to unlock the message.

Restrictions

A retrieve call with lock set to **ON** must have been previously issued, because an application can only return a message it has previously locked.

For detailed information about how to specify message lengths, refer to "Processing" on page 113.

Example

The following example shows you how to use the **ENMRoutePut** call to route an existing message from one queue to the next queue defined in the routing definition for this queue. When the routing is defined to be from API_OUT to SLPRINT1, the message is routed to SLPRINT1 in this example.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"


main()
{
   USHORT rc = 0;
   MMSG   msg;
   USHORT usLen;
   FIELD  Field;

   rc = ENMSetAppl("PGM1");
   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      /* get the oldest message and lock it */
      rc = ENMNextEntry( "API_OUT", ON, &msg, &usLen );
      if( rc == NO_ERROR )
      {
         /* now put it back and route it to the destination */
         /* defined by the routing that was set up          */
         rc = ENMRoutePut();
         if( rc == NO_ERROR)
         {
            printf("Message routed to destination queues\n");
         }
         else
         {
            printf("Error in ENMRoutePut %d\n",rc);
            rc = ENMFree();
         }
      }
      /* now do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
          printf( "Error in detach %d\n", rc );
      }
      else
```

```
         {
             printf("Program detached...\n");
         }
      }
      else
      {
         printf( "Error attaching to MERVA, return code %d\n", rc );
      }
   }
```

ENMSetAppl—Set Application Name

Purpose

The **ENMSetAppl** function sets the application name used to identify the application against MERVA. With this call, several programs with the same name can be attached to MERVA at the same time.

Format

```
USHORT ENMSetAppl(PUCHAR pszApplName)
```

Parameters

pszApplName (PUCHAR) - input
The name of the application up to 8 characters long.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

20 (ERR_APPLICATION_SET)
The application identifier is already set.

21 (ERR_WRONG_LENGTH)
The application identifier is too long.

106 (ERR_INVALID_ID)
The application name is not valid; it does begin with one of the approved prefixes.

Processing

The **ENMSetAppl** function must be called before the **ENMAttach** function. If it is omitted, the program name of the application becomes the default value. The application name must be unique. For example, only one program with the application ID PGM1 can be attached to MERVA at any one time. The **ENMAttach** function checks whether the application name is unique.

Restrictions

Each API program that communicates with MERVA is identified by its name set in the API call **ENMSetAppl()**. The name must be unique. If this call is missing, the program name is used as identifier. For example, only one program with the application ID PGM1 can be attached to MERVA at one time. To prevent a conflict with names used by MERVA, do not specify the application name with one of the following 3-character prefixes:

- **enm**
- **ENM**
- **enn**
- **ENN**
- **eka**
- **EKA**
- **enl**
- **ENL**

Note: Before you can use a user ID for an API program, you have to define it in the MERVA Users program. Even if the user ID has the access right **API - without password**, you have to define a valid password for this user ID.

Example

The following example shows you how to use the **ENMSetAppl** call.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;

   rc = ENMSetAppl("PGM1");

   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");

      /* now do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
          printf( "Error in detach %d\n", rc );
      }
      else
      {
          printf("Program detached...\n");
      }
   }
   else
   {
      printf( "Error attaching to MERVA, return code %d\n", rc );
   }
}
```

ENMSetProfile—Set a Connection Profile

Purpose

This function sets a profile for a connection to a remote MERVA system. To ensure compatibility between MERVA API programs and MERVA Connection/NT programs, this function is also provided for the local MERVA system, however for the local system it is only a dummy function that always returns 0 and otherwise does nothing.

For a description of the format and contents fo the profile, refer to the section of the *MERVA Connection/NT* manual that describes customization.

Format

```
ENMSetProfile(PUCHAR pucProfileName)
```

Parameters

pucProfileName (PUCHAR) - input
Pointer to a null-terminated string with a maximum length of 80 characters. This is the full path name of the profile.

Example

The following example shows you how to use an **ENMSetProfile** call.

```
#include "enmcapi.h" /* or include "enmrapi.h" for MERVA Connection/NT */

             ...
             ENMSetProfile("enm6ri.prf");
             ENMStartRAPI("SAMPLE");
             /* ... do some API calls */
             ENMEndRAPI();
             ...
```

ENMSetSecurity—Set Conversation Security Information

Purpose

A MERVA application program can use this function to provide conversation security information for client authorization at the remote MERVA system. To ensure compatibility between MERVA API programs and MERVA Connection/NT programs, this function is also provided for the local MERVA system, however for the local system it is only a dummy function that always returns 0 and otherwise does nothing.

Notes:

1. If this function is used, it must be called before **ENMStartRAPI()** or **ENMRestartRAPI**.
2. Regardless of whether this function is called before or after **ENMSetProfile()**, the conversation security information this function provides takes precedence over that provided in the profile established by **ENMSetProfile()**. However, if this function does not provide security information, the parameters of the profile are used.
3. The conversation security set by this function does not affect the MERVA user information set by **ENMAttach()**.

Format

```
USHORT ENMSetSecurity(PUCHAR pucUserID,
                      PUCHAR pucPassword)
```

Parameters

pucUserID (PUCHAR) - input
Pointer to a null-terminated string of up to eight characters containing the user ID.

pucPassword (PUCHAR) - input
Pointer to a null-terminated string of up to eight characters containing the user password.

rc (USHORT) - output
Values are:

0 (NO_ERROR)
The function completed successfully.

2 (ERR_SYSTEM)
An internal error has occurred. For further information refer to the *MERVA USE & Branch for Windows NT User's Guide*.

Example

The following example shows you how to use an **ENMSetSecurity** call.

```
#include "enmcapi.h" /* or include "enmrapi.h" for MERVA Connection/NT */
               ...
               ENMSetProfile("enm6ri.prf");
               ENMSetSecurity("SAMPLE","SAMPLE1");
               ENMStartRAPI("SAMPLE");
               /* ... do some API calls */
               ENMEndRAPI();
               ...
```

ENMSetTestEnv—Set Test Environment

Purpose

For specific sections of an application program, a MERVA application program can use this function to activate or inactivate the test environment of a remote MERVA API client. This can be done as often as required.

To ensure compatibility between MERVA API programs and MERVA Connection/NT programs, this function is also provided for the local MERVA system, however for the local system it is only a dummy function that always returns 0 and otherwise does nothing.

Format

```
ENMSetTestEnv(UCHAR ucTestEnvIndicator)
```

Parameters

ucTestEnvIndicator (UCHAR) - input
Function parameter **'1'** activates the test environment, function parameter **'0'** inactivates the test environment.

Example

The following example shows you how to use an **ENMSetTestEnv** call.

```
#include "enmcapi.h" /* or include "enmrapi.h" for MERVA Connection/NT */
              ...
              ENMSetProfile("enm6ri.prf");
              ENMStartRAPI("SAMPLE");
              ENMSetTestEnv('1');
              /* ... do some API calls */
              ENMEndRAPI();
              ...
```

ENMSetSem—Set a Semaphore

Purpose

The **ENMSetSem** function sets a semaphore unconditionally. In MERVA, the semaphore can be cleared by raising an alarm.

Format

```
USHORT ENMSetSem(ULONG SemHandle)
```

Parameters

SemHandle (ULONG) - input
Semaphore handle created by **ENMCreateSem** or **ENMOpenSem**.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

| **1** (ERR_SYSTEM_NOT_UP)
| The MERVA instance has not been started.

6 (ERR_OUT_OF_MEMORY)
The system does not have enough memory to complete the function.

| **7** (ERR_WRITE_TRACE)
| An error ocurred while writing to the trace file.

255 (ERR_SEMAPHORE_FAILED)
The semaphore call failed with an internal error.

Example

The following example shows you how to use an **ENMSetSem** call to block the running process until an alarm is raised by MERVA.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

#define  TRIGGER  "apisem"
main()
{
   USHORT  usIndex = 0;
   USHORT  lRc = 0;
   ULONG   SemHandle;

   lRc = ENMSetAppl("PGM1");

   lRc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( lRc == NO_ERROR )
   {

      lRc = ENMCreateSem(&SemHandle, TRIGGER);
      if( lRc == NO_ERROR )
      {
         printf("Semaphore successfully created by MERVA\n");

         lRc = ENMSetSem(SemHandle);

         /* Wait indefinitely until the semaphore will be cleared. */
         /* For example, if a message reaches an API queue and the */
         /* defined alarm will be raised.                          */
         lRc = ENMWaitSemList(&usIndex, -1, SemHandle, 0);

         /* ... do some processing with the received message ... */

         lRc = ENMCloseSem( SemHandle );
         if( lRc == NO_ERROR )
         {
            printf("Semaphore successfully deleted\n");
         }
         else
         {
           printf( "Error in ENMCloseSem, rc = %d\n", lRc );
         }
      }
      else
      {
         printf( "Error creating a semaphore by MERVA, return code %d\n", lRc );
      }
      lRc = ENMDetach();
   }
   else
   {
      printf( "Error attaching to MERVA, return code %d\n", lRc );
   }
}
```

ENMStartRAPI—Establish Connection to Another MERVA System

Purpose

In MERVA Connection/NT, this call establishes the connection to the Remote MERVA API Server and initializes internal buffers for communication. After this call, the program must end with the call **ENMEndRAPI**. To ensure compatibility between MERVA API programs and MERVA Connection/NT programs, this function is also provided for the local MERVA system, however for the local system it internally calls **ENMSetAppl**.

Format

```
USHORT ENMStartRAPI(PUCHAR pucApplicationName)
```

Parameters

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

2 (ERR_SYSTEM)
If ENMStartRAPI is called in:

The remote MERVA system
An internal error occurred. The API receives further information by calling the function **ENMGetReason** (see “ENMGetReason—Get Reason Code for Internal Error” on page 61).

The local MERVA system
An error occurred while calling **ENMSetAppl**.

Example

The following example shows you how to use an **ENMStartRAPI** call.

```
#include "enmcapi.h" /* or include "enmrapi.h" for MERVA Connection/NT */

            ...
            ENMSetProfile("enm6ri.prf");
            ENMStartRAPI("SAMPLE");
            /* ... do some API calls */
            ENMEndRAPI();
            ...
```

ENMTrace—Turn API Trace ON or OFF

Purpose

The **ENMTrace** function turns the API trace on or off.

Format

```
USHORT ENMTrace(SWITCH Status)
```

Parameters

Status (SWITCH) - input
: If the variable contains the value **ON**, the API trace is turned on. If it contains the value **OFF**, it is turned off.

rc (USHORT) - return
: Values are:

  0 (NO_ERROR)
  The function completed successfully.

  1 (ERR_SYSTEM_NOT_UP)
  The MERVA instance is not started.

  7 (ERR_WRITE_TRACE)
  An error occurred while writing to the trace file.

  107 (ERR_NOT_SWITCH)
  A value other than ON or OFF has been passed to a variable that has a SWITCH data type.

Processing

With trace set to **ON**, the API puts an entry in the API trace file of MERVA for every call to one of its functions. In addition to the application and function name, the entry contains the values of all parameters passed to the function.

Example

The following example shows you how to use the **ENMTrace** call to write trace information from an API program to the MERVA API trace file.

Note: ATTACH or DETACH functions are not necessary for using the trace functions.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;

   rc = ENMTrace( ON );
   if( rc == NO_ERROR )
   {
      rc = ENMWriteTrace( "This is trace information from the API program" );
      if( rc == NO_ERROR )
      {
          printf("Trace written\n");
      }
      else
```

```
         {
             printf("Error writing trace, rc = %d\n", rc );
         }
      }
      else
      {
         printf("Error setting trace to ON, rc = %d\n",rc);
      }
   }
```

ENMWaitSemList—Wait for a List of Semaphores

Purpose

The **ENMWaitSemList** function blocks the current process until one of the specified semaphores is cleared. It allows the API program to wait for a list of up to 16 semaphores and up to 16 different MERVA alarms.

Format

```
USHORT ENMWaitSemList(PUSHORT pusIndex, LONG lTimeout, ULONG SemHandle, ...)
```

Parameters

pusIndex (PUSHORT) - output
Pointer to the index value that tells which of the semaphores is cleared (0 .. 15).

lTimeout (LONG) - input
Time to be waited until the function call returns.

Code	**Meaning**
-1	Wait indefinitely for a semaphore to be cleared.
0	Return immediately.
>0	Wait the indicated number of milliseconds for a semaphore to be cleared before resuming execution.

SemHandle (ULONG) - input
Semaphore handle, created by **ENMCreateSem** or **ENMOpenSem**.

... (ULONG) - input
Up to 15 further semaphore handles. The list of SemHandle parameters must be terminated by the value 0.

rc (SHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

| **1** (ERR_SYSTEM_NOT_UP)
| The MERVA instance has not been started.

6 (ERR_OUT_OF_MEMORY)
The system does not have enough memory to complete the function.

| **7** (ERR_WRITE_TRACE)
| An error ocurred while writing to the trace file.

11 (ERR_PROCESS_EXCEEDED)
The total number of processes running concurrently in the system is exceeded. The system cannot create a further process.

36 (ERR_SEMAPHORE_REMOVED)
One of the semaphores is removed from the system.

121 (ERR_SEMAPHORE_TIMEOUT)
The waiting time is passed.

255 (ERR_SEMAPHORE_FAILED)
The semaphore call failed with an internal error.

