MERVA for ESA

IBM

Application Program Interface Guide

Version 4 Release 1

SH12-6374-01

MERVA for ESA

Application Program Interface Guide

Version 4 Release 1

SH12-6374-01

Note!

Before using this information and the product it supports, be sure to read the general information under "Appendix E. Notices" on page 291.

Second Edition, May 2001

This edition applies to Version 4 Release 1 of IBM MERVA for ESA (5648-B29) 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 (abbreviated to API in this book) of the IBM licensed program Message Entry and Routing with Interfaces to Various Applications for ESA Version 4 Release 1, abbreviated to MERVA for ESA. The API is an assembler and high-level language application-programming interface that enables you to access the following MERVA for ESA services:

- Queue Management
- TOF Supervisor
- Message Format Service
- Print Services
- Journal
- User File Services
- Operator Interface

Who Should Read This Book

This book is written for application programmers. You will find technical information along with examples showing coding methods. The type of information in this book requires that you have some familiarity with the concepts of MERVA for ESA and its services:

- Queue Management (including queue elements, queues, and the queue data set)
- Message Format Service (including SWIFT message structures and Message Control Blocks)
- TOF Supervisor (including field reference structure)
- Print Services
- Journal
- User File Services
- Operator Interface

The MERVA for ESA concepts are described in *MERVA for ESA Concepts and Components*.

You must also be familiar with one of the following programming languages:

- Assembler
- C
- COBOL
- PL/I
- REXX

How to Use This Book

The book is made up of a using part, a reference part, and the appendixes.

Before you write your first MERVA for ESA API program, read the first part, especially:

- “Chapter 1. Introduction and Concepts” on page 3

- “Chapter 2. The MERVA ESA API Services” on page 13
- “Chapter 3. Writing an API Program” on page 23

Use the reference part and appendixes as needed. Of particular interest are:

- “Chapter 9. DSLAPI Functions” on page 97, which describes each function, as well as the return codes it issues, and contains a sample code fragment that demonstrates the use of the function.
- “Chapter 8. Data Structures” on page 81, which describes the data structures used by the functions.
- “Appendix B. Sample Programs” on page 203, which lists and describes the sample programs distributed with MERVA for ESA. Not only do they show how various API functions can be used, but they also show how to use the data structures.

Summary of Changes

Note: The MERVA ESA V4.1 API services are fully compatible with API services of MERVA ESA V3.3. Applications that use MERVA ESA V3.3 API services will run unchanged under MERVA ESA V4.1.

This edition of this manual reflects the following differences between the current version of MERVA ESA (Version 4.1) and the previous version (Version 3.3):

API Runtime Environment
You can use the following DSLAPI variables to customize the API runtime environment:

APICQBIN	Enable the use of binary keys
APICQDIR	Enable direct DB2® queue management calls
APICQLAZ	Defer write requests
APICQMIT	Switch DB2 commit on or off
APICQWRB	Suppress the use of the DOUBLE (write-back) indicator
APICMCLR	Clear the internal queue buffer before the (next) message read function
APICMCHK	Switch message checking on or off.

See "Runtime Environment Settings" on page 52 for details.

Enhanced API Functions
The MSGG and MSGP functions now support formatting from and to the internal queue buffer format ('Q').

New API Functions
Six new API functions have been added:

PRTI, PRTL, and PRTT
The PRTx functions can be used to print messages line by line to a system printer or a file. See "PRTI Initialize Printing Environment" on page 148, "PRTL Create a Print Line of a Message" on page 150, and "PRTT Terminate Printing Environment" on page 152 for details.

PUTR The PUTR function restores a message to a specified queue and QSN. See "PUTR Restore a Queue Element" on page 160 for details.

ROUD, ROUN
The ROUD and ROUN functions route a queue element directly from one queue to one or more target queues. See "ROUD Route Queue Element Directly" on page 183 and "ROUN Route Next Queue Element Directly" on page 185 for details.

Batch Utilities Written in REXX
The batch utilities written in REXX are distributed:

- For MVS, in the library **MERVA.SDSLSAM0**
- For VSE, in the sublibrary **MERVA.LIBS** with the extension **PROC**

All of these utilities now support a DSLPRM customization parameter *PRTNAME*, which can be used to print an institution's name in the header of the printout of the utility.

MERVA ESA V4.1 enhances the following batch utilities written in REXX:

DSLBA13R Print the MERVA ESA journal

The records of the following journal entries are printed now formatted:

X'14' Command response

X'19' Routing trace.

DSLBA15R Print the MERVA ESA user file

The new user file field USRUDATS (last sign-on date) is supported.

MERVA ESA V4.1 provides the following new batch utilities written in REXX, all of which run under both MVS and VSE.:

DSLBA17R	Check date fields in the user file
DSLBA50R	Print queue status list
DSLBA51R	Print queue key list
DSLBA52R	Copy or move messages from one queue to another
DSLBA53R	Scan a queue for 'old' messages
DSLSDIR	Sequential data set input
DSLSDLR	Sequential data set load
DSLSDOR	Sequential data set output
DSLSDUR	Sequential data set unload
DSLSDYR	Sequential data set print

See "Appendix C. Batch Utilities in REXX" on page 227 for details. The utilities with names of the form **DSLSDxR** are described in the *MERVA for ESA Operations Guide*.

Field-Level Access Fields
The list of the fields in the MERVA internal structures that can be accessed using the FLDG and FLDP API services has been updated. You can now read the fields of the function table DSLFNTT with the FLDG function. See "Appendix D. Field-Level Access Fields" on page 269 for details.

Part 1. Using the MERVA ESA Application Program Interface

This part describes the concepts of the MERVA ESA application program interface (API), the MERVA ESA services that you can use in your application programs, and various types of programs you might write.

Chapter 1. Introduction and Concepts

MERVA ESA provides an application program interface (API) that allows you to write your own application programs that interact with MERVA ESA. You can write your application programs in:

- Assembler
- COBOL
- C/370™
- PL/I
- REXX

Programs written in Assembler, COBOL, C/370, and PL/I can run in batch, or as background or foreground CICS® or IMS™ transactions. You can also use these languages to write exit routines for MERVA message formatting services (MFS) that use API services to modify the way MERVA processes messages.

Additionally, you can write batch programs (for MVS™ and VSE) in REXX, which lets you prototype applications quickly, and provides powerful string and mathematical functions.

The MERVA ESA API lets you call the following MERVA services from your application program:

- Queue management services, which you use to manage queue elements (queued messages)
- TOF (tokenized form) services, which manipulate fields in messages
- Message formatting services, which map messages between the MERVA internal form and external formats
- Print services to print messages line by line
- Journal services, which read and write records in the MERVA journal
- User file services, with which you can read records from the MERVA ESA user file
- The operator command service, which lets you execute MERVA operator commands and to inspect the response
- The WTO service, which sends messages to the operator and to the MERVA display message table
- Additional services to facilitate programming S.W.I.F.T and EDI messages

MERVA ESA is distributed with a number of sample programs showing how these API services can be used in each of the supported programming languages. There are sample batch programs, sample foreground and background transactions, and sample MFS exit routines.

Some of the most important MERVA ESA API concepts are:

- The interface working storage area (INTWSTOR)
- Parameters for the API services
- Buffers used to move data to and from an application
- How to call DSLAPI
- How to process messages using the API

The MERVA ESA API is implemented by the MERVA DSLAPI program. To use MERVA API services, invoke DSLAPI from your application using standard OS calling conventions:

1. Prepare the various parameters required by the API service you want to use.
2. Call DSLAPI passing it the parameters in a parameter list.
3. When control returns to your program, inspect return codes and any other results of the call.

In COBOL a simple API service would be called like this:

```
...
INTFUNC = 'INIT'.
MOVE 0 TO INTCWA.
CALL 'DSLAPI' USING INTWSTOR.
IF INTRC NOT = SPACES THEN
...
```

The API function to be carried out (INIT) is one parameter to be passed to DSLAPI. INTRC is the basic DSLAPI return code. Both these fields are defined in the INTWSTOR structure.

In Assembler you would write:

```
...
MVC   INTFUNC,=C'INIT'    The API service
XC    INTCWA,INTCWA
LA    R1,INTWSTOR         ..is specified in this structure
ST    R1,PARMLIST         ..which is the only parameter
LA    R1,PARMLIST         ..in this parameter list
L     R15,=V(DSLAPI)      The API program
BALR  R14,R15             (R13 addresses a register savearea)
CLC   INTRC,=C'  '        The return code
...
```

These examples show a *static* call: The DSLAPI program is statically linked to your application. Of course, you can also link to DSLAPI dynamically. The various ways of invoking DSLAPI, and how you do it from a CICS environment, which enforces other conventions, is described in the section “Calling DSLAPI” on page 6.

The parameter list passed to DSLAPI can contain one, two, three, or four parameters, depending on the API function. Parameters are passed using the standard OS convention: The address of a parameter list is passed to DSLAPI; this list is a list of addresses.

Interface Working Storage

The first DSLAPI parameter is always the interface working storage structure. In the example at the beginning of this chapter, the API initialization call (INIT), the interface working storage structure (or INTWSTOR) is the only parameter.

The interface working storage is a data structure, defined by the copybook or include-file DSLAPIWS, which you declare in your program’s automatic storage (for example, in DFHEISTG under CICS). It is the basic medium of communication between your program and DSLAPI, and is also used by DSLAPI to hold internal data structures and storage anchors. INTWSTOR, in effect, defines the API environment.

Because this environment must first be established before you can begin to use API services, the first API call must be an INIT call. This causes DSLAPI to set up its environment and to initialize your INTWSTOR structure. Similarly, at the end of your program, you must call DSLAPI one last time with a *terminate* (TERM) call, so that DSLAPI can close down the environment in an orderly fashion.

Parameters

In addition to general interface information, INTWSTOR also contains the parameters specific to the API queue management and message formatting services. When you use one of these services you indicate precisely what you want to do by setting these parameters. For example, if you want to add a message to a MERVA queue, one parameter you must specify is the name of the queue. You set this name into the INTQUEUE field in the interface working storage structure.

The interface working storage structure is described in detail in "Chapter 8. Data Structures" on page 81.

Other API Parameter Structures

Parameters for the API queue and message formatting services are defined in the INTWSTOR structure. Parameters for other services are defined in separate structures. For example, if you are accessing individual fields in a message (using TOF services), you must supply a *field-reference* structure as the second parameter of the DSLAPI call to identify precisely the field you want. This structure is called TOFPARM, and is defined in the copybook DSLAPITP. TOFPARM defines the parameters specific to the API TOF services.