Example

The following example shows you how to use an **ENMWaitSemList** call to block the running process until an alarm is raised by MERVA, or a stop process is executed.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

#define  TRIGGER  "apisem"
#define  STOP     "apistop"
main()
{
   USHORT  usIndex = 0;
   USHORT  lRc = 0;
   ULONG   SemHandleStop;
   ULONG   SemHandle;

   lRc = ENMSetAppl("PGM1");
   lRc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( lRc == NO_ERROR )
   {

      lRc = ENMCreateSem(&SemHandleStop, STOP);
      if( lRc == NO_ERROR )
      {
         lRc = ENMCreateSem(&SemHandle, TRIGGER);
         if( lRc == NO_ERROR )
         {
            lRc = ENMSetSem(SemHandleStop);
            lRc = ENMSetSem(SemHandle);

            /* Wait indefinitely until one semaphore will be cleared.*/
            /* For example, if a message reaches an API queue        */
            /* and the defined alarm will be raised                  */
            /* or a stop request is performed.                       */

            lRc = ENMWaitSemList(&usIndex, -1, SemHandle, SemHandleStop, (ULONG) 0);

            if ( lRc == NO_ERROR )
            {
               if ( usIndex == 0 )
               {
                   /* ... do some processing with the received message ... */
               }
               if ( usIndex == 1 )
               {
                   printf( "Process stopped without message processing\n" );
               }
            }

            lRc = ENMCloseSem( SemHandle );
            if( lRc != NO_ERROR )
            {
              printf( "Error in ENMCloseSem, rc = %d\n", lRc );
            }
         }
         else
         {
            printf( "Error creating trigger semaphore by MERVA, return code %d\n", lRc );
         }
         lRc = ENMCloseSem( SemHandleStop );
         if( lRc != NO_ERROR )
         {
            printf( "Error in ENMCloseSem, rc = %d\n", lRc );
         }
      }
      else
      {
         printf( "Error creating stop semaphore by MERVA, return code %d\n", lRc );
      }
      lRc = ENMDetach();
   }
   else
   {
      printf( "Error attaching to MERVA, return code %d\n", lRc );
   }
}
```

ENMWhereIs—Query Location of Message

Purpose

The **ENMWhereIs** function returns the purpose group where the message with a specified key resides. An application can, for example, check whether a message is already sent to the SWIFT network or whether it waits to be sent.

Format

```
USHORT ENMWhereIs(KEYTYPE KeyType, KEY Key, PGROUP Group)
```

Parameters

KeyType (KEYTYPE) - input
The variable defines the type of key to be searched for. The supported types are defined in the KEYTYPE enumerated data type.

Key (KEY) - input
The key is the identification of a message in a queue. If a key has a special format, the API checks for that format.

Group (PGROUP) - output
Name of the group in which the message is queued.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

2 (ERR_SYSTEM)
An error in the MERVA instance occurred.

5 (ERR_NOT_ATTACHED)
The application is not attached to the MERVA instance.

6 (ERR_OUT_OF_MEMORY)
The API could not allocate the memory required for processing.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

103 (ERR_NO_KEY)
The specified key is empty or too long.

104 (ERR_INVALID_MRN)
The specified MRN has an invalid format.

105 (ERR_INVALID_ISN)
The specified ISN has an invalid format.

108 (ERR_INVALID_KEYTYPE)
The specified key type is not in the range of the KEYTYPE enumerated data type.

302 (ERR_MSG_NOT_FOUND)
No matching message could be found.

Processing

If routing is arranged so that the message is duplicated at some point in the system, this call returns the purpose group of the most recent message entered. The **ENMWhereIs** call is independent of any other message function calls, such as **ENMCreate**, **ENMWriteField**, or **ENMFirstEntry**.

Restrictions

A call to **ENMWhereIs** erases all information stored about an **ENMKeyRead** call. The search key value and the position in a sequence of matching messages is lost. Consequently, a call to **ENMKeyNext** after a call to **ENMWhereIs** will fail with ERR_NO_KEY, even if the call had previously completed successfully.

Example

The following example shows you how to use the **ENMWhereIs** call to ask for an MRN and print the group identifier for this message. For a description of the identifiers, refer to "Purpose Group (GROUP)" on page 16.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;
   GROUP  group;
   UCHAR  mrnin[ MRNlen + 1 ];
   KEY    key;

   rc = ENMSetAppl("PGM1");
   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      printf("Please enter the MRN you are looking for: ");
      scanf( "%s", mrnin );
      strcpy( key.MRN, mrnin );

      rc = ENMWhereIs( KEY_MRN, key, &group );

      if( rc == NO_ERROR )
      {
         printf("The MRN %s is in Group %d\n", mrnin, group );
      }
      else
      {
         printf( "Error in ENMWhereIs, rc = %d\n", rc );
      }
      /* now do the detach from MERVA */
      rc = ENMDetach();
      if( rc != NO_ERROR )
      {
          printf( "Error in detach %d\n", rc );
      }
      else
      {
          printf("Program detached...\n");
      }
   }
   else
```

```
    {
       printf( "Error attaching to MERVA, return code %d\n", rc );
    }
}
```

ENMWriteField—Write Field Associated with Message

Purpose

The **ENMWriteField** function updates information associated with a message.

Format

```
USHORT ENMWriteField(FIELDTYPE FieldType, PFIELD Field)
```

Parameters

FieldType (FIELDTYPE) - input
: The field type contains the name of the field the application wants to write.

Field (PFIELD) - input
: The field union contains the contents of the field the application wants to write.

rc (USHORT) - return
: Values are:

  0 (NO_ERROR)
  : The function completed successfully.

  1 (ERR_SYSTEM_NOT_UP)
  : The MERVA instance has not been started.

  5 (ERR_NOT_ATTACHED)
  : The application is not attached to the MERVA instance.

  7 (ERR_WRITE_TRACE)
  : An error occurred while writing to the trace file.

  112 (ERR_INVALID_FIELDTYPE)
  : The specified field type is not in the range of the FIELDTYPE enumerated data type.

  113 (ERR_INVALID_FIELD)
  : The specified field does not match the rules for the field given by the union FIELD.

  114 (ERR_FIELD_PROTECTED)
  : The specified field is protected against writing.

  203 (ERR_NO_MSG)
  : No message has been retrieved by the application.

Processing

The message length field parameter **FLD_MSGLEN** is optional. Messages returned to a MERVA queue with the **ENMAdd**, **ENMRouteAdd**, **ENMPut**, or **ENMRoutePut** function must always end with a null terminator. This enables the MERVA API to determine the message length. To make use of this automatic length calculation, the **FLD_MSGLEN** field in the message space must be 0.

Alternatively, the **FLD_MSGLEN** field can be set to the correct message length. This means that if the message has been retrieved before and the length has not been changed, the **FLD_MSGLEN** does not need to be set. Unwanted results are caused if the **FLD_MSGLEN** field is not set to **0**, or the correct length is not specified.

The **FLD_MSGLEN** field of user-defined messages containing X'00' characters must always be set to the correct message length before using any of the message routing functions.

Newly created messages require certain information, such as the destination network for the message, to be set using this function.

Restrictions

The following fields are write protected:

- FLD_MRN
- FLD_ISN
- FLD_MSGUSER
- FLD_MSGTRUSR

Before an application can apply changes to a message in a queue, it must:

1. Retrieve the message with a lock.
2. Change the necessary information.
3. Put the message back in the queue (or route it to a different queue).

Example

The following example shows you how you set the NETWORK identifier for a newly created message.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;
   CHAR   msgTxt[ 200 ];
   MMSG   msg;
   FIELD  fldAssociated;

   rc = ENMSetAppl("PGM1");
   rc = ENMAttach( "SAMPLE", "SAMPLE1", "API"  );
   if( rc == NO_ERROR )
   {
      printf("Program successfully attached to MERVA\n");
      rc = ENMCreate( &msg );
      if( rc == NO_ERROR )
      {
         fldAssociated.msgnet = NET_SWIFT;

         rc = ENMWriteField( FLD_MSGNET, &fldAssociated );

         if( rc == NO_ERROR )
         {
            /* create message example type 399 */
            strcpy( msgTxt,
                    "{1:F01VNDPBET2AXXX0000000299}{2:I399VNDPBET2AXXXN}"
                    "{3:{108:399-14}}{4:\r\n:20:399-14\r\n:79:REPLACE MT 101\r\n-}");
            memcpy( msg, msgTxt, strlen(msgTxt) );
            if( ENMRouteAdd( "API_IN" ) == NO_ERROR )
            {
               printf("Message added to Queue API_IN and routed\n");
            }
         }
         else
```

```
            {
               printf("Error in ENMWriteField, rc %d\n", rc );
            }
            rc = ENMClear();
         }
         /* now do the detach from MERVA */
         rc = ENMDetach();
         if( rc != NO_ERROR )
         {
             printf( "Error in detach %d\n", rc );
         }
         else
         {
             printf("Program detached...\n");
         }
      }
      else
      {
         printf( "Error attaching to MERVA, return code %d\n", rc );
      }
   }
```

ENMWriteLog—Writing Diagnosis and Console Log Entries

Purpose

The **ENMWriteLog** function writes a string of up to 240 characters to the diagnosis log file. Additionally, the messages can be specified to appear in the MERVA message console.

The program does not have to be attached to the MERVA instance to call this function. The functionality is independent of the API trace and its functions (**ENMTrace**, **ENMWriteTrace**).

Format

```
USHORT ENMWriteLog(TRACEDATA Line, CON_MSG_ID ConMsgID, INTERVENTION Intervention)
```

Parameters

Line (TRACEDATA) - input
Line contains the information the application requires added to the API log file.

ConMsgID (CON_MSG_ID) - input
The console message identifier parameter specifies the kind of information that is passed in the Line parameter. It can be set to one of the values listed in the **CON_MSG_ID** enumeration. If it is **CON_ID_NONE**, the information is just written to the API log file, but not to the message console. With the other values the message can be specified as information, error, or fatal error message.

Intervention (INTERVENTION) - input
The intervention parameter informs you if operator intervention is required to clear the problem status described by the message. It can be set to one of the values listed in the INTERVENTION enumeration. If the console message identifier parameter is **CON_ID_NONE**, this parameter is ignored.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

12 (ERR_WRITE_LOG)
An error occurred while writing diagnosis information.

Example

The following example shows you how to use the ENMWriteLog to write logging information from an API program to the MERVA API log file and to the MERVA Message Console.

```
#include <signal.h>
#include "enmcapi.h"

main(int argc, CHAR *argv[]")
{
  USHORT    usRc = NO_ERROR;
  TRACEDATA achLogbuf;
  usRc = ENMSetAppl( "PGM1" );
  if (usRc == NO_ERROR)
  {
     usRc = ENMAttach("SAMPLE", "SAMPLE1", "API");
     if (usRc == NO_ERROR)
     {
        ENMWriteLog("Program successfully attached to MERVA",
                     CON_ID_NONE, INT_NOT_REQ);

        /* now do the detach from MERVA */
        usRc = ENMDetach();
        if ( usRc != NO_ERROR)
        {
           sprintf(achLogbuf, "Error in detach rc=%d", usRc);
           ENMWriteLog(achLogbuf, CON_ID_ERROR, INT_REQ);
        }
        else
        {
           ENMWriteLog("Program detached..", CON_ID_NONE, INT_NOT_REQ);
        }
     }
     else
     {
        sprintf(achLogbuf, "Error attaching to MERVA rc=%d", usRc);
        ENMWriteLog(achLogbuf, CON_ID_FATAL, INT_REQ);
     }
  }
  else
  {
     sprintf(achLogbuf, "Error calling ENMSetAppl rc=%d", usRc);
     ENMWriteLog(achLogbuf, CON_ID_ERROR, INT_REQ);
  } /* endif */
}
```

ENMWriteTrace—Write Application Information to Trace File

Purpose

The **ENMWriteTrace** function writes a string of up to 240 characters to the API trace file.

Format

```
USHORT ENMWriteTrace(TRACEDATA Line)
```

Parameters

Line (TRACEDATA) - input
Line contains the information the application requires added to the API trace file.

rc (USHORT) - return
Values are:

0 (NO_ERROR)
The function completed successfully.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

7 (ERR_WRITE_TRACE)
An error occurred while writing to the trace file.

111 (ERR_TRACE_OFF)
Trace is turned off; no information can be written.

Example

The following example shows you how to use the **ENMWriteTrace** call to write trace information from an API program to the MERVA API trace file.

Note: ATTACH or DETACH functions are not necessary to use the trace functions.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmcapi.h"

main()
{
   USHORT rc = 0;

   rc = ENMTrace( ON );
   if( rc == NO_ERROR )
   {
      rc = ENMWriteTrace( "This is trace information from the API program" );
      if( rc == NO_ERROR )
      {
          printf("Trace written\n");
      }
      else
      {
          printf("Error writing trace, rc = %d\n", rc );
      }
   }
   else
```

```
    {
       printf("Error setting trace to ON, rc = %d\n",rc);
    }
}
```

Chapter 4. How to Use, Build, and Load an API Program

This chapter tells you how to prepare MERVA for an API program and how to build an application program.

Preparing MERVA for an API Program

Before you can use an API program, take these steps:

- Use the MERVA Customization program to:
  1. Create the queues to be used by the application program in the API purpose group. If triggering is used, connect the appropriate alarms to the queues.
  2. Set up alarms with the alarm maintenance function, if triggering is used for queues.
  3. Add the routing parameters (fields and constants) used in the conditional routing of these queues.
  4. Set up the routing for these queues.
- Use the MERVA Users program to:
  1. Authorize a user ID passed with the **ENMAttach** call to the API function.

     Note: The following API user rights are provided:
     - Using **API - without password**

       The **ENMAttach** call does not require a password.
     - Using **API - with password**

       The ENMAttach call requires a password.
  2. Assign queues to the user access rights with the **Details** option.
  3. Approve the user created in the previous step.

Using an API Program

To run a program that uses the MERVA API functions, ensure the following:

1. The MERVA instance must be running in multi-user mode before you start the application program.
2. Ensure that the environment variable **ENMD_IPC_DIR** is set.

   Usually, this variable is automatically set in your system environment when you create the MERVA instance.

   If **ENMD_IPC_DIR** is not set, or if your MERVA instance does not run, your program cannot connect to the MERVA services. The program then fails.

An API program that runs when you shut down your MERVA instance is not shut down automatically. The next access to a function that requires the MERVA instance to be running then fails.

Therefore, ensure that your program either shuts down correctly before your MERVA instance is terminated, orhandles the shutdown of the MERVA instance correctly. Do this by:

- Checking every return code for errors indicating that MERVA is not running
- Installing a signal handler in your API program.

Building an API Program

MERVA currently supports the C programming language to write application programs using the API functions. The following compilers are supported:

- IBM VisualAge (R) for C++ 3.5.4 or a subsequent release
- Microsoft Visual C++ 6.0 or a subsequent release

To build an application program:

1. Add the header file **enmcapi.h** to the source file to include all API data definitions, structures, and function prototypes.
2. If you use **enmcapi.dll** for the linking, add the API functions library **enmcapi.lib** to the link step.

If you compile with VisualAge for C++, you can use the sample make file **sample1.mak** as a template for your make files.

To generate an executable, for example, for the **sample1** program:

1. Copy the files **sample1.c** and **sample1.mak** to your working directory. These files are contained in the directory **samples\API** of your MERVA installation directory.
2. Copy the files **enmcapi.h**, **enmcapid.h**, **enmcapif.h**, and **enmcapi.lib** to your working directory. These files are contained in the directory **samples\API** of your MERVA installation directory.
3. Compile and link the program with the following command:

   ```
   nmake -f sample1.mak
   ```

Note: All API programs use the library **enmcapi.dll**. This library is located in the **bin** directory of your installation directory. You can access this library only if you are a member of one of the following groups:

- **Administrator**
- **mervasys**
- **mervalpp**

Note that the developer and the user of the API program have to be a member of one of these groups.

Loading API Functions Dynamically

You have to use dynamically loaded functions if you want to write an API program that:

- Connects to the locally installed MERVA system by using the MERVA API
- Connects to another MERVA system by using MERVA Connection/NT

Each function of the API has a corresponding type definition that is contained in the delivered header file. The following example shows you the type definition for **ENMQueryQueueEx**:

```
USHORT ENMQueryQueueEx(QNAME QueueID, PLONG MessageCount)
typedef USHORT (* PFUNCENMQueryQueueEx)(QNAME,PLONG)
```

To load API functions dynamically, you have to load the library and retrieve the address of every function that you want to use. The following example shows you how to do this:

```
#include <windows.h>
#include "enmcapi.h"
...
HINSTANCE hLibrary;
PFUNCENMAttach  pENMAttach;
PFUNCENMDetach  pENMDetach;

...
hLibrary = LoadLibrary("ENMCAPI");
if (hLibrary!=NULL)
{
    pENMAttach = (PFUNCENMAttach) GetProcAddress(hLibrary,"ENMAttach");
    pENMDetach = (PFUNCENMDetach) GetProcAddress(hLibrary,"ENMDetach");
    ...
    if (pENMAttach!=NULL && pENMDetach!=NULL)
    {
        if (pENMAttach("SAMPLE","SAMPLE1","API")==NO_ERROR)
        {

            /* ... do some processing here ... */

            pENMDetach();
        }
    }
    FreeLibrary(hLibrary);
}
```

Adding an API Program to the MERVA Menu window

To make it easier to access your customer-written programs, for example, an API program, you can add them to the MERVA Menu window. Note that you have to be a DB2 (R) administrator to be able to process the following calls.

Important

Be careful when using these commands. Incorrect use of these commands can make your MERVA instance useless.

You have to do the following:

1. Create a program group:

   ```
   db2 "INSERT into merva2.enmprggp VALUES(<grpID>, <GroupName>)"
   ```

   Where the following applies:

   <grpID>
   The group IDs below 1000 are reserved for MERVA. Use a number higher than 1000 for your own programs.

   <GroupName>
   The name of the program group. The string may not exceed 25 characters.

2. Create one or more program entries:

   ```
   db2 "INSERT into merva2.enmprg VALUES(<ProgramName>,<executable>,<rightID>,<grpID>)"
   ```

   Where the following applies:

   <ProgramName>
   The displayed name of the started program. The length of the name may not exceed 40 characters.

<executable>
The executable that is called if this program is started. The length of the name may not exceed 25 characters.

<rightID>
The right that is connected to this program. Use the numbers 40 to 47 to indicate the rights USER_R1 to USER_R10:

USER_R1 to USER_R7
Use numbers 40 to 46.

USER_R8
Use number 46.

USER_R9, USER_R10
Use number 47.

<grpID>
The ID of the group to which the program belongs.

3. Update the display of the right USER_R1 through USER_R10 by using the following calls:

```
db2 "UPDATE merva2.enmfkdef SET descript = <newText> WHERE right_item = <extRightID>"
```

Where the following applies:

<newText>
The new description for the right USER_R*xx*. The description may not exceed 40 characters. For example, use `USER_R1: Start Company's Application`.

<extRightID>
The right that is to be changed. Use the following values:

4001
For USER_R1

4101
For USER_R2

4201
For USER_R3

4301
For USER_R4

4401
For USER_R5

4501
For USER_R6

4601
For USER_R7

4602
For USER_R8

4701
For USER_R9

4702
For USER_R10

API programs that are called from the MERVA Menu window must use **MEN** as function ID for the call **ENMAttach**. For more information, refer to the description of the **ENMAttach** function.

The following example adds the Windows NT Notepad and the Windows NT Explorer to the MERVA Menu window:

```
db2 "connect to enmcntrl"
db2 "INSERT into merva2.enmprggp VALUES(1000, 'Windows-NT Programs')";
db2 "INSERT into merva2.enmprg   VALUES('Windows-Notepad','notepad', 40, 1000)";
db2 "INSERT into merva2.enmprg   VALUES('Windows-Explorer','notepad', 41, 1000)";
db2 "UPDATE merva2.enmfkdef SET descript = 'USER_R1: Use Windows-Notepad'
                                             WHERE right_item = 4001";
db2 "UPDATE merva2.enmfkdef SET descript = 'USER_R2: Use Windows-Explorer'
                                             WHERE right_item = 4101";
db2 "connect reset"
```

API Sample Programs

The sample programs and source code that show you the use of API function calls to access MERVA with the MERVA API are contained in the samples directory of your MERVA installation directory.

Note: All sample programs, except sample 4, use the user ID **SAMPLE** and the password **SAMPLE1**. User ID and password are case sensitive.

Ensure that the user ID **SAMPLE** is a valid MERVA user ID that has the right to access the API.

For example, the following sample programs are provided:

sample1
: To load messages to the API_IN queue or unload messages from the API_OUT queue.

sample2
: Shows you the triggering concept of MERVA.

sample2s
: Stops the **sample2** program.

sample3
: To load telex messages to the TP2_SND queue.

sample4
: To load or unload messages. It is identical to the **sample1** program, with the exception that the following parameters are variable:
- The queue name of the processed API queue.
- The name of the user who is authorized to use the API functions.
- The password if the API access right **API - with password** is required.
- The way how the length bytes are written to the unload file.
- The network type of the processed message.

The following files are necessary to build your own API programs:

sample1.mak ... sample4.mak
: A sample to compile and link the API sample programs **sample1.c** to **sample4.c** contained in the directory **samples\API**.

enmcapi.h
: The API function prototypes and type definitions contained in the directory **samples\API**.

enmcapif.h
: The API function prototypes and type definitions contained in the directory **samples\API**.

enmcapid.h	The API function prototypes and type definitions contained in the directory **samples\API**.
enmoapi.h	This file ensures compatibility to MERVA AIX. Do not use this file for new development projects.
enmcapi.dll	The API function library contained in the directory **bin**.
enmcapi.lib	The file that links API programs contained in the directory **samples\API**.

Chapter 5. The REXX Function Package

This chapter tells you how to use the Restructured Extended Executor (REXX) language to write application programs with the REXX function package.

Preparing MERVA for a REXX Program

The REXX function package works like an API program that is, for example, written in C. For information on how to prepare MERVA for a REXX program refer to “Chapter 4. How to Use, Build, and Load an API Program” on page 121.

Using the REXX Function Package

The REXX function package for the MERVA API is contained in the file **enmcarex.dll**. This shared library is located in the directory **bin** of the MERVA installation path. To access the functions in the REXX function package, use the following REXX code:

```
rc = RxFuncAdd('ENMLoadFuncs','ENMCAREX','ENMLoadFuncs')
if (rc = 0) then
  call ENMLoadFuncs
else do
  say 'Could not find REXX function package. Is your PATH variable set correctly?'
  exit 0
end
call ENMInit
The ENMInit() function ...
```

The **ENMInit()** function loads all MERVA API constants of the following types into the REXX variable environment:

```
SWITCH
KEYTYPE
FIELDTYPE
NETWORK
GROUP
CON_MSG_ID
INTERVENTION
```

For further details refer to “Chapter 2. MERVA API Data Types” on page 7.

You do not have to call **ENMInit()** if you want to use values instead of variables.

Note: The variables set by **ENMInit()** are only available in the procedure with which **ENMInit()** is called. You must therefore call **ENMInit()** for each procedure in which you want to use the variables.

After you use the MERVA API REXX functions, drop the loaded functions by using the following command:

```
call ENMUnloadFuncs
call RxFuncDrop 'ENMunloadFuncs'
```

Note: This command drops the functions of all REXX programs running on your workstation. If you want to run several REXX programs that use MERVA API REXX functions, use this command after the last running REXX program ends.

REXX Function Calls

Most of the function names and parameters are equal to the API subcommands listed in “Chapter 3. MERVA API Function Calls” on page 19.

Output parameters are denoted as variable names. To avoid that the REXX interpreter evaluates the output parameters, you have to quote them. After the function call ends, the specified variable contains the information.

The following function is different to the description above:

ENMCreate(msg)	The output parameter **msg** is not needed because the memory space for the message is allocated by the shared library. You can set the message text by using the function **ENMPutMsgText(msg)**. This function is new. It is explained in the following paragraph.

The following functions are new:

ENMInit()	Loads all MERVA API constants.
ENMPutMsgText(msg)	This function sets the message text of the currently active message to the value of **MSG**.
ENMGetErrorText(rc)	This function returns the description of a MERVA API return code listed in “Appendix A. Return Codes” on page 141.

The following functions are not supported by the REXX Function Package:

- **ENMCheckUserRight(Right,bRight)**
- All functions to ensure compatibility between MERVA and MERVA Connection/NT. For a detailed description of these functions refer to page 21.

Additionally, the REXX Function Package does not support functions to write and read telex headers.

REXX Return Codes

The MERVA REXX functions return the MERVA API return codes as numbers. For a description of the return codes, refer to “Appendix A. Return Codes” on page 141. Additionally, the variable **ENMerrno** is set to the defined name of the return value, except for the new function **ENMGetErrorText()**.

The following example shows you how to use the **ENMerrno** variable:

```
rc = ENMAttach('SAMPLE',", 'API')
if (ENMerrno = 'ERR_SYSTEM_NOT_UP') then
  say 'Sorry, MERVA is not running.'
else if (rc <> 0) then
  say 'Could not attach to MERVA: 'ENMGetErrorText(rc)
```

Example

The following example shows you how to use the REXX function calls:

```
rc = RxFuncAdd('ENMLoadFuncs','ENMCAREX','ENMLoadFuncs')
if (rc = 0) then
  call ENMLoadFuncs
else do
```

```
  say 'Could not find REXX function package. Is your PATH variable set correctly?'
  exit 0
end
call ENMInit

n = d2c(13) || d2c(10)                   /* CR + LF = new line (for message)      */

rc = ENMAttach('SAMPLE', ', 'API')       /* Attach to MERVA with user ID 'SAMPLE' */
if (ENMerrno = 'NO_ERROR') then do
  say 'Program successfully attached to MERVA'

  rc = ENMCreate('msg')                  /* MSG is only a dummy variable          */
  if (ENMerrno = 'NO_ERROR') then do
    say 'New message created'

    /* Use API constants FLD_MSGNET and NET_SWIFT */
    rc = ENMWriteField(FLD_MSGNET, NET_SWIFT)                /* SWIFT message */
    if (ENMerrno = 'NO_ERROR') then do
      /* create message example */
      msg = '{1:F01VNDPBET2AXXX0000000299}&
{2
:I399VNDPBET2AXXXN}'
      msg = msg||'{3:{108:399-14}};{4:'||n
      msg = msg||':20:399-14'||n||':79:REPLACE MT 101'||n||'-}'

      rc = ENMPutMsgText(msg)            /* Set the text of the message           */

      rc = ENMRouteAdd('API_IN')         /* Add message to queue and route it     */
      if (ENMerrno = 'NO_ERROR') then
        say 'New message added to Queue API_IN and routed.'
      else do
        say 'Error in ENMAdd: 'ENMGetErrorText(rc)
        rc = ENMClear()                  /* If an error occurred clear the message */
      end
    end
    else do
      say 'Error in ENMWriteField: 'ENMGetErrorText(rc)
      rc = ENMClear()                    /* If an error occurred clear the message */
    end
  end
  else
    say 'Error in ENMCreate: 'ENMGetErrorText(rc)

  rc = ENMDetach()                       /* Now do the detach                     */
  if (ENMerrno <> 'NO_ERROR') then
    say 'Error in ENMDetach: 'ENMGetErrorText(rc)
  else
    say 'Program detached from MERVA'

end
else
  say 'Error in ENMAttach: 'ENMGetErrorText(rc)

call ENMUnloadFuncs
call RxFuncDrop 'ENMunloadFuncs'

exit 0                                   /* End REXX program with error code 0    */
```

REXX Sample Files

All sample programs that demonstrate the usage of the REXX function package to access MERVA are contained in the directory **samples/API** of your MERVA directory.

The following sample programs are available:

sample1.rex	Loads messages to the API_IN queue or unloads messages from the API_OUT queue.
sample2.rex	Shows you the triggering concept of MERVA.
sample2s.rex	Stops the **sample2.rex** program.
sample4.rex	Loads or unloads messages like the **sample1.rex** program with the exception that the following parameters are variable: • The queue name of the processed API queue. • The name of the user who is authorized to use the API functions. • The password if the API access right **API - with password** is required. • The way how the length bytes are written to the unload file. • The network type of the processed message.

Note: All sample programs, except sample 4, use the user ID **SAMPLE** and the password **SAMPLE1**. User ID and password are case sensitive.

Ensure that the user ID **SAMPLE** is a valid MERVA user ID that has the right to access the API.

Chapter 6. The SWIFT Link API

The SWIFT Link API offers you new functions that help you handle checksum and authentication for a message. You can:

- Calculate, compare, add, and replace the checksum of a message
- Authenticate a message

With the SWIFT Link API you can, for example, retrieve or create a message by using a MERVA API program. You can then check or authenticate this message with the new SWIFT Link API functions.

The sample program **samp_auth.c** and the corresponding make file**samp_auth.mak** show how to use the API.

Note: Ensure that you are attached to MERVA before you call any of the SWIFT Link API functions.

The functions are contained in the library **enmcaaut.dll**. You also need the files **enmcaaut.lib** and **enmcaaut.h** to use these functions.

The functions are:

- ENMChecksum
  ```
  INT ENMChecksum(INT iBuflen, PINT piMsglen, MMSG pszMsg, INT iOption)
  ```
- ENMAuthenticate
  ```
  LONG ENMAuthenticate(USHORT usBuflen, USHORT *pusMsglen, MMSG pszMsg,
                       USHORT usOption, INF_MMSG infMsg1);
  ```

The following list explains the abbreviations used in the function code:

LONG	Long
USHORT	Unsigned short
INT	Integer
PINT	Pointer to integer
MMSG	Pointer to unsigned char string
INF_MMSG	Array of 256 characters

Note: You must start MERVA before you can use the SWIFT Link API functions.

The following sections describe the new functions in detail.

ENMChecksum—Handle Message Checksum

Purpose

The **ENMChecksum** function calculates the checksum and adds or replaces the checksum at the end of the message. It also compares the checksum with the passed checksum of the message. If the message is sent from a training LT and if the training trailer is missing, the training trailer is added.

Format

```
INT ENMChecksum(INT iBuflen, PINT piMsglen, MMSG pszMsg, INT iOption)
```

Parameters

iBuflen (INT) - input
Length of allocated memory for the message.

piMsglen (PINT) - input
Pointer to input message length. If the message length changes, it points to the changed output message length.

pszMsg (MMSG) - input
Pointer to message. Must be terminated by zero.

iOption (INT) - input
Indicates processing of output messages. This parameter is valid only for output messages.

CHK_CHKSM - 0
Calculates the checksum and checks it with the passed checksum.