Similarly, when reading or writing the MERVA journal, you must supply the journal record key as the second parameter of the API call, using the JRNKEY structure (copybook DSLAPIJK). The fields in this key structure are the parameters specific to the API journal services. Additionally, you must supply a buffer for the journal record.

Buffers

External Buffers

If an API service moves data to or from your application, you must also supply a buffer for the data. Some buffers have an unvarying length so that you do not need to supply the length of the buffer. For example, the buffer for the response from the operator command service must always be 700 bytes long; the response always contains 10 lines of 70 characters, even if most of the lines are blank.

Other buffers are of variable length. With these buffers you must not only supply the buffer, but also tell DSLAPI how long the buffer is. There are two ways of doing this, depending on the type of buffer you are using:

Small buffers

Buffers that are always less than 32KB long have a standard MERVA buffer format. Such buffers contain a buffer prefix with two half-word binary length values:

- The buffer length, which does not change after storage for the buffer has been allocated
- The data length, which defines the length of the data in the buffer at any one time

For example, if you are using the API JRNN service to read sequentially through the MERVA ESA journal, you might allocate a 32,000 byte buffer for the journal record:

```
01 JOURNAL-BUFFER.
   02 BUFFER-PREFIX.
      03  BUFFER-SIZE          PIC S9(4) BINARY.
      03   filler              PIC X(2).
      03  DATA-SIZE            PIC S9(4) BINARY.
      03   filler              PIC X(2).
   02 JOURNAL-RECORD           PIC X(31992).
```

You would set `BUFFER-SIZE` to 32,000, the overall size of the buffer, when you allocate storage for the buffer. After each JRNN call the API journal service itself will have set `DATA-SIZE` to the actual length of the journal record it read plus 4 for the `DATA-SIZE` part of the prefix. The buffer prefix structure, BUFFER_PREFIX, is defined by the copybook DSLAPIBP.

Large buffers

For larger buffers (up to 2MB), API services are provided to set the length fields of buffers with a buffer prefix. Instead of a single parameter identifying a buffer with a standard prefix, these alternative services require two parameters: The address of the message buffer, which contains no prefix, and a fullword binary length value.

This length value is used to define both the buffer size and the actual data size. For example, the alternative to JRNN for reading the MERVA journal sequentially is JRLN. When you invoke this service you specify in the length parameter the size of your buffer; on return, DSLAPI sets the actual size of the record it read into this length parameter.

Using these new services you can process messages that are smaller than 32KB as well as larger messages.

Internal Buffers

When you use API Queue Management services to access a message in one of the MERVA message queues you do not supply a buffer yourself. Instead, the API uses its own *internal queue buffer* to hold the message. You do not directly manipulate this buffer. You can only indirectly manipulate it by using API services.

For example, when you use a Message Formatting service to map a message from the internal MERVA form to an external format, or you use TOF services to extract fields from a message, you are accessing the message in this internal queue buffer. This is discussed in more detail in the section “Processing Messages with DSLAPI” on page 10.

Calling DSLAPI

Unless your application is running in the CICS environment, use standard OS calling conventions to call DSLAPI. Do one of:

- Combine DSLAPI and your application into a single executable module using the linkage editor (this means that your application is statically linked to DSLAPI).
- Link your application separately from DSLAPI and establish the connection to DSLAPI at run time (this involves dynamic linkage).

Whichever method you choose, refer to your programming language manuals for any special considerations on static and dynamic calling. The DSLAPI program is an Assembler program, is reentrant, runs in 31-bit addressing mode (AMODE 31), and can reside either below or above the 16MB boundary (RMODE ANY).

Static Call

The following example shows how to call DSLAPI using static calling from an assembler language application program:

```
         EXTRN DSLAPI
ADSLAPI  DC    A(DSLAPI)          ADDRESS OF API PROGRAM
PARMLIST DC    A(WS)              PARAMETER LIST
         DC    A(0)               ..MAY CONTAIN UP TO
         DC    A(0)               ..FOUR PARAMETERS
         DC    A(0)
WS       DS    CL(INTWSTLL)       INTWSTOR
         ...
         LA    R9,WS              ESTABLISH BASE-REGISTER..
         USING INTWSTOR,R9        ..FOR THE INTERFACE WORKING STORAGE
         ...
         LA     R1,PARMLIST       LOAD ADDRESS OF PARMLIST
         L      R15,ADSLAPI       LOAD ADDRESS OF API PROGRAM
         BALR   R14,R15           BRANCH TO API PROGRAM
         ...
```

Static calling is used in the MERVA ESA sample programs because it is simpler than dynamic calling, but it has several disadvantages. For example, applications need to be re-linked when service updates are applied to DSLAPI, or when you install a new release of MERVA ESA.

Dynamic Call

Using dynamic calling keeps your applications completely separate from MERVA programs. Only while an application is executing is a link to DSLAPI established. This means that applications do not need to be re-linked when service updates are applied to DSLAPI, or when you install a new release of MERVA ESA.