ADD_CHKSM - 1
Calculates the checksum and adds or replaces it at the end of the message.

rc (INT) - output
Values are:

0 (NO_ERROR)

Input message:
The checksum is added or replaced successfully.

Output message:

- Option = 1: The checksum is added or replaced successfully.
- Option = 0: The checksum at the end of the message is correct.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

5 (ERR_NOT_ATTACHED)
The API program is not yet attached to MERVA.

1002 (ERR_MSG_TOO_LONG)
The message is too long. Therefore, the trailer could not be added.

1021 (NO_FIN_MSG)
The message is not a user-to-user or a system message (01 service identifier).

1022 (ERR_CHKSM_FAILED)
The checksum for the output message failed.

1023 (ERR_MMSG_FORMAT)

A format error occurred, for example, braces are not paired, or incorrect CHK trailer occurred.

Changes: If the trailer is empty, it is inserted.

- Input:

  ```
  ...{CHK:}...
  ```

- Output:

  ```
  ...{CHK:NNNNNNNNNNNN}...
  ```

Example

```
before:
 {1:F01IBMADEF0AXXX0116001378}{2:I100IBMADEF0XXXXN}{3:{108:100-02}}{4:
 :20:100-01
 :32A:960326NLG958,47
 :50:FRANZ HOLZAPFEL G.M.B.H
 WIEN
 :59:H.F.JANSSEN
 LEDEBOERSTRAAT 27
 AMSTERDAM
 -}{5:{MAC:E187CE93}{CHK:111111111111}{TNG:}}";
after:
 {1:F01IBMADEF0AXXX0116001378}{2:I100IBMADEF0XXXXN}{3:{108:100-02}}{4:
 :20:100-01
 :32A:960326NLG958,47
 :50:FRANZ HOLZAPFEL G.M.B.H
 WIEN
 :59:H.F.JANSSEN
 LEDEBOERSTRAAT 27
 AMSTERDAM
 -}{5:{MAC:E187CE93}{CHK:06D9BD12CB96}{TNG:}}";

before:
 {1:F01IBMADEF0AXXX0116001378}{2:I100IBMADEF0XXXXN}{3:{108:100-02}}{4:
 :20:100-01
 :32A:960326NLG958,47
 :50:FRANZ HOLZAPFEL G.M.B.H
 WIEN
 :59:H.F.JANSSEN
 LEDEBOERSTRAAT 27
 AMSTERDAM
 -}";
after:
 {1:F01IBMADEF0AXXX0116001378}{2:I100IBMADEF0XXXXN}{3:{108:100-02}}{4:
 :20:100-01
 :32A:960326NLG958,47
 :50:FRANZ HOLZAPFEL G.M.B.H
 WIEN
 :59:H.F.JANSSEN
 LEDEBOERSTRAAT 27
 AMSTERDAM
 -}{5:{CHK:06D9BD12CB96}{TNG:}}";


#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmoapi.h"
#include "enmcaaut.h"

#define MAXMSGLEN  10001    /* Message buffer of msg got from ENMFirstEntry
                               is 28000, but maxlength of SWIFT messages
                               is 10000. */
```

```
void main(int argc, char *argv[])
{
   INT    rc = NO_ERROR;
   MMSG   msg;
   USHORT usorglen;
   INT    option = ADD_CHKSM;   /* Add or Replace CHK Trailer */
   INT    msglen;
   INT   msg_found = TRUE;

   if (4 != argc)
   {
      printf("\n   This test program add or replace a CHK Trailer");
      printf("\n   of a SWIFT message, which is got from a API queue.\n");
      printf("\n   PARAMETER:\n");
      printf("\n   userid   = UserID which is used for API program.");
      printf("\n   password = Password of UserID");
      printf("\n   queue    = Queue from which a message is modified");
      printf("\n              and routed.");
      printf("\n   Usage : ex2 userid password queue \n");
      printf("\n   e.g. ex2 merva1 xxxxx API_IN\n");
   }
   else
   {
      rc = ENMAttach( argv[1], argv[2], "API" );

      if ( rc == NO_ERROR )
      {
         printf("Program successfully attached to MERVA\n");

         rc = ENMFirstEntry(argv[3],ON,&msg,&usorglen);
         if ( rc != NO_ERROR )
         {
            printf( "Error in ENMFirstEntry rc %d\n", rc );
            msg_found = FALSE;
         }
         else
         {
             printf( "FirstEntry locked\n" );
         }

         if (msg_found == TRUE)
         {
            msglen = strlen(msg);
            rc = ENMChecksum(MAXMSGLEN, &msglen, msg, option );
            if ( rc != NO_ERROR )
            {
               printf( "Error in ENMChecksum rc %d\n", rc );
            }
            else
            {
               printf( "ENMChecksum successful msg = %s.\n",msg );
            }

            if (rc != NO_ERROR)
            {
               rc = ENMFree();
               if ( rc != NO_ERROR )
               {
                  printf( "Error in ENMFree rc %d\n", rc );
               }
               else
               {
                  printf( "ENMFree successful.\n" );
               }
            }
            else
            {
               rc = ENMRoutePut();
               if ( rc != NO_ERROR )
               {
                  printf( "Error in ENMRoutePut rc %d\n", rc );
               }
               else
               {
                  printf( "ENMRoutePut successful.\n" );
               }
            }
         }

         /* Now do the detach from MERVA */
         rc = ENMDetach();
         if ( rc != NO_ERROR )
         {
            printf( "Error in detach %d\n", rc );
         }
         else
         {
            printf( "Program detached...\n" );
         }
      }
      else
      {
         printf( "Error attaching to MERVA, return code %d\n", rc );
      }
   }
}
```

ENMAuthenticate—Authenticate Message

Purpose

The **ENMAuthenticate** function authenticates a SWIFT message. The MAC and PAC trailer handling depends on the option parameter and the message type. The message type can be input or output message.

Format

```
LONG ENMAuthenticate(USHORT usBuflen, USHORT *pusMsglen, MMSG pszMsg,
                     USHORT usOption, INF_MMSG infMsg1)
```

Parameters

usBuflen (USHORT) - input
Length of allocated memory for the message.

pusMsglen (USHORT*) - input/output

Input	Pointer to input message length.
Output	New message length if the message is changed.

pszMsg (MMSG) - input
Pointer to message. Must be terminated by zero.

usOption (USHORT) - input
Indicates processing of MAC and PAC trailer.

infMsg1 (INF_MMSG) - input/output
Returns an information or error message. You should define this parameter as an array with 256 characters.

The following list shows you the information or error messages that are returned in the **infMsg1** array:

ENN9128E	AUTH failed <homeDest> / <correspDest> record.
ENN9129I	Auth OK, discontinued key <ID>, <homeDest> / <correspDest> record.
ENN9130I	Auth OK, day = <dayDifference>, key <ID>, <homeDest> / <correspDest> record.
ENN9131I	Auth OK, key <ID>, <homeDest> / <correspDest> record.
ENN9132I	Message not to be authenticated.
ENN9133I	PAC trailer is empty (bypassed mode).
ENM9979E	<homeDest> / <correspDest> key for authentication not found.
ENM9980I	<homeDest> / <correspDest> suspended or excluded BK record found.
ENM9981I	Auth OK, suspended or excluded BK record used <homeDest> / <correspDest>, key <ID>
ENM9982I	<homeDest> / <correspDest> message not to be authenticated.
ENM9983E	<homeDest> message format error.
ENM9984E	<homeDest> message does not contain text.
ENM9985E	<homeDest> / <correspDest> authentication failed.

ENM9986E Message too long. Authentication trailer could not be added.

rc (LONG) - output
Values are:

0 (NO_ERROR)
Authentication OK with any key.

1 (ERR_SYSTEM_NOT_UP)
The MERVA instance has not yet been started.

5 (ERR_NOT_ATTACHED)
The API program is not yet attached to MERVA.

1001 (ERR_INV_HEADER)
The header of the message is not valid.

1002 (ERR_MSG_TOO_LONG)
The message is too long. Therefore, the authentication trailer could not be added.

1003 (ERR_KEY_NOT_FOUND)
The key for authentication could not be found.

1004 (NOT_TO_BE_AUTH)
The message is not to be authenticated.

1005 (ERR_MSG_EMPTY)
The message does not contain text.

1007 (ERR_AUTH_FAILED)
Authentication failed.

1008 (ERR_NO_PAC_DEF)
The FIN copy definition for the PAC trailer could not be found.

1009 (ERR_PAC_BYPASS)
The PAC trailer is empty. This state is also called bypassed mode.

1010 (ERR_CONN_CTLDB_FAILED)
The connection to the control database failed.

Processing

With this call, you can add or check the MAC and PAC trailer of a SWIFT message, depending on the option parameter and the message type:

- Input Messages:

  With option **ADDMAC_ADDPAC**, the MAC trailer is added if the message has to be authenticated according the SWIFT rules. The PAC trailer is added if it is a FIN Copy message.

  With option **ADDMAC_NOPAC**, the MAC trailer is added if the message has to be authenticated according the SWIFT rules. The PAC trailer is not to be handled.
- Output Messages:

  With option **CHKMAC_CHKPAC**, the MAC Trailer is checked. The PAC trailer is checked if it is customized accordingly.

  With option **ADDMAC_NOPAC**, the MAC trailer is added or replaced. The PAC trailer is not handled.

If a trailer is added, the message length is increased. If a message contains an empty MAC trailer (`{MAC:}`), the MAC trailer is extended with `00000000` (`{MAC:00000000}`). Depending on the outcome of the authentication, additional information is returned in parameter **infMsg1**, for example, the authentication key that was used. Additional MAC information is written from byte 0 to byte 127 of **infMsg1**, additional PAC information is written from byte 128 to byte 255 of **infMsg1**.

The **ENMAttach** function must be called before you can use the **ENMAuthenticate** function.

Example

```
before:
 {1:F01IBMDDEFFAXXX0414005032}{2:I199IBMADEFFXXXXN}{4:
 :20:SCH
 :79:TEST MMSG
 -}{5:CHK:794D4B15701B}}
after:
 {1:F01IBMDDEFFAXXX0414005032}{2:I199IBMADEFFXXXXN}{4:
 :20:SCH
 :79:TEST MMSG
 -}{5:{MAC:CCCB8B8F}{CHK:794D4B15701B}}

before:
 {1:F01IBMDDEFFAXXX0414005032}{2:I199IBMADEFFXXXXN}{4:
 :20:SCH
 :79:TEST MMSG
 -}
after:
 {1:F01IBMDDEFFAXXX0414005032}{2:I199IBMADEFFXXXXN}{4:
 :20:SCH
 :79:TEST MMSG
 -}{5:{MAC:CCCB8B8F}}


#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "enmoapi.h"
#include "enmcaaut.h"
#define MAXMSGLEN  10001    /* Messagebuffer of msg got from ENMFirstEntry
                               is 28000, but maxlength of SWIFT messages
                               is 10000. */

void main(int argc, char *argv[])
{
   INT    rc = NO_ERROR;
   MMSG   msg;
   USHORT usorglen;
   USHORT option = ADDMAC_NOPAC;   /* Add or Replace MAC Trailer */
   USHORT msglen;
   INT   msg_found = TRUE;
   UCHAR inf_msg[INF_MSG_LEN + 1];

   if (4 != argc)
   {
      printf("\n   This test program checks or replace a MAC Trailer");
      printf("\n   of a SWIFT message, which is got from a API queue.\n");
      printf("\n   PARAMETER:\n");
      printf("\n   userid   = UserID which is used for API program.");
      printf("\n   password = Password of UserID");
      printf("\n   queue    = Queue from which a message is modified");
      printf("\n              and routed.");
      printf("\n   Usage : ex1 userid password queue \n");
      printf("\n   e.g. ex1 merva1 xxxxx API_IN\n");
   }
   else
   {
```

```
rc = ENMAttach( argv[1], argv[2], "API" );

if ( rc == NO_ERROR )
{
   printf("Program successfully attached to MERVA\n");

   rc = ENMFirstEntry(argv[3],ON,&msg,&usorglen);
   if ( rc != NO_ERROR )
   {
      printf( "Error in ENMFirstEntry rc %d\n", rc );
      msg_found = FALSE;
   }
   else
   {
         printf( "FirstEntry locked\n" );
   }

   if (msg_found == TRUE)
   {
      msglen = strlen(msg);
      rc = ENMAuthenticate(MAXMSGLEN, &msglen, msg, option, inf_msg);
      if ( rc != NO_ERROR )
      {
         printf( "Error in ENMAuthenticate rc %d\n", rc );
      }
      else
      {
         printf( "ENMAuthenticate successful msg = %s.\n",msg );
      }

      if (rc != NO_ERROR)
      {
         rc = ENMFree();
         if ( rc != NO_ERROR )
         {
            printf( "Error in ENMFree rc %d\n", rc );
         }
         else
         {
            printf( "ENMFree successful.\n" );
         }
      }
      else
      {
         rc = ENMRoutePut();
         if ( rc != NO_ERROR )
         {
            printf( "Error in ENMRoutePut rc %d\n", rc );
         }
         else
         {
            printf( "ENMRoutePut successful.\n" );
         }
      }
   }

   /* Now do the detach from MERVA */
   rc = ENMDetach();
   if ( rc != NO_ERROR )
   {
      printf( "Error in detach %d\n", rc );
   }
   else
   {
      printf( "Program detached...\n" );
   }
}
```

```
      else
      {
         printf( "Error attaching to MERVA, return code %d\n", rc );
      }
   }
}
```

Appendix A. Return Codes

Parameter name (data type of parameter**)** - **input** or **output**

If the function type is not **void**, the last entry in the parameter list is **rc**. It contains a list of return codes that are valid for the function.

Each description contains the following parts:
- Value of the return code
- Defined name of the return code
- Meaning of the return code

This appendix shows you the return codes that are returned by the API function calls.

0 (NO_ERROR)

Problem Determination: The function completed successfully. The MERVA instance completed an API call without an error.

System Action: The API call is processed.

User Response: Continue with normal processing.

1 (ERR_SYSTEM_NOT_UP)

Problem Determination: The MERVA instance has not yet been started or is started in customization mode.

System Action: The API call could not be processed because the MERVA system does not run in the right mode.

User Response: Start the MERVA instance, then restart your application.

2 (ERR_SYSTEM)

Problem Determination: An error occurred in the MERVA instance. An internal MERVA error occurred during processing of the API call. The processing is not complete. Further API processing can lead to unpredictable results.

System Action: The API call is not processed.

User Response: Shut down the application program and MERVA. Read the MERVA diagnosis log file to find the error and its reason.

3 (ERR_ATTACH_FAILED)

Problem Determination: MERVA did not attach to the application program because another application program with the same name is already connected to MERVA.

System Action: The API call is not processed.

User Response: Use a different identifier to attach the application program to MERVA or wait until the identifier is free.