In COBOL you specify dynamic calling by using a variable as the program name in a call statement or by specifying the DYNAM compiler option:

```
...
77  API-PROGRAM    PIC X(8).
...
MOVE 'DSLAPI' TO API-PROGRAM.
CALL API-PROGRAM USING INTWSTOR.
...
```

In other languages, you must explicitly load DSLAPI yourself before you want to call it for the first time. In C, for example, you use the library function **fetch**:

```
...
#include "dslapc.h"                        /* DSLAPI definitions */
#include <stdlib.h>
typedef void API();

int main(int argc, char *argv[]) {
  API * api_ptr;
  struct INTWSTOR ws;
  ...
  api_ptr = (API *) fetch("DSLAPI");  /* dynamically link DSLAPI */
  if (api_ptr == NULL)...                    /* .. failed      */
  memcpy(ws.INTFUNC,"INIT",4);
  ws.INTCWA = NULL;
  api_ptr(&ws);                              /* invoke DSLAPI */
```

```
   if (memcmp(ws.INTRC,"  ",2) != 0)
...
release(api_ptr);                            /* release DSLAPI */
}
```

Similarly in PL/I you use the **fetch** and **release** statements:

```
...
dcl dslapi entry options(assembler,inter);
%include dslapiws;
...
fetch dslapi;
allocate intwstor;
INTFUNC = 'INIT';
unspec(INTCWA) = 0;
call dslapi (intwstor);
if INTRC ¬= '  ' then do;
...
free intwstor;
release dslapi;
return;
```

In MVS Assembler you would use the LOAD and DELETE macros:

```
      ...
      COPY  DSLAPIWS            DSLAPI INTERFACE WORKING STORAGE
      ...
      LOAD  EP=DSLAPI           LOAD *DSLAPI*
      ST    R0,ADSLAPI          SAVE ENTRY-POINT-ADDRESS
      XC    INTCWA,INTCWA
      MVC   INTFUNC,=C'INIT'    FUNCTION = I N I T
      LA    R1,ASMPARM          DSLAPI PARMLIST
      L     R15,ADSLAPI         ADDRESS OF DSLAPI
      BALR  R14,R15             INITIALIZE DSLAPI
      ...
      DELETE EP=DSLAPI          FREE *DSLAPI*
      ...
```

CICS Considerations

CICS does not support the standard OS convention for parameter passing. It defines its own application-independent parameter list, one parameter of which (the CICS COMMAREA) is the application-specific parameter list. You have to build the API parameter list in the COMMAREA before passing control to DSLAPI.

Setting Addresses with COBOL

The API parameter list is a sequence of addresses. If you are using COBOL, you cannot set these addresses into the CICS COMMAREA unless all the parameters are defined in the linkage section (in which case you can use the SET TO ADDRESS statement). For the case where the parameters are not defined in the Linkage Section, MERVA ESA provides an additional routine, DSLAPIPL, which builds the parameter list for you. You call DSLAPIPL just as you would call DSLAPI directly, and provide additionally, as the first two parameters, a 72-byte work area and the address of your CICS COMMAREA:

```
►►—DSLAPIPL—(—workarea—,—commarea—,—INTWSTOR—,—...—)——————————————►◄
```

DSLAPIPL merely moves the addresses of the parameters into the COMMAREA, then returns control to your COBOL program, after which you can invoke the API service.

For an example of the use of DSLAPIFL, see “Alternative API Entry Name” .

Alternative API Entry Name

Under CICS:

- You cannot call DSLAPI statically; you must use dynamic linkage
- You cannot use the language-specific mechanisms for dynamic calls discussed in the previous section; you must use CICS services, specifically `EXEC CICS LINK`

Because DSLAPI expects to receive a standard OS parameter list, MERVA ESA provides an interface module, DSLAPCIC, which accepts the CICS parameter list and transforms it into the form expected by DSLAPI before passing control to DSLAPI. When invoking the API under CICS, your program must call DSLAPCIC instead of DSLAPI.

The following COBOL example shows the use of DSLAPIPL to set the API parameter list. Note that DSLAPIPL is statically linked to the application, while a dynamic call is made to DSLAPCIC.

```
...
working storage section.
copy dslapiws.
01 api-parm-list.
    02 parm             pointer occurs 4.
01  workarea.
    02 filler           pointer occurs 18.
procedure division.
    move 'INIT' to intfunc
    move 0 to intcwa
    call 'DSLAPIPL' using workarea, api-parm-list, intwstor
    exec cics link program('DSLAPCIC')
         commarea(api-parm-list) length(4) end-exec
    if intrc not = spaces then
    ...
```

Here is the same program written in C/370. It is not necessary to use DSLAPIPL to set up the API parameter list:

```
...
#include <cics.h>
#include "dslapc.h"                 /* API definitions */

int main() {
  struct INTWSTOR ws;
  struct {struct INTWSTOR *parm1;   /* CICS commarea */
          char            *parm2;
          char            *parm3;
         } ca;
  ...
  memcpy(ws.INTFUNC,"INIT",4);
  ws.INTCWA = NULL;
  ca.parm1 = &ws;;
  EXEC CICS LINK PROGRAM("DSLAPCIC") COMMAREA(&ca) LENGTH(4);
  if (memcmp(ws.INTRC,"  ",2) != 0) }
  ...
```

IMS Considerations

When DSLAPI runs in an MPP under control of an IMS transaction, DSLAPI needs the address of the PCB list, which IMS passed to your application program as the calling parameter list address. The PCB list is needed to perform MERVA ESA file service requests by accessing IMS databases such as the SWIFT Link currency file.

After the API INIT function has been called, the PCB list address must be set into the field COMPCBLA. To set the address, use the FLDP service with field name COMPCBLA.

Processing Messages with DSLAPI

MERVA ESA is a message-processing system. Within MERVA, messages are held in message queues. Messages in queues are in a MERVA internal format called *compressed tokenized form*, or *compressed TOF*. When you are handling messages in the internal format, you do not have direct access to the message; the message is managed for you by DSLAPI, and you must use API services to access the message.

When you write an application using API services, you will probably want to process messages in one or more of the following ways:

Reading
: For example, to gather statistics on messages in a particular queue or queues.

Updating and routing
: If you are going to route messages from one queue to another, or update messages, you should indicate that you want exclusive use of the message.

Exporting
: Removing messages from the MERVA system involves two API steps: retrieving the message from its queue into the internal form, and transforming the message from the MERVA-internal form to an external format.

Importing
: To import messages into the MERVA system two API steps are again necessary, mapping the message from external to internal format, and then adding the message to one or more queues.

Printing
: Also to print messages two API steps are necessary: retrieving the message from its queue into the internal form, and formatting the message into print lines.

Reading

To read messages from queues, use a queue management service such as GET. GET is intended for read-only sequential queue retrieval. Do not use GET if you are going to modify the message; instead use a queue management service that gives you exclusive control of the message.

To extract fields from the retrieved message you use the TOF READ service:

```
           ...
           perform until intrc  not = spaces
             move 'GET ' to intfunc
             call 'DSLAPI' using intwstor
             if intrc = spaces then
      *          now the message is in API internal storage
               move 'READ' to intfunc
               move field-name to toffdnam
               move 'VFIRST' to tofmodif
               call 'DSLAPI' using intwstor tofparm tof-data
           ...
           end-perform
```

Note that you do not provide a buffer when you retrieve a queued message; instead, the message is held by DSLAPI in an internal buffer. But you do specify a buffer for the field you read (in this example, **tof-data**).

Updating and Routing

Before you update or route a message, indicate that you want exclusive access to the message. Otherwise, someone else might simultaneously update or route the message, and the results will be unpredictable. To indicate tha you want exclusive use of a message, retrieve it using a queue function that sets the *in-service* indicator, for example:

GETC	For direct access to a message
GETK	For direct keyed access to a message
GETN	If you are processing a queue sequentially

Other programs can still read a message flagged *in-service* by using a nonexclusive GET, but attempting to read it with an exclusive get will fail with an appropriate return code.

After it is retrieved, the message is in the API internal buffer. To update the message, that is, to update fields in the message, you use TOF services. You can READ fields, update or add fields (WRIT), and delete fields (EMPT).

You can then write the message back into the same queue again using the queue management replace (REPL) service, or invoke a routing table to route the message to other queues (ROUB). If, after inspecting the message, you decide you do not want to update or route it, you can relinquish exclusive access using the FREE service.

To route a queue element from one queue to another without updating or inspecting it, you can use the ROUD and ROUN function. This avoids retrieving the message into the internal queue buffer.

Exporting

To export or remove a message from the MERVA system, first retrieve the message from its queue using a function that indicates you want exclusive access (GETC, GETK, or GETN). Then, use API message formatting services to transform the message from internal form (***TOF?) to the external format you require. For example, if the message is a S.W.I.F.T message and you want the message in SWIFT II format, specify format identifier **W**.

After extracting the message, you must delete it from the queue explicitly using the queue delete service (DELE), for example:

```
          ...
          move our-key to intkey1
          move queue-name to intqueue
          move 'GETK' to intfunc
          call 'DSLAPI' using intwstor
     *        the message is now in API internal storage
          move 'W' to intfrmid
          move 'MSGG' to intfunc
          move length of msgswift-buffer to msg-length
          call 'DSLAPI' using intwstor msgswift-buffer msg-length
     *        the message is in 'msgswift-buffer' in SWIFT-II format
          ...
          move 'DELE' to intfunc
          call 'DSLAPI' using intwstor
     *        the message has been deleted from the MERVA queue
          ...
```

If, instead of exporting the message, you are merely copying messages from the MERVA system, your program would be similar but you would not need exclusive access to the message in the queue. Instead of the GETK service you would use GEKU (for read-only, keyed retrieval).

Importing

To import messages into the MERVA system, two API steps are necessary:

- Use a message formatting service to transform (map) your external message to an API internal buffer in the MERVA-internal format (***TOF?).
- Then, write the message from the API internal buffer to a queue or queues. You can write to a specific queue using PUT, or invoke a routing table to determine the target queue (or queues) from the message content (ROU service). To identify a routing table, supply the MERVA function name (queue name) with which the routing table is associated:

  ```
          ...
          move 'S100' to intmsgid
          move 'W' to intfrmid
          move 'MSGP' to intfunc
          move length of msgswift-buffer to msg-length
          call 'DSLAPI' using intwstor msgswift-buffer msg-length
     *        the message is now in API internal storage
          move function-name to intqueue
          move 'ROU ' to intfunc
          call 'DSLAPI' using intwstor
     *        the message has been routed to one or more queues
          ...
  ```

  If you are not using a MERVA-supplied routing table, you must write one yourself (how to do this is described in the *MERVA for ESA Customization Guide*).

Printing

To print a message, first retrieve it from its queue, then use the:

- PRTI service to customize the printing environment (if necessary)
- PRTL service to print it line by line to a system printer or a file
- PRTT service to terminate the printing environment and release the resources

Chapter 2. The MERVA ESA API Services

This chapter discusses the various MERVA ESA services you can use via the API:

- Queue management services, which you use to manage queue elements (queued messages)
- TOF services, which manipulate fields in messages
- Message Format Service (MFS) services, which map messages between the MERVA internal form and external formats
- Print services to print messages line by line
- Journal services, which read and write records in the MERVA journal
- User file services, with which you can read records from the MERVA ESA User file
- The Command (CMD) service to execute operator commands
- The WTO service, which sends messages to the operator.