4 (ERR_DETACH_FAILED)

Problem Determination: The detach failed because of internal errors. MERVA could not disconnect the application program due to an error.

System Action: The API call is not processed.

User Response: Shut down the application program and MERVA. Read the MERVA diagnosis log file to find the error and its reason.

5 (ERR_NOT_ATTACHED)

Problem Determination: The application program is not attached to the MERVA instance.

System Action: MERVA does not process the API call because the application program is not connected to the MERVA instance.

User Response: Connect the application program to the **ENMAttach()** call.

6 (ERR_OUT_OF_MEMORY)

Problem Determination: Windows NT could not supply the requested amount of memory to the API.

System Action: The API call is not processed.

User Response: Increase the amount of memory or stop other programs.

7 (ERR_WRITE_TRACE)

Problem Determination: Information could not be added to the trace file.

System Action: The API tried to add information to the trace file but did not succeed.

User Response: Ensure that the disk space for the API file is sufficient and try again.

8 (ERR_ROUTING)

Problem Determination: The router could not identify a destination queue for the message.

System Action: The message is left in the message space. It is still part of the queue from which it was read.

User Response: Check the routing conditions for the queue from which the message was read.

9 (ERR_NO_FREE_SLOT)

Problem Determination: All slots to attach to Base Functions services are currently in use.

System Action: The API program is not attached to MERVA.

User Response: Stop one of the MERVA API programs or wait until a program ends.

10 (ERR_SIGNON_FAILED)

Problem Determination: The application program cannot sign on to the MERVA Control Process.

The signon failed for one of the following reasons:

Reason	Explanation
702	MERVA Control Process not in MERVA up status
907	Maximum number of MERVA instances reached
911	A necessary process does not run
912	Another process that may not run is running

System Action: The API program is not attached to MERVA.

User Response: Shut down the application program and MERVA. Read the MERVA diagnosis log file to find the error and its reason.

11 (ERR_PROCESS_EXCEEDED)

Problem Determination: The **ENMWaitSemList** call executes several subprocesses. Windows NT could not supply the requested amount of processes.

System Action: The API call is not processed.

User Response: Stop other processes on the system.

12 (ERR_WRITE_LOG)

Problem Determination: An error occurred while writing diagnosis information.

System Action: The API tried to add information to the trace file but did not succeed.

User Response: Ensure that the disk space for the API file is sufficient and try again.

20 (ERR_APPLICATION_SET)

Problem Determination: The application name is already set.

System Action: The API call is not processed.

User Response: Check the sequence of the calls **ENMSetAppl** and **ENMAttach**. Check whether the **ENMSetAppl** call is used more than once.

21 (ERR_WRONG_LENGTH)

Problem Determination: The application identifier is too long.

System Action: The API call is not processed.

User Response: Correct the application identifier in the **ENMSetAppl** function. Its length can be up to 8 characters.

22 (ERR_NO_API_QUEUE)

Problem Determination: There is no predefined queue that belongs to the API purpose group.

System Action: The API call is not processed.

User Response: Use the MERVA Customization program to define a queue that belongs to the API purpose group.

23 (ERR_NO_API_QUEUE_ASSIGNED)

Problem Determination: A queue that belongs to the API purpose group is not assigned to the user access right.

System Action: The API call is not processed.

User Response: Use the MERVA Users program to assign a queue to the API access right **API - with password** or **API - without password**

31 (ERR_SEMAPHORE_NO_AUTHORITY)

Problem Determination: The API semaphore call **ENMCloseSem** causes a problem regarding authority. The semaphore identifier could not be removed because the calling process does not have the necessary permission.

System Action: The API call is not processed.

User Response: Synchronize the API programs that use the semaphore calls. The program that called **ENMCreateSem** must close the semaphore with **ENMCloseSem**.

---

36 (ERR_SEMAPHORE_REMOVED)

Problem Determination: The API semaphore call **ENMWaitSemList** causes an error. A waiting semaphore is removed from the system.

System Action: The API call is not processed.

User Response: Restart the API program.

---

100 (ERR_TOO_MANY_SEMAPHORES)

Problem Determination: The API semaphore call **ENMCreateSem** causes an error. The Windows NT system has reached the maximum number of concurrently running semaphores.

System Action: The API call is not processed.

User Response: Remove one or more semaphores from the system.

---

101 (ERR_NO_QUEUE_NAME)

Problem Determination: The specified queue name is empty or too long. The character length of the input queue name is incorrect. A queue name was not supplied, or the queue name contained too many characters.

System Action: The API call is not processed.

User Response: Set the queue name variable to a null-terminated string. Then restart your application.

---

102 (ERR_INVALID_QUEUE_NAME)

Problem Determination: The specified queue does not belong to the API purpose group, or the user has no right to use the specified queue. A queue with the specified name is not defined in the API purpose group, or the queue is defined but it is not assigned to the user API right.

System Action: The API call is not processed.

User Response: Use the MERVA Customization program to define a queue that belongs to the API purpose group, or use a queue name that is already defined for this purpose group. To assign a queue to a user right, use the MERVA Users program.

---

103 (ERR_NO_KEY)

Problem Determination: The specified key is empty. The length of the input key is not correct.

System Action: The API call is not processed.

User Response: Set the key variable to a null-terminated string. Then restart your application.

---

104 (ERR_INVALID_MRN)

Problem Determination: The format of the MRN specified as key is not valid. The last eight characters of the MRN string are not numeric, or the first eight characters are not alphanumeric.

System Action: The API call is not processed.

User Response: Correct the MRN, then restart your application.

---

105 (ERR_INVALID_ISN)

Problem Determination: The format of the ISN specified as key is not valid. A character in the ISN string is not numerical.

System Action: The API call is not processed.

User Response: Correct the ISN, then issue the call again.

---

106 (ERR_INVALID_ID)

Problem Determination: The application identifier set by the **ENMSetAppl** call starts with an illegal prefix.

System Action: The API call is not processed.

User Response: Correct the application identifier.

---

107 (ERR_NOT_SWITCH)

Problem Determination: A value other than ON or OFF has been passed to a variable with the SWITCH data type. The value of the input variable is not defined by the SWITCH data type.

System Action: The API call is not processed.

User Response: Correct the value, then restart your application.

---

108 (ERR_INVALID_KEYTYPE)

Problem Determination: The specified key type is not in the range of the KEYTYPE enumerated data type. The value of the input key type is outside the range of the defined key types.

System Action: The API call is not processed.

User Response: Correct the value, then restart your application.

---

109 (ERR_NO_PASSWD)

Problem Determination: The password is not specified or too long. The variable for the input password has a length of zero or more than 8 characters.

System Action: The API call is not processed.

User Response: Set the password variable to a null-terminated string of up to 8 characters, then restart your application.

110 (ERR_NO_AUTHORIZATION)

Problem Determination: An attempt was made to attach to MERVA but an authorization problem occurred. The application program is not authorized to access messages.

System Action: The application program is not attached.

User Response: Read the MERVA log file to find the error and its reason.

111 (ERR_TRACE_OFF)

Problem Determination: The application program tried to write information to the API trace file but the API trace was not started.

System Action: Information is not written to the API trace file.

User Response: Set the API trace to ON, then restart your application.

112 (ERR_INVALID_FIELDTYPE)

Problem Determination: The specified field type is not in the range of the FIELDTYPE enumerated data type. The information in the FIELD structure is not valid.

System Action: Information is not retrieved.

User Response: Use one of the field types defined by the FIELDTYPE enumerated data type.

113 (ERR_INVALID_FIELD)

Problem Determination: The specified field does not conform to the rules for the field specified by the field type.

System Action: The information associated with the actual message is not changed.

User Response: Check the strings regarding null terminators. Fields that have to conform to additional rules are described in "Chapter 2. MERVA API Data Types" on page 7.

114 (ERR_FIELD_PROTECTED)

Problem Determination: The field for information associated with a message cannot be changed by the application.

System Action: The information is not changed.

User Response: Use an unprotected field.

115 (ERR_SWIFT_HEAD)

Problem Determination: The header of the message does not conform to the rules for SWIFT headers.

System Action: The message is not passed to MERVA.

User Response: Change the message header information to comply with the SWIFT network header rules. They are described in "Appendix B. Message Header Checking" on page 149.

116 (ERR_TELEX_HEAD)

Problem Determination: The header of the message does not conform to the rules for telex network headers.

System Action: The message is not passed to MERVA.

User Response: Change the message header information to comply with telex network header rules. They are described in "Appendix B. Message Header Checking" on page 149.

117 (ERR_NETWORK)

Problem Determination: The **MSG_NET** field does not contain a valid network identifier. A value is outside the range defined for network identifiers in the field.

System Action: The API call is not processed.

User Response: Set the value to one of the defined networks.

118 (ERR_NO_USERID)

Problem Determination: The user ID does not exist, or the user ID is more than eight characters long.

System Action: The API call is not processed.

User Response: Check that the user ID exists, or check whether it is more than eight characters long.

119 (ERR_NO_FUNCID)

Problem Determination: The function ID is empty, or the function ID does not contain the string **API**.

System Action: The API call is not processed.

User Response: Check that the function ID contains the string **API**.

121 (ERR_SEMAPHORE_TIMEOUT)

Problem Determination: The waiting time has passed.

System Action: The API program is reactivated and continues processing.

User Response: Test this error in API program take the necessary steps.

123 (ERR_INVALID_SEMAPHORE_NAME)

Problem Determination: The semaphore name is not a valid Windows NT file name.

System Action: The API call is not processed.

User Response: Change the semaphore name to comply with the Windows NT file naming conventions.

183 (ERR_SEMAPHORE_ALREADY_EXISTS)

Problem Determination: The semaphore already exists.

System Action: The API call is not processed.

User Response: Rename the semaphore or remove it from the system. Then retry the call.

187 (ERR_SEMAPHORE_NOT_EXISTS)

Problem Determination: The semaphore to be opened does not exist.

System Action: The API call is not processed.

User Response: Find out why the semaphore does not exist. For example, the API program that creates the semaphore ended or did not run.

201 (ERR_NO_MSG_LOCKED)

Problem Determination: A message has not been locked by the application program. The application program called an API function that can handle only locked messages.

System Action: The API call is not processed.

User Response: Retrieve and lock a message with an **ENMKeyxxx()** or an or an **ENMxxxEntry()** call with the lock set to **ON**. Then restart your application.

202 (ERR_NO_MSG_CREATED)

Problem Determination: The message space does not contain a message, or the message was not created by the API. The application program called an API function that can handle only messages created immediately before this call.

System Action: The API call is not processed.

User Response: Create a new message with the **ENMCreate()** call, then restart your application.

203 (ERR_NO_MSG)

Problem Determination: The application program did not retrieve a message. The application program called an API function that requires a message for processing.

System Action: The API call is not processed.

User Response: First retrieve a message with an **ENMKeyxxx()**, **ENMxxxEntry()**, or **ENMGetxxx()** call. Then retry the call.

204 (ERR_MSG_INUSE)

| **Problem Determination:** One of the following:

- | The application program called a function that needs an empty message space. However, the API contains a locked or created message in the message space.
- | The application program requires a message that is locked by another MERVA user or MERVA program. Probably the user or program terminated abnormally without first freeing the message.

System Action: The API call is not processed.

| **User Response:** Check whether there is a locked or created message in the message space. If so, free the message by calling an API function that unlocks a message after processing. If not, the problem is that the IN_USE flag for the message is set. Either reset this flag manually, or restart the MERVA system (this will reset this flag automatically).

207 (ERR_CRC_CHECK)

Problem Determination: A CRC error on the control database occurred.

System Action: The API call is not processed.

User Response: Check the corrupted control database. Repair the database, then try again.

214 (ERR_USERID_NOT_FOUND)

Problem Determination: The specified user ID is not defined in MERVA.

System Action: The API call is not processed.

User Response: Define the user ID in MERVA with the Users program or specify another user ID. Then try again.

216 (ERR_RIGHTS_NOT_APPROVED)

Problem Determination: The API access rights assigned to the user are not yet approved.

System Action: The API call is not processed.

User Response: Approve the user rights with the Users program. Then try again.

217 (ERR_NO_RIGHTS)

Problem Determination: You are not authorized to execute an API application program, or you have only the access right **API - with password** but a password is not specified.

System Action: The API call is not processed.

User Response: Use the MERVA Users program to assign the API access rights or specify a valid password in the **ENMAttach** call.

---

255 (ERR_SEMAPHORE_FAILED)

Problem Determination: The semaphore call failed with an internal error.

System Action: The API call is not processed.

User Response: Stop the application program and read the MERVA log file to find the error and its reason.

---

301 (ERR_MSG_LOCKED)

Problem Determination: The message found is already locked. The API tried to get a lock for the message and was rejected because another application program had that lock already.

System Action: The message space is returned empty.

User Response: Wait for the other application program to free the message, or try to read another message.

---

302 (ERR_MSG_NOT_FOUND)

Problem Determination: No matching message could be found. The API call did not produce data.

System Action: The message space is returned empty.

User Response: Try a different call or a different queue. Possibly, there is no data that the application program can process.

---

303 (ERR_CHECK_MSG)

Problem Determination: A message-processing error occurred while the message was checked.

System Action: The API call is not processed.

User Response: Shut down the application program and MERVA. Read the MERVA diagnosis log file to find the error and its reason.

---

304 (NO_CHECK_ERROR)

Problem Determination: No checking error was found.

System Action: The message is syntactically and semantically correct.

User Response: Continue with normal processing.

---

305 (ERR_MSG_SYNTAX)

Problem Determination: A syntactical error was found in the message.

System Action: The API call is processed.

User Response: Read the MERVA diagnosis log to get information about the found errors. Correct the message and retry.

---

306 (ERR_MSG_SEMANTIC)

Problem Determination: A semantic error was found in the message.

System Action: The API call is processed.

User Response: Read the MERVA diagnosis log file to get information about the found errors. Correct the message and retry.

---

406 (ERR_WRONG_PASSWD)

Problem Determination: The specified password does not conform to your MERVA password.

System Action: The API call is not processed.

User Response: Set the password variable to your MERVA password, then restart your application.

---

407 (ERR_USERID_REVOKED)

Problem Determination: The specified user ID is revoked by the MERVA instance. You tried to log on to MERVA with a wrong password more than five times.

System Action: The API call is not processed.

User Response: Reset the user ID to a valid user ID with the Users program.

---

410 (ERR_NO_PASSWD_SET)

Problem Determination: No initial password is defined for this user in the Users program.

System Action: The API call is not processed.

User Response: Use the MERVA Users program to reset the password of this user to an initial value.

---

414 (ERR_GET_PSW)

Problem Determination: Reading the user's locally defined password information fails because the MERVA instance does not have root user authority.

System Action: The API call is not processed.

User Response: Start the API program without a password or contact your MERVA system administrator to restart the MERVA instance with root user authority.

---

415 (ERR_WRONG_AIX_PSW)

Problem Determination: The specified password does not conform to your Windows NT password.

System Action: The API call is not processed.

User Response: Set the password variable to your

Windows NT password, then restart your application. Ensure that your Windows NT and MERVA passwords are identical.

416 (ERR_NOTIFY_FAILED)

Problem Determination: Cannot get the MERVA logon user ID and password.

System Action: The API call is not processed.

User Response: Check that the API program is started from the MERVA Menu window, or that MERVA is started correctly.

417 (ERR_NO_NOTIFY)

Problem Determination: **ENMAttach** has the wrong function ID.

System Action: The API call is not processed.

User Response: Correct **ENMAttach**, then restart the program.

418 (ERR_CKRIGHT_FAILED)

Problem Determination: The **ENMCheckUserRight** function failed due to an internal error.

System Action: The API call is not processed.

User Response: Shut down the application program and read the MERVA diagnosis log file to find the error and its reason.

| **419 (ERR_MSG_INVALID_LENGTH)**

| **Problem Determination:** The **ENMCheckSwiftMsg**
| function failed because the supplied message was too
| long for the corresponding message type.

| **System Action:** The API call is processed, and the API
| continues with normal processing.

| **User Response:** None.

| **420 (ERR_NO_MSG_TYPE)**

| **Problem Determination:** No message type
| information was found in the supplied message buffer.

| **System Action:** The API call is not processed, and the
| API continues with normal processing.

| **User Response:** If this is a problem, modify your
| program so that the appropriate message type
| information is provided in the message buffer.

| **421 (ERR_NO_DATA)**

| **Problem Determination:** The message buffer that was
| prepared for the API call contains no data (that is, it
| contains an empty string) or is not allocated.

| **System Action:** The API call is not processed, and the
| API continues with normal processing.

| **User Response:** If this is a problem, modify your
| program so that the message buffer is filled with the
| appropriate data.

1001 (ERR_INV_HEADER)

Problem Determination: The message header is incorrect.

System Action: The API call is not processed.

User Response: Check the format of the message to be authenticated.

1002 (ERR_MSG_TOO_LONG)

Problem Determination: The message is too long to add a trailer. Depending on the message type, the maximum length of SWIFT messages is 2000 or 10000 bytes.

System Action: The API call is not processed.

User Response: Reduce the length of the message including trailer to 2000 or 10000 bytes depending on the message type.

1003 (ERR_KEY_NOT_FOUND)

Problem Determination: Key not found for authentication.

System Action: The API call is not processed.

User Response: Refer to the parameter **infMsg1** of **ENMAuthenticate** and check whether a valid key for the destinations exists.

1004 (NOT_TO_BE_AUTH)

Problem Determination: The message does not have to be authenticated.

System Action: The API call is not processed.

User Response: The message type that you want to authenticate does not have to be authenticated.

1005 (ERR_MSG_EMPTY)

Problem Determination: The message does not contain text.

System Action: The API call is not processed.

User Response: Check the message that you want to authenticate.

1007 (ERR_AUTH_FAILED)

Problem Determination: Authentication failed.

System Action: The API call is not processed.

User Response: Refer to the parameter **infMsg1** of **ENMAuthenticate** for more information.

1008 (ERR_NO_PAC_DEF)

Problem Determination: The FIN copy definition for the PAC trailer was not found.

System Action: The API call is not processed.

User Response: Check the customization of your FIN copy. Refer to the parameter **infMsg1** of **ENMAuthenticate** for more information.

1009 (ERR_PAC_BYPASS)

Problem Determination: The PAC trailer is empty.

System Action: The API call is processed.

User Response: None.

1010 (ERR_INVALID_OPTION)

Problem Determination: Incorrect input parameter option.

System Action: The API call is not processed.

User Response: Check the input parameter option of the API function call. For more information refer to the diagnosis log.

1011 (ERR_INVALID_BUFLEN)

Problem Determination: Incorrect input parameter **Buflen**.

System Action: The API call is not processed.

User Response: Check the input parameter **Buflen** of the API function call. For more information refer to the diagnosis log.

1021 (NO_FIN_MSG)

Problem Determination: The message type is not FIN. Checksum is not mandatory.

System Action: The API call is not processed.

User Response: None.

1022 (ERR_CHKSM_FAILED)

Problem Determination: Checksum of the output message failed.

System Action: The API call is not processed.

User Response: The check of the CHK trailer failed because the CHK trailer is incorrect. The message might be corrupted.

1023 (ERR_MSG_FORMAT)

Problem Determination: Format error because braces are not in pairs or because the trailer is incorrect.

System Action: The API call is not processed.

User Response: Check the message format.

Appendix B. Message Header Checking

The API includes functions to check that the header of a message conforms to certain basic rules of the destination network.

Depending on the value of the **msgnet** field (NET_SWIFT or NET_TELEX), the rules of the respective network are applied. If the value of the **msgnet** field contains NET_OWN, no header checking is performed.

The header checking performed by the API is not intended as a comprehensive check of the header's validity; a message passing these checks may still be rejected by the network.

S.W.I.F.T. Rules

The following rules are checked for SWIFT message headers:

1. Each message must start with a basic header block, consisting of:
   Block identifier: **{1:**
   Application identifier: **F**, **A**, or **L**
   Data unit identifier: two digits
   SWIFT LT *address*: customized LT
   Session number: four digits (optional)
   Sequence number: six digits (optional)
   Block end: **}**
2. If the application identifier is **F**, the basic header block is followed by an application header, consisting of:
   Block identifier: **{2:**
   Direction identifier: **I** or **O**
   With direction identifier **I**:
   - Message type: three digits
   - Recipient's address:
     - Bank code: four characters
     - Country code: two characters
     - Location code: two characters
     - Logical terminal code: one alphanumeric character
     - Branch code: three alphanumeric characters
   - Message priority: **S**, **U**, or **N**
   - Delivery monitoring: **1**, **2**, or **3**
   - Obsolescence period: three digits

   With direction identifier **O**:
   - Message type: three digits
   - Input time: four digits
   - Message input reference: 28 alphanumeric characters
   - Output date: six digits
   - Output time: four digits
   - Message priority: **S**, **U**, or **N**.

   Block end: **}**

Telex Rules

The fields checked for telex message headers are described in “Telex Header (TX_HEADER)” on page 11.

Appendix C. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Deutschland
Informationssysteme GmbH
Department 3982
Pascalstrasse 100

70569 Stuttgart
Germany

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement or any equivalent agreement between us.

The following paragraph does apply to the US only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

- Advanced Peer-to-Peer Networking
- AIX
- APPN
- C/370
- CICS
- CICS/ESA
- CICS/MVS
- CICS/VSE
- DB2
- Distributed Relational Database Architecture
- DRDA
- eNetwork
- IBM
- IMS/ESA
- Language Environment
- MQSeries

- MVS
- MVS/ESA
- MVS/XA
- OS/2
- OS/390
- RACF
- VSE/ESA
- VTAM

Workstation (AWS) and Directory Services Application (DSA) are trademarks of S.W.I.F.T., La Hulpe in Belgium.

Pentium is a trademark of Intel Corporation.

PC Direct is a trademark of Ziff Communications Company in the United States, other countries, or both, and is used by IBM Corporation under license.

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Glossary of Terms and Abbreviations

This glossary defines terms as they are used in this book. If you do not find the terms you are looking for, refer to the *IBM Dictionary of Computing*, New York: McGraw-Hill, and the *S.W.I.F.T. User Handbook*.

A

ACB. Access method control block.

ACC. MERVA Link USS application control command application. It provides a means of operating MERVA Link USS in USS shell and MVS batch environments.

Access method control block (ACB). A control block that links an application program to VSAM or VTAM.

ACD. MERVA Link USS application control daemon.

ACT. MERVA Link USS application control table.

address. See *SWIFT address*.

address expansion. The process by which the full name of a financial institution is obtained using the SWIFT address, telex correspondent's address, or a nickname.

AMPDU. Application message protocol data unit, which is defined in the MERVA Link P1 protocol, and consists of an envelope and its content.

answerback. In telex, the response from the dialed correspondent to the WHO R U signal.

answerback code. A group of up to 6 letters following or contained in the answerback. It is used to check the answerback.

APC. Application control.

API. Application programming interface.

APPC. Advanced Program-to-Program Communication based on SNA LU 6.2 protocols.

APPL. A VTAM definition statement used to define a VTAM application program.

application programming interface (API). An interface that programs can use to exchange data.

application support filter (ASF). In MERVA Link, a user-written program that can control and modify any data exchanged between the Application Support Layer and the Message Transfer Layer.

application support process (ASP). An executing instance of an application support program. Each application support process is associated with an ASP entry in the partner table. An ASP that handles outgoing messages is a *sending ASP*; one that handles incoming messages is a *receiving ASP*.

application support program (ASP). In MERVA Link, a program that exchanges messages and reports with a specific remote partener ASP. These two programs must agree on which conversation protocol they are to use.

ASCII. American Standard Code for Information Interchange. The standard code, using a coded set consisting of 7-bit coded characters (8 bits including parity check), used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

ASF. Application support filter.

ASF. (1) Application support process. (2) Application support program.

ASPDU. Application support protocol data unit, which is defined in the MERVA Link P2 protocol.

authentication. The SWIFT security check used to ensure that a message has not changed during transmission, and that it was sent by an authorized sender.

authenticator key. A set of alphanumeric characters used for the authentication of a message sent via the SWIFT network.

authenticator-key file. The file that stores the keys used during the authentication of a message. The file contains a record for each of your financial institution's correspondents.

B

Back-to-Back (BTB). A MERVA Link function that enables ASPs to exchange messages in the local MERVA Link node without using data communication services.

bank identifier code. A 12-character code used to identify a bank within the SWIFT network. Also called a SWIFT address. The code consists of the following subcodes:
- The bank code (4 characters)
- The ISO country code (2 characters)
- The location code (2 characters)
- The address extension (1 character)

- The branch code (3 characters) for a SWIFT user institution, or the letters “BIC” for institutions that are not SWIFT users.

Basic Security Manager (BSM). A component of VSE/ESA Version 2.4 that is invoked by the System Authorization Facility, and used to ensure signon and transaction security.

BIC. Bank identifier code.

BIC Bankfile. A tape of bank identifier codes supplied by S.W.I.F.T.

BIC Database Plus Tape. A tape of financial institutions and currency codes, supplied by S.W.I.F.T. The information is compiled from various sources and includes national, international, and cross-border identifiers.

BIC Directory Update Tape. A tape of bank identifier codes and currency codes, supplied by S.W.I.F.T., with extended information as published in the printed BIC Directory.

body. The second part of an IM-ASPDU. It contains the actual application data or the message text that the IM-AMPDU transfers.

BSC. Binary synchronous control.

BSM. Basic Security Manager.

BTB. Back-to-back.

buffer. A storage area used by MERVA programs to store a message in its internal format. A buffer has an 8-byte prefix that indicates its length.

C

CBT. SWIFT computer-based terminal.

CCSID. Coded character set identifier.

CDS. Control data set.

central service. In MERVA, a service that uses resources that either require serialization of access, or are only available in the MERVA nucleus.

CF message. Confirmed message. When a sending MERVA Link system is informed of the successful delivery of a message to the receiving application, it routes the delivered application messages as CF messages, that is, messages of class CF, to an ACK wait queue or to a complete message queue.

COA. Confirm on arrival.

COD. Confirm on delivery.

coded character set identifier (CCSID). The name of a coded set of characters and their code point assignments.

commit. In MQSeries, to commit operations is to make the changes on MQSeries queues permanent. After putting one or more messages to a queue, a commit makes them visible to other programs. After getting one or more messages from a queue, a commit permanently deletes them from the queue.

confirm-on-arrival (COA) report. An MQSeries report message type created when a message is placed on that queue. It is created by the queue manager that owns the destination queue.

confirm-on-delivery (COD) report. An MQSeries report message type created when an application retrieves a message from the queue in a way that causes the message to be deleted from the queue. It is created by the queue manager.

control fields. In MERVA Link, fields that are part of a MERVA message on the queue data set and of the message in the TOF. Control fields are written to the TOF at nesting identifier 0. Messages in SWIFT format do not contain control fields.

correspondent. An institution to which your institution sends and from which it receives messages.

correspondent identifier. The 11-character identifier of the receiver of a telex message. Used as a key to retrieve information from the Telex correspondents file.

cross-system coupling facility. See *XCF*.

coupling services. In a sysplex, the functions of XCF that transfer data and status information among the members of a group that reside in one or more of the MVS systems in the sysplex.

couple data set. See *XCF couple data set*.

CTP. MERVA Link command transfer processor.

currency code file. A file containing the currency codes, together with the name, fraction length, country code, and country names.

D

daemon. A long-lived process that runs unattended to perform continuous or periodic systemwide functions.

DASD. Direct access storage device.

data area. An area of a predefined length and format on a panel in which data can be entered or displayed. A field can consist of one or more data areas.

data element. A unit of data that, in a certain context, is considered indivisible. In MERVA Link, a data element consists of a 2-byte data element length field, a 2-byte data-element identifier field, and a field of variable length containing the data element data.

datagram. In TCP/IP, the basic unit of information passed across the Internet environment. This type of message does not require a reply, and is the simplest type of message that MQSeries supports.

data terminal equipment. That part of a data station that serves as a data source, data link, or both, and provides for the data communication control function according to protocols.

DB2. A family of IBM licensed programs for relational database management.

dead-letter queue. A queue to which a queue manager or application sends messages that it cannot deliver. Also called *undelivered-message queue*.

dial-up number. A series of digits required to establish a connection with a remote correspondent via the public telex network.

direct service. In MERVA, a service that uses resources that are always available and that can be used by several requesters at the same time.

display mode. The mode (PROMPT or NOPROMPT) in which SWIFT messages are displayed. See *PROMPT mode* and *NOPROMPT mode.*

distributed queue management (DQM). In MQSeries message queuing, the setup and control of message channels to queue managers on other systems.

DQM. Distributed queue management.

DTE. Data terminal equipment.

E

EBCDIC. Extended Binary Coded Decimal Interchange Code. A coded character set consisting of 8-bit coded characters.

ECB. Event control block.

EDIFACT. Electronic Data Interchange for Administration, Commerce and Transport (a United Nations standard).

ESM. External security manager.

EUD. End-user driver.

exception report. An MQSeries report message type that is created by a message channel agent when a message is sent to another queue manager, but that message cannot be delivered to the specified destination queue.

external line format (ELF) messages. Messages that are not fully tokenized, but are stored in a single field in the TOF. Storing messages in ELF improves performance, because no mapping is needed, and checking is not performed.

external security manager (ESM). A security product that is invoked by the System Authorization Facility. RACF is an example of an ESM.