Queue Management Services

The DSLAPI queue management services let you manipulate queue elements in MERVA ESA queues. A queue element is a message in the internal MERVA ESA format, that is, in tokenized form (or TOF). Queue elements are identified by:

- The name of the queue containing the queue element.
- The queue sequence number (QSN) of the element. A QSN is unique within a queue.
- Optionally, one or two symbolic keys (if the queue has been defined with keys). Keys are not necessarily unique.

Internal Queue Buffer

Central to queue management services is the internal queue buffer, in which DSLAPI holds the messages it reads from a queue or is to write to a queue. You cannot access this buffer directly. Instead, you use API TOF services to modify a message in this internal buffer, and API message formatting services (MFS) to transfer a message between the internal buffer and your own, external buffer.

Overview of Queue Management Functions

Using API queue management functions you can list, retrieve, insert, replace, delete, and route queue elements.

When retrieving a message you can indicate that you want:

- Exclusive use of the message, for example if you intend to update or remove it. The queue element is flagged as being *in-service* to prevent other users from trying to simultaneously update the message, which might result in lost updates.
- Nonexclusive use of the message, for example if you just want to read the message. This is referred to as *browsing*, and lets you read messages, even if they are exclusively held by other users.

Note: MERVA ESA does not prevent a message flagged as being *in-service* from being updated by another application. It is the responsibility of application programmers to respect this flag. Refer to *MERVA for ESA Concepts and Components* for more information on the *in-service* indicator.

You can retrieve messages:
- Sequentially, by ascending QSN.
- Directly by QSN.
- Directly by key, if the queue has been defined with keys. Note that keys need not be unique.

And, since the API updates the queue management parameters after each invocation, you can freely alternate between direct and sequential retrieval.

To list the elements in a queue, use the following functions:

QLF Retrieves the first QSN and the key values for a MERVA ESA queue.

QLL Retrieves the last QSN and the key values for a MERVA ESA queue.

QLN Retrieves the next QSN and the key values for a MERVA ESA queue.

QLP Retrieves the previous QSN and the key values for a MERVA ESA queue.

The following retrieval functions give you *exclusive* use of messages; use them to read messages you intend to update or move:

GETC The *get conditionally* function retrieves the specified queue element from the specified MERVA ESA queue and puts it in the internal queue buffer, but only if the queue element is not *in-service*.

GETK The *get by key* function retrieves the queue element with the specified key from the MERVA ESA queue and puts it in the internal queue buffer.

GETN The *get next* function retrieves the next queue element that is not *in-service* from the specified MERVA ESA queue and puts it in the internal queue buffer.

The following retrieval functions give you *nonexclusive* use of messages; use these functions for read-only access to a queue. They are nonexclusive because they allow you to retrieve all messages, even those that have been flagged *in-service* by other users:

GET The *get unconditionally* function retrieves the queue element with the specified QSN from the MERVA ESA queue and puts it in the internal queue buffer.

GETU The *get next unconditionally* function browses the next queue element from the MERVA ESA queue and puts it in the internal queue buffer.

GEKU The *get by key unconditionally* function browses the queue element with the specified key from the MERVA ESA queue and puts it in the internal queue buffer.

The following functions are used to write messages to queues. The PUTB and ROUB functions require a back reference to an existing queue element, which will be automatically deleted at the same time as the new message is enqueued. This automatic delete facility ensures queue integrity when moving messages from one queue to another:

REPL The *replace* function replaces the specified queue element with the queue element in the internal queue buffer.

PUT The *put* function appends the queue element from the internal queue buffer to the specified MERVA ESA queue. Use this service when adding a new message to a queue.

PUTB The *put with back reference* function appends the queue element from the internal queue buffer to the specified MERVA ESA queue, automatically deleting the message specified in the back reference. Use this service when moving a message from one queue to another.

PUTR The *restore* function writes the queue element from the internal queue buffer to the specified MERVA ESA queue with the specified QSN, keys, and DOUBLE (write-back) indicator. Use this service when reloading messages to a queue.

ROU The *route* function routes the queue element from the internal queue buffer to one or more MERVA ESA queues selected by the routing table of the specified MERVA function. Use this service when introducing a new message to the MERVA queuing system.

ROUB The *route with back reference* function routes the queue element from the internal queue buffer to one or more MERVA ESA queues selected by the routing table of the specified MERVA function, automatically deleting the message specified in the back reference. Use this service when routing a message from one queue to another.

ROUD
The *direct route* function takes the queue element with the specified QSN from the MERVA ESA queue and routes it to one or more MERVA ESA queues selected by the routing table of the specified MERVA function. The queue element is also put in the internal queue buffer. Use this service when routing a queue element unchanged from one queue to another.

ROUN
The *route next* function takes the next queue element with a QSN higher than the specified QSN from the MERVA ESA queue and routes it to one or more MERVA ESA queues selected by the routing table of the specified MERVA function. The queue element is also put in the internal queue buffer. Use this service when routing a queue element unchanged from one queue to another.

The remaining queue management functions are:

DELE The *delete* function deletes the specified queue element from its queue.

FREE The *free* function turns off the *in-service* indicator of the queue element in the MERVA ESA queue. Use this service to relinquish exclusive use of a message if you decide not to update a message you obtained with an exclusive retrieval.

Refer to “Chapter 9. DSLAPI Functions” on page 97 for a description of each of these functions.

Use of the BUSY (In-Service) Indicator

The BUSY, or *in-service*, indicator shows whether another user has acquired exclusive access to a queue element. The BUSY indicator is contained in the field INTBUSY of INTWSTOR.

The BUSY indicator is returned after:

- Any nonexclusive retrieval of a message that has been exclusively retrieved by another user. Despite the *in-service* status, the message is retrieved.
- A direct retrieval for exclusive use if the message has already been exclusively obtained by another user. Your retrieval will not be successful.
- A sequential retrieval for exclusive use if all the remaining messages in the queue have already been exclusively retrieved by other users. Your retrieval will not be successful.

A message's BUSY indicator is cleared by:

- The FREE function
- The PUTB function
- The ROUB function

Note: The BUSY indicator is not cleared by a REPL, and is not preserved after MERVA ESA termination.

Use of the DOUBLE (Write-Back) Indicator

The DOUBLE, or *write-back*, indicator is returned in the field INTDOUBL of INTWSTOR. The indicator is set on in a queue element when the element is read by an exclusive retrieval. The flagged element is then written back to the queue data set to preserve the indicator across MERVA ESA termination/startup. It is intended as a recovery aid for programs removing messages sequentially from the MERVA ESA message queuing system. It can otherwise be ignored.

If the DOUBLE indicator is not needed, you can suppress its use for performance reasons. Refer to "Runtime Environment Settings" on page 52 for details about the flag APICQWRB, which allows you to suppress the writing of the DOUBLE indicator.

Messages being exported from MERVA should be held in their own queue to which no users have access. A program exporting the messages, for example, to a sequential data set, should first *copy* the messages sequentially from the queue. Then, when all messages have been copied, the program can delete the messages from the queue. Following abnormal termination, the program can use this indicator to determine where to restart.

The indicator is used by the MERVA batch output programs DSLSDO and DSLSDOR and the print transaction program DSLHCP. Be careful that your application does not interfere with the message queues of these programs.

The DOUBLE indicator is discussed in more detail in *MERVA for ESA Concepts and Components* where it is called the *write-back* indicator.

Automatic Delete

The functions PUTB and ROUB ensure message integrity when moving messages from one queue to another by using a back reference to delete the queue element only after it has been moved. Using this mechanism, MERVA ESA ensures that messages can never be lost or duplicated, even if there is an abnormal system termination during the PUTB or ROUB process.

The following PL/I example moves all messages from QUEUEA to QUEUEB (afterward, QUEUEA is empty):

```
...
saveqsn   = 0;              /* read from the beginning of the queue */
do until (intrc ¬= '  ');
  intfunc = 'GETN';         /* get sequential exclusive             */
  intqsn = saveqsn;         /* ..the msg after this qsn             */
  intqueue = 'QUEUEA';      /* ..from this queue                    */
  call dslapi (intwstor);
  if intrc = '  ' then do;  /* a message is now in the int. buffer  */
    saveqsn = intqsn;       /* save qsn of gotten message           */
    intfunc = 'PUTB';       /* put with automatic delete            */
    intbque = intqueue;     /* ..from this queue                    */
    intbqsn = intqsn;       /* ..with this qsn                      */
    intqueue = 'QUEUEB';    /* ..into this queue                    */
    call dslapi (intwstor);
    put skip edit ('moved from',intbque,intbqsn,'to',intqueue,intqsn)
                  ( 2( 2(a,x(1)), f(9),x(1) ) );
  end; /* if */
end; /* do */
...
```

Tokenized Form (TOF) Supervisor Services

MERVA ESA is a system for managing formatted messages. Formatted messages have a defined data-field structure. This structure can be complex, with repeated sequences of fields, nested sequences of fields, and nested (embedded) messages. Fields, too, can be divided into subfields and multiple data areas.

If a message is in tokenized form (TOF), you can use API TOF services to manipulate its fields. You can read, write, or delete TOF fields, sequences of fields, embedded messages, subfields, and data areas of fields. The concepts of TOF, the TOF supervisor, and the field-address structure are explained in *MERVA for ESA Concepts and Components* and *MERVA for ESA Customization Guide*.

Note: MERVA ESA provides two alternative application interfaces for reading and writing fields in a S.W.I.F.T message. Instead of using API queue and TOF services you can use these alternative services to read and write fields in S.W.I.F.T messages. These alternative interface programs, DSLAPFFS and DSLAPFTS, are described in “Chapter 7. Auxiliary API Services” on page 59.

Overview of TOF Supervisor Functions

The following functions are used for TOF field management:

READ The *read* function reads a field from the internal queue buffer through the internal TOF and puts it in the field buffer.

WRIT The *write* function writes a field from the field buffer through the internal TOF to the internal queue buffer.

EMPT The *empty* function deletes a field in the internal queue buffer.

Refer to “Chapter 9. DSLAPI Functions” on page 97 for a description of each of these functions. Their precise function can be varied by specifying *modifiers* (see “Request Modifiers” on page 87).

API TOF Parameters

Like all other API calls, the first parameter of a TOF service call must be the interface working storage structure (INTWSTOR). All TOF service calls require as their second parameter the TOF parameter structure, TOFPARM (copybook DSLAPITP). Before invoking a TOF service you must define the field you want to access by setting values, the field reference, into the TOFPARM. After the call your current position in the TOF is returned in the TOFPARM structure.