F

FDT. Field definition table.

field. In MERVA, a portion of a message used to enter or display a particular type of data in a predefined format. A field is located by its position in a message and by its tag. A field is made up of one or more data areas. See also *data area*.

field definition table (FDT). The field definition table describes the characteristics of a field; for example, its length and number of its data areas, and whether it is mandatory. If the characteristics of a field change depending on its use in a particular message, the definition of the field in the FDT can be overridden by the MCB specifications.

field group. One or several fields that are defined as being a group. Because a field can occur more than once in a message, field groups are used to distinguish them. A name can be assigned to the field group during message definition.

field group number. In the TOF, a number is assigned to each field group in a message in ascending order from 1 to 255. A particular field group can be accessed using its field group number.

field tag. A character string used by MERVA to identify a field in a network buffer. For example, for SWIFT field 30, the field tag is **:30:**.

FIN. Financial application.

FIN-Copy. The MERVA component used for SWIFT FIN-Copy support.

finite state machine. The theoretical base describing the rules of a service request's state and the conditions to state transitions.

FMT/ESA. MERVA-to-MERVA Financial Message Transfer/ESA.

form. A partially-filled message containing data that can be copied for a new message of the same message type.

G

GPA. General purpose application.

H

HFS. Hierarchical file system.

hierarchical file system (HFS). A system for organizing files in a hierarchy, as in a UNIX system. OS/390 UNIX System Services files are organized in an HFS. All files are members of a directory, and each directory is in turn a member of a directory at a higher level in the HFS. The highest level in the hierarchy is the root directory.

I

IAM. Interapplication messaging (a MERVA Link message exchange protocol).

IM-ASPDU. Interapplication messaging application support protocol data unit. It contains an application message and consists of a heading and a body.

incore request queue. Another name for the request queue to emphasize that the request queue is held in memory instead of on a DASD.

InetD. Internet Daemon. It provides TCP/IP communication services in the OS/390 USS environment.

initiation queue. In MQSeries, a local queue on which the queue manager puts trigger messages.

input message. A message that is input into the SWIFT network. An input message has an input header.

INTERCOPE TelexBox. This telex box supports various national conventions for telex procedures and protocols.

interservice communication. In MERVA ESA, a facility that enables communication among services if MERVA ESA is running in a multisystem environment.

intertask communication. A facility that enables application programs to communicate with the MERVA nucleus and so request a central service.

IP. Internet Protocol.

IP message. In-process message. A message that is in the process of being transferred to another application.

ISC. Intersystem communication.

ISN. Input sequence number.

ISN acknowledgment. A collective term for the various kinds of acknowledgments sent by the SWIFT network.

ISO. International Organization for Standardization.

ITC. Intertask communication.

J

JCL. Job control language.

journal. A chronological list of records detailing MERVA actions.

journal key. A key used to identify a record in the journal.

journal service. A MERVA central service that maintains the journal.

K

KB. Kilobyte (1024 bytes).

key. A character or set of characters used to identify an item or group of items. For example, the user ID is the key to identify a user file record.

key-sequenced data set (KSDS). A VSAM data set whose records are loaded in key sequence and controlled by an index.

keyword parameter. A parameter that consists of a keyword, followed by one or more values.

KSDS. Key-sequenced data set.

L

LAK. Login acknowledgment message. This message informs you that you have successfully logged in to the SWIFT network.

large message. A message that is stored in the large message cluster (LMC). The maximum length of a message to be stored in the VSAM QDS is 31900 bytes. Messages up to 2MB can be stored in the LMC. For queue management using DB2 no distinction is made between messages and large messages.

large queue element. A queue element that is larger than the smaller of:

- The limiting value specified during the customization of MERVA
- 32KB

LC message. Last confirmed control message. It contains the message-sequence number of the application or acknowledgment message that was last confirmed; that is, for which the sending MERVA Link system most recently received confirmation of a successful delivery.

LDS. Logical data stream.

LMC. Large message cluster.

LNK. Login negative acknowledgment message. This message indicates that the login to the SWIFT network has failed.

local queue. In MQSeries, a queue that belongs to a local queue manager. A local queue can contain a list of messages waiting to be processed. Contrast with *remote queue*.

local queue manager. In MQSeries, the queue manager to which the program is connected, and that provides message queuing services to that program. Queue managers to which a program is not connected are remote queue managers, even if they are running on the same system as the program.

login. To start the connection to the SWIFT network.

LR message. Last received control message, which contains the message-sequence number of the application or acknowledgment message that was last received from the partner application.

LSN. Login sequence number.

LT. See *LTERM*.

LTC. Logical terminal control.

LTERM. Logical terminal. Logical terminal names have 4 characters in CICS and up to 8 characters in IMS.

LU. A VTAM logical unit.

M

maintain system history program (MSHP). A program used for automating and controlling various installation, tailoring, and service activities for a VSE system.

MCA. Message channel agent.

MCB. Message control block.

MERVA ESA. The IBM licensed program Message Entry and Routing with Interfaces to Various Applications for ESA.

MERVA Link. A MERVA component that can be used to interconnect several MERVA systems.

message. A string of fields in a predefined form used to provide or request information. See also *SWIFT financial message.*

message body. The part of the message that contains the message text.

message category. A group of messages that are logically related within an application.

message channel. In MQSeries distributed message queuing, a mechanism for moving messages from one queue manager to another. A message channel comprises two message channel agents (a sender and a receiver) and a communication link.

message channel agent (MCA). In MQSeries, a program that transmits prepared messages from a transmission queue to a communication link, or from a communication link to a destination queue.

message control block (MCB). The definition of a message, screen panel, net format, or printer layout made during customization of MERVA.

Message Format Service (MFS). A MERVA direct service that formats a message according to the medium to be used, and checks it for formal correctness.

message header. The leading part of a message that contains the sender and receiver of the message, the message priority, and the type of message.

Message Integrity Protocol (MIP). In MERVA Link, the protocol that controls the exchange of messages between partner ASPs. This protocol ensures that any loss of a message is detected and reported, and that no message is duplicated despite system failures at any point during the transfer process.

message-processing function. The various parts of MERVA used to handle a step in the message-processing route, together with any necessary equipment.

message queue. See *queue*.

Message Queue Interface (MQI). The programming interface provided by the MQSeries queue managers. It provides a set of calls that let application programs access message queuing services such as sending messages, receiving messages, and manipulating MQSeries objects.

Message Queue Manager (MQM). An IBM licensed program that provides message queuing services. It is part of the MQSeries set of products.

message reference number (MRN). A unique 16-digit number assigned to each message for identification purposes. The message reference number consists of an 8-digit domain identifier that is followed by an 8-digit sequence number.

message sequence number (MSN). A sequence number for messages transferred by MERVA Link.

message type (MT). A number, up to 7 digits long, that identifies a message. SWIFT messages are identified by a 3-digit number; for example SWIFT message type MT S100.

MFS. Message Format Service.

MIP. Message Integrity Protocol.

MPDU. Message protocol data unit, which is defined in P1.

MPP. In IMS, message-processing program.

MQA. MQ Attachment.

MQ Attachment (MQA). A MERVA feature that provides message transfer between MERVA and a user-written MQI application.

MQH. MQSeries queue handler.

MQI. Message queue interface.

MQM. Message queue manager.

MQS. MQSeries nucleus server.

MQSeries. A family of IBM licensed programs that provides message queuing services.

MQSeries nucleus server (MQS). A MERVA component that listens for messages on an MQI queue, receives them, extracts a service request, and passes it via the request queue handler to another MERVA ESA instance for processing.

MQSeries queue handler (MQH). A MERVA component that performs service calls to the Message Queue Manager via the provided Message Queue Interface.

MRN. Message reference number.

MSC. MERVA system control facility.

MSHP. Maintain system history program.

MSN. Message sequence number.

MT. Message type.

MTP. (1) Message transfer program. (2) Message transfer process.

MTS. Message Transfer System.

MTSP. Message Transfer Service Processor.

MTT. Message type table.

multisystem application. (1) An application program that has various functions distributed across MVS systems in a multisystem environment. (2) In XCF, an authorized application that uses XCF coupling services. (3) In MERVA ESA, multiple instances of MERVA ESA that are distributed among different MVS systems in a multisystem environment.

multisystem environment. An environment in which two or more MVS systems reside on one or more processors, and programs on one system can communicate with programs on the other systems. With XCF, the environment in which XCF services are available in a defined sysplex.

multisystem sysplex. A sysplex in which one or more MVS systems can be initialized as part of the sysplex. In a multisystem sysplex, XCF provides coupling services on all systems in the sysplex and requires an XCF couple data set that is shared by all systems. See also *single-system sysplex*.

MVS/ESA. Multiple Virtual Storage/Enterprise Systems Architecture.

N

namelist. An MQSeries for MVS/ESA object that contains a list of queue names.

nested message. A message that is composed of one or more message types.

nested message type. A message type that is contained in another message type. In some cases, only part of a message type (for example, only the mandatory fields) is nested, but this “partial” nested message type is also considered to be nested. For example, SWIFT MT 195 could be used to request information about a SWIFT MT 100 (customer transfer). The SWIFT MT 100 (or at least its mandatory fields) is then nested in SWIFT MT 195.

nesting identifier. An identifier (a number from 2 to 255) that is used to access a nested message type.

network identifier. A single character that is placed before a message type to indicate which network is to be used to send the message; for example, **S** for SWIFT

network service access point (NSAP). The endpoint of a network connection used by the SWIFT transport layer.

NOPROMPT mode. One of two ways to display a message panel. NOPROMPT mode is only intended for experienced SWIFT Link users who are familiar with the structure of SWIFT messages. With NOPROMPT mode, only the SWIFT header, trailer, and pre-filled fields and their tags are displayed. Contrast with *PROMPT mode*.

NSAP. Network service access point.

nucleus server. A MERVA component that processes a service request as selected by the request queue handler. The service a nucleus server provides and the way it provides it is defined in the nucleus server table (DSLNSVT).

O

object. In MQSeries, objects define the properties of queue managers, queues, process definitions, and namelists.

occurrence. See *repeatable sequence*.

option. One or more characters added to a SWIFT field number to distinguish among different layouts for and meanings of the same field. For example, SWIFT field 60 can have an option F to identify a first opening balance, or M for an intermediate opening balance.

origin identifier (origin ID). A 34-byte field of the MERVA user file record. It indicates, in a MERVA and SWIFT Link installation that is shared by several banks, to which of these banks the user belongs. This lets the user work for that bank only.

OSN. Output sequence number.

OSN acknowledgment. A collective term for the various kinds of acknowledgments sent to the SWIFT network.

output message. A message that has been received from the SWIFT network. An output message has an output header.

P

P1. In MERVA Link, a peer-to-peer protocol used by cooperating message transfer processes (MTPs).

P2. In MERVA Link, a peer-to-peer protocol used by cooperating application support processes (ASPs).

P3. In MERVA Link, a peer-to-peer protocol used by cooperating command transfer processors (CTPs).

packet switched public data network (PSPDN). A public data network established and operated by network common carriers or telecommunication administrations for providing packet-switched data transmission.

panel. A formatted display on a display terminal. Each page of a message is displayed on a separate panel.

parallel processing. The simultaneous processing of units of work by several servers. The units of work can be either transactions or subdivisions of larger units of work.

parallel sysplex. A sysplex that uses one or more coupling facilities.

partner table (PT). In MERVA Link, the table that defines how messages are processed. It consists of a header and different entries, such as entries to specify the message-processing parameters of an ASP or MTP.

PCT. Program Control Table (of CICS).

PDE. Possible duplicate emission.

PDU. Protocol data unit.

PF key. Program-function key.

positional parameter. A parameter that must appear in a specified location relative to other parameters.

PREMIUM. The MERVA component used for SWIFT PREMIUM support.

process definition object. An MQSeries object that contains the definition of an MQSeries application. A queue manager uses the definitions contained in a process definition object when it works with trigger messages.

program-function key. A key on a display terminal keyboard to which a function (for example, a command) can be assigned. This lets you execute the function (enter the command) with a single keystroke.

PROMPT mode. One of two ways to display a message panel. PROMPT mode is intended for SWIFT Link users who are unfamiliar with the structure of SWIFT messages. With PROMPT mode, all the fields and tags are displayed for the SWIFT message. Contrast with *NOPROMPT mode*.

protocol data unit (PDU). In MERVA Link a PDU consists of a structured sequence of implicit and explicit data elements:
- Implicit data elements contain other data elements.
- Explicit data elements cannot contain any other data elements.

PSN. Public switched network.

PSPDN. Packet switched public data network.

PSTN. Public switched telephone network.

PT. Partner table.

PTT. A national post and telecommunication authority (post, telegraph, telephone).

Q

QDS. Queue data set.

QSN. Queue sequence number.

queue. (1) In MERVA, a logical subdivision of the MERVA queue data set used to store the messages associated with a MERVA message-processing function. A queue has the same name as the message-processing function with which it is associated. (2) In MQSeries, an object onto which message queuing applications can put messages, and from which they can get messages. A queue is owned and maintained by a queue manager. See also *request queue*.

queue element. A message and its related control information stored in a data record in the MERVA ESA Queue Data Set.

queue management. A MERVA service function that handles the storing of messages in, and the retrieval of messages from, the queues of message-processing functions.

queue manager. (1) An MQSeries system program that provides queueing services to applications. It provides an application programming interface so that programs can access messages on the queues that the queue manager owns. See also *local queue manager* and *remote queue manager*. (2) The MQSeries object that defines the attributes of a particular queue manager.

queue sequence number (QSN). A sequence number that is assigned to the messages stored in a logical queue by MERVA ESA queue management in ascending order. The QSN is always unique in a queue. It is reset to zero when the queue data set is formatted, or when a queue management restart is carried out and the queue is empty.

R

RACF. Resource Access Control Facility.

RBA. Relative byte address.

RC message. Recovered message; that is, an IP message that was copied from the control queue of an inoperable or closed ASP via the **recover** command.

ready queue. A MERVA queue used by SWIFT Link to collect SWIFT messages that are ready for sending to the SWIFT network.

remote queue. In MQSeries, a queue that belongs to a remote queue manager. Programs can put messages on remote queues, but they cannot get messages from remote queues. Contrast with *local queue*.

remote queue manager. In MQSeries, a queue manager is remote to a program if it is not the queue manager to which the program is connected.

repeatable sequence. A field or a group of fields that is contained more than once in a message. For example, if the SWIFT fields 20, 32, and 72 form a sequence, and if this sequence can be repeated up to 10 times in a message, each sequence of the fields 20, 32, and 72 would be an occurrence of the repeatable sequence.

In the TOF, the occurrences of a repeatable sequence are numbered in ascending order from 1 to 32767 and can be referred to using the occurrence number.

A repeatable sequence in a message may itself contain another repeatable sequence. To identify an occurrence within such a nested repeatable sequence, more than one occurrence number is necessary.

reply message. In MQSeries, a type of message used for replies to request messages.

reply-to queue. In MQSeries, the name of a queue to which the program that issued an MQPUT call wants a reply message or report message sent.

report message. In MQSeries, a type of message that gives information about another message. A report message usually indicates that the original message cannot be processed for some reason.

request message. In MQSeries, a type of message used for requesting a reply from another program.

request queue. The queue in which a service request is stored. It resides in main storage and consists of a set of request queue elements that are chained in different queues:

- Requests waiting to be processed
- Requests currently being processed
- Requests for which processing has finished

request queue handler (RQH). A MERVA ESA component that handles the queueing and scheduling of service requests. It controls the request processing of a nucleus server according to rules defined in the finite state machine.

Resource Access Control Facility (RACF). An IBM licensed program that provides for access control by identifying and verifying users to the system, authorizing access to protected resources, logging detected unauthorized attempts to enter the system, and logging detected accesses to protected resources.

retype verification. See *verification*.

routing. In MERVA, the passing of messages from one stage in a predefined processing path to the next stage.

RP. Regional processor.

RQH. Request queue handler.

RRDS. Relative record data set.

S

SAF. System Authorization Facility.

SCS. SNA character string

SCP. System control process.

SDI. Sequential data set input. A batch utility used to import messages from a sequential data set or a tape into MERVA ESA queues.

SDO. Sequential data set output. A batch utility used to export messages from a MERVA ESA queue to a sequential data set or a tape.

SDY. Sequential data set system printer. A batch utility used to print messages from a MERVA ESA queue.

service request. A type of request that is created and passed to the request queue handler whenever a nucleus server requires a service that is not currently available.

sequence number. A number assigned to each message exchanged between two nodes. The number is increased by one for each successive message. It starts from zero each time a new session is established.

sign off. To end a session with MERVA.

sign on. To start a session with MERVA.

single-system sysplex. A sysplex in which only one MVS system can be initialized as part of the sysplex. In a single-system sysplex, XCF provides XCF services on the system, but does not provide signalling services between MVS systems. A single-system sysplex requires an XCF couple data set. See also *multisystem sysplex*.

small queue element. A queue element that is smaller than the smaller of:

- The limiting value specified during the customization of MERVA
- 32KB

SMP/E. System Modification Program Extended.

SN. Session number.

SNA. Systems network architecture.

SNA character string. In SNA, a character string composed of EBCDIC controls, optionally mixed with user data, that is carried within a request or response unit.

SPA. Scratch pad area.

SQL. Structured Query Language.

SR-ASPDU. The status report application support PDU, which is used by MERVA Link for acknowledgment messages.

SSN. Select sequence number.

subfield. A subdivision of a field with a specific meaning. For example, the SWIFT field 32 has the subfields date, currency code, and amount. A field can have several subfield layouts depending on the way the field is used in a particular message.

SVC. (1) Switched Virtual Circuit. (2) Supervisor call instruction.

S.W.I.F.T. (1) Society for Worldwide Interbank Financial Telecommunication s.c. (2) The network provided and managed by the Society for Worldwide Interbank Financial Telecommunication s.c.

SWIFT address. Synonym for *bank identifier code*.

SWIFT Correspondents File. The file containing the bank identifier code (BIC), together with the name, postal address, and zip code of each financial institution in the BIC Directory.

SWIFT financial message. A message in one of the SWIFT categories 1 to 9 that you can send or receive via the SWIFT network. See *SWIFT input message* and *SWIFT output message*.

SWIFT header. The leading part of a message that contains the sender and receiver of the message, the message priority, and the type of message.

SWIFT input message. A SWIFT message with an input header to be sent to the SWIFT network.

SWIFT link. The MERVA ESA component used to link to the SWIFT network.

SWIFT network. Refers to the SWIFT network of the Society for Worldwide Interbank Financial Telecommunication (S.W.I.F.T.).

SWIFT output message. A SWIFT message with an output header coming from the SWIFT network.

SWIFT system message. A SWIFT general purpose application (GPA) message or a financial application (FIN) message in SWIFT category 0.

switched virtual circuit (SVC). An X.25 circuit that is dynamically established when needed. It is the X.25 equivalent of a switched line.

sysplex. One or more MVS systems that communicate and cooperate via special multisystem hardware components and software services.

System Authorization Facility (SAF). An MVS or VSE facility through which MERVA ESA communicates with an external security manager such as RACF (for MVS) or the basic security manager (for VSE).

System Control Process (SCP). A MERVA Link component that handles the transfer of MERVA ESA commands to a partner MERVA ESA system, and the receipt of the command response. It is associated with a system control process entry in the partner table.

System Modification Program Extended (SMP/E). A licensed program used to install software and software changes on MVS systems.

Systems Network Architecture (SNA). The description of the logical structure, formats, protocols, and operating sequences for transmitting information units through, and for controlling the configuration and operation of, networks.

T

tag. A field identifier.

TCP/IP. Transmission Control Protocol/Internet Protocol.

Telex Correspondents File. A file that stores data about correspondents. When the user enters the corresponding nickname in a Telex message, the corresponding information in this file is automatically retrieved and entered into the Telex header area.

telex header area. The first part of the telex message. It contains control information for the telex network.

telex interface program (TXIP). A program that runs on a Telex front-end computer and provides a communication facility to connect MERVA ESA with the Telex network.

Telex Link. The MERVA ESA component used to link to the public telex network via a Telex substation.

Telex substation. A unit comprised of the following:

- Telex Interface Program
- A Telex front-end computer
- A Telex box

Terminal User Control Block (TUCB). A control block containing terminal-specific and user-specific information used for processing messages for display devices such as screen and printers.

test key. A key added to a telex message to ensure message integrity and authorized delivery. The test key is an integer value of up to 16 digits, calculated manually or by a test-key processing program using the significant information in the message, such as amounts, currency codes, and the message date.

test-key processing program. A program that automatically calculates and verifies a test key. The Telex Link supports panels for input of test-key-related data and an interface for a test-key processing program.

TFD. Terminal feature definitions table.

TID. Terminal identification. The first 9 characters of a bank identifier code (BIC).

TOF. Originally the abbreviation of *tokenized form*, the TOF is a storage area where messages are stored so that their fields can be accessed directly by their field names and other index information.

TP. Transaction program.

transaction. A specific set of input data that triggers the running of a specific process or job; for example, a message destined for an application program.

transaction code. In IMS and CICS, an alphanumeric code that calls an IMS message processing program or a CICS transaction. Transaction codes have 4 characters in CICS and up to 8 characters in IMS.

Transmission Control Protocol/Internet Protocol (TCP/IP). A set of communication protocols that support peer-to-peer connectivity functions for both local and wide area networks.

transmission queue. In MQSeries, a local queue on which prepared messages destined for a remote queue manager are temporarily stored.

trigger event. In MQSeries, an event (such as a message arriving on a queue) that causes a queue manager to create a trigger message on an initiation queue.

trigger message. In MQSeries, a message that contains information about the program that a trigger monitor is to start.

trigger monitor. In MQSeries, a continuously-running application that serves one or more initiation queues. When a trigger message arrives on an initiation queue, the trigger monitor retrieves the message. It uses the information in the trigger message to start a process that serves the queue on which a trigger event occurred.

triggering. In MQSeries, a facility that allows a queue manager to start an application automatically when predetermined conditions are satisfied.

TUCB. Terminal User Control Block.

TXIP. Telex interface program.

U

UMR. Unique message reference.

unique message reference (UMR). An optional feature of MERVA ESA that provides each message with a unique identifier the first time it is placed in a queue. It is composed of a MERVA ESA installation name, a sequence number, and a date and time stamp.

UNIT. A group of related literals or fields of an MCB definition, or both, enclosed by a DSLLUNIT and DSLLUEND macroinstruction.

UNIX System Services (USS). A component of OS/390, formerly called OpenEdition (OE), that creates a UNIX environment that conforms to the XPG4 UNIX 1995 specifications, and provides two open systems interfaces on the OS/390 operating system:

- An application program interface (API)
- An interactive shell interface

UN/EDIFACT. United Nations Standard for Electronic Data Interchange for Administration, Commerce and Transport.

USE. S.W.I.F.T. User Security Enhancements.

user file. A file containing information about all MERVA ESA users; for example, which functions each user is allowed to access. The user file is encrypted and can only be accessed by authorized persons.

user identification and verification. The acts of identifying and verifying a RACF-defined user to the system during logon or batch job processing. RACF identifies the user by the user ID and verifies the user by the password or operator identification card supplied during logon processing or the password supplied on a batch JOB statement.

USS. UNIX System Services.

V

verification. Checking to ensure that the contents of a message are correct. Two kinds of verification are:

- Visual verification: you read the message and confirm that you have done so
- Retype verification: you reenter the data to be verified

Virtual LU. An LU defined in MERVA Extended Connectivity for communication between MERVA and MERVA Extended Connectivity.

Virtual Storage Access Method (VSAM). An access method for direct or sequential processing of fixed and variable-length records on direct access devices. The records in a VSAM data set or file can be organized in logical sequence by a key field (key sequence), in the physical sequence in which they are written on the data set or file (entry sequence), or by relative-record number.

Virtual Telecommunications Access Method (VTAM). An IBM licensed program that controls communication and the flow of data in an SNA network. It provides single-domain, multiple-domain, and interconnected network capability.

VSAM. Virtual Storage Access Method.

VTAM. Virtual Telecommunications Access Method (IBM licensed program).

W

Windows NT service. A type of Windows NT application that can run in the background of the Windows NT operating system even when no user is logged on. Typically, such a service has no user interaction and writes its output messages to the Windows NT event log.

X

X.25. An ISO standard for interface to packet switched communications services.

XCF. Abbreviation for *cross-system coupling facility*, which is a special logical partition that provides high-speed caching, list processing, and locking functions in a sysplex. XCF provides the MVS coupling services that allow authorized programs on MVS systems in a multisystem environment to communicate with (send data to and receive data from) authorized programs on other MVS systems.

XCF couple data sets. A data set that is created through the XCF couple data set format utility and, depending on its designated type, is shared by some or all of the MVS systems in a sysplex. It is accessed only by XCF and contains XCF-related data about the sysplex, systems, applications, groups, and members.

XCF group. The set of related members defined to SCF by a multisystem application in which members of the group can communicate with (send data to and receive data from) other members of the same group. All MERVA systems working together in a sysplex must pertain to the same XCF group.

XCF member. A specific function of a multisystem application that is defined to XCF and assigned to a group by the multisystem application. A member resides on one system in a sysplex and can use XCF services to communicate with other members of the same group.

Bibliography

MERVA ESA Publications

- *MERVA for ESA Version 4: Application Programming Interface Guide*, SH12-6374
- *MERVA for ESA Version 4: Advanced MERVA Link*, SH12-6390
- *MERVA for ESA Version 4: Concepts and Components*, SH12-6381
- *MERVA for ESA Version 4: Customization Guide*, SH12-6380
- *MERVA for ESA Version 4: Diagnosis Guide*, SH12-6382
- *MERVA for ESA Version 4: Installation Guide*, SH12-6378
- *MERVA for ESA Version 4: Licensed Program Specifications*, GH12-6373
- *MERVA for ESA Version 4: Macro Reference*, SH12-6377
- *MERVA for ESA Version 4: Messages and Codes*, SH12-6379
- *MERVA for ESA Version 4: Operations Guide*, SH12-6375
- *MERVA for ESA Version 4: System Programming Guide*, SH12-6366
- *MERVA for ESA Version 4: User's Guide*, SH12-6376

MERVA ESA Components Publications

- *MERVA Automatic Message Import/Export Facility: User's Guide*, SH12-6389
- *MERVA Connection/NT*, SH12-6339
- *MERVA Connection/400*, SH12-6340
- *MERVA Directory Services*, SH12-6367
- *MERVA Extended Connectivity: Installation and User's Guide*, SH12-6157
- *MERVA Message Processing Client for Windows NT: User's Guide*, SH12-6341
- *MERVA Traffic Reconciliation*, SH12-6392
- *MERVA USE: Administration Guide*, SH12-6338
- *MERVA USE & Branch for Windows NT: User's Guide*, SH12-6334
- *MERVA USE & Branch for Windows NT: Installation and Customization Guide*, SH12-6335
- *MERVA USE & Branch for Windows NT: Application Programming Guide*, SH12-6336
- *MERVA USE & Branch for Windows NT: Diagnosis Guide*, SH12-6337
- *MERVA USE & Branch for Windows NT: Migration Guide*, SH12-6393
- *MERVA USE & Branch for Windows NT: Installation and Customization Guide*, SH12-6335
- *MERVA Workstation Based Functions*, SH12-6383

Other IBM Publications

- *DB2 Administration Guide*, S10J-8157
- *DB2 Building Applications for Windows and OS/2 Environment*, S10J-8160
- *DB2 API Reference*, S10J-8167
- *DB2 Troubleshooting Guide*, S10J-8169
- *eNetwork Personal Communications Version 4.2 for Windows 95 and Windows NT Quick Beginnings*, GC31-8476
- *eNetwork Personal Communications Version 4.2 for Windows 95 and Windows NT Reference*, GC31-8477
- *CID Enablement Guidelines*, S10H-9666
- *CICS-RACF Security Guide*, SC33-1185
- *ITSC Redbook APPC Security: MVS/ESA, CICS/ESA, and OS/2*, GG24-3960
- *IMS/ESA Version 4 Data Communication Administration Guide*, SC26-3060
- *MQSeries Application Programming Reference*, SC33-1673

S.W.I.F.T. Publications

The following are published by the Society for Worldwide Interbank Financial Telecommunication, s.c., in La Hulpe, Belgium:

- *S.W.I.F.T. User Handbook*
- *S.W.I.F.T. Dictionary*
- *S.W.I.F.T. FIN Security Guide*
- *S.W.I.F.T. Card Readers User Guide*

Index

Readers' Comments — We'd Like to Hear from You

MERVA ESA Components
MERVA USE & Branch for Windows NT
Application Programming
Version 4 Release 1

Publication No. SH12-6336-02

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name	Address
Company or Organization	
Phone No.	

Readers' Comments — We'd Like to Hear from You
SH12-6336-02

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Deutschland Entwicklung GmbH
Information Development & User Centered Design
Dept. 0446
Schoenaicher Strasse 220
71032 Boeblingen
Germany

Fold and Tape **Please do not staple** Fold and Tape

SH12-6336-02

Cut or Fold
Along Line

Program Number: 5648-B30

SH12-6336-02

Spine information:

MERVA ESA Components

MERVA USE & Branch for Windows NT Application Programming

Version 4
Release 1