The TOFPARM also contains the return codes and reason codes from the TOF supervisor. Note that the INTRC return code in INTWSTOR merely indicates whether DSLAPI successfully invoked the TOF supervisor; the result of the TOF service is given by the TOFPARM codes.

Refer to "TOF Access Parameters TOFPARM" on page 85 for a description of the TOFPARM structure.

When reading or writing data you must specify a third parameter, the MERVA buffer, into which data is to be read, or from which data is to be written.

The Internal TOF Buffer

A message in TOF form is held in an API internal buffer. You cannot manipulate this buffer directly; you can only manipulate the contents of a message by using API TOF services. The internal TOF buffer is an expanded form of the internal queue buffer. When using API services, these two internal buffers are logically identical: If there is a message in the internal queue buffer, you can immediately use TOF services to access its contents. You do not explicitly move a message between the TOF and queue buffers.

Defining a Field Buffer

With the READ and WRIT functions you must define a buffer, with a standard MERVA buffer prefix, for the field data. See the description of small buffers in "External Buffers" on page 5.

Notes:

1. When reading, if the length of the data is not 0, DSLAPI sets the actual data length to the length plus 4 (LL). If the length is 0, DSLAPI fills the buffer with blanks.
2. When writing, DSLAPI uses the specified length or, if no length is specified, it calculates the length from the data in the buffer by finding the last non-blank character.

Message Formatting Services (MFS)

The Message Format Services (MFS) map (transform) messages from external to TOF format, and vice versa. In MERVA ESA, a message is described by a message control block (MCB). The MCB defines both the message (that is, the sequence of fields that make up the message) and its various external formats (that is, its layouts on panels, printers, and networks).

Overview of MFS Functions

The following functions are provided by the Message Format Service:

MSGG	The *message get* function maps the message from the internal queue buffer through the internal TOF to the message buffer.
MSGP	The *message put* function maps the message in the specified format from the message buffer through the internal TOF to the internal queue buffer.
MPFG	The *message prefix get* function extracts just the MSGSWIFT_PREFIX structure (see copybook DSLAPIMP).

MPFP	The *message prefix put* function can be used to import the fields from the MSGSWIFT_PREFIX structure.

The following functions can be used with messages smaller than 32KB:

GETS	The *get SWIFT message* function maps the message in SWIFT format from the internal queue buffer through the internal TOF to the message buffer.
GETM	The *get message* function maps the message using the specified MCB and the specified format from the internal queue buffer through the internal TOF to the message buffer.
PUTS	The *put SWIFT message* function maps the message in SWIFT format from the message buffer through the internal TOF to the internal queue buffer.
PUTM	The *put message* function maps the message in the specified format from the message buffer through the internal TOF, to the internal queue buffer.

Refer to “Chapter 9. DSLAPI Functions” on page 97 for a description of each of these functions.

In addition to a buffer for the message in its external form, the MFS needs to know the message type, that is, which MCB to use, and the external format identifier. These two parameters are specified in the INTMSGID and INTFRMID fields of the interface working storage INTWSTOR.

Importing Messages and Determining Message Types

When importing a message into the MERVA ESA system, MERVA needs to know the messages type. If you do not know the message type, leave the message type parameter (INTMSGID) blank, and MERVA uses message type determination exit routines to determine the type of the message.

MERVA ESA provides message determination exit routines for the message types it supports: SWIFT messages, Telex messages, and financial EDIFACT messages. You can write your own message type determination exit routines (see exit DSLMU054 in the *MERVA for ESA Customization Guide*).

Exporting Messages

When exporting a message from MERVA, you can specify the message type parameter (INTMSGID), but it is better not to. This is because the message type is known by MERVA; the message identification is stored with the message when it is created. When INTMSGID is set to blank, the original message identification is used. If you specify the wrong type, the message will be formatted using any fields MFS finds that are also defined in the MCB you specify. The result will be an incomplete message.

However, you should specify the format in which you want the message to be mapped (parameter INTFRMID). If you leave the format identifier blank, MFS will use as a default the first format defined in the MCB. This is not recommended because new formats might be added to the MCB.

Refer to “Message Exit Fields” on page 54 for information on exit fields that you can use to determine the message type in nested and combined messages.

The MFS Message Buffer (MSGSWIFT)

When you use API MFS services, you must provide, as the second parameter of the DSLAPI call, a buffer for the external form of the message. This buffer is called MSGSWIFT, and for the GETS, GETM, PUTS, and PUTM functions, its layout is defined by the copybook DSLAPIMS.

MSGSWIFT is comprised of a header (prefix), defined by the structure MSGSWIFT_PREFIX in the copybook DSLAPIMP, followed by the actual message in a standard MERVA buffer. The overall length of the MSGSWIFT buffer is defined in the MERVA ESA parameter module, DSLPRM. Because DSLAPI takes the length from DSLPRM and initializes the buffer accordingly, you must ensure that the buffer in your program is as long as the length given in DSLPRM. DSLAPI does not check to see if the storage area provided by your application program is large enough. If your storage area is too small, DSLAPI will overwrite other data in your program with unpredictable results.

When using the MSGG and MSGP functions, you must supply the following parameters:

- The address of a buffer to hold the message. This buffer contains neither the MSGSWIFT prefix nor a MERVA buffer prefix.
- The length of the buffer, when exporting a message (MSGG), or of the message in the buffer when importing a message (MSGP).

Print Services

The print services let you print the message that is currently in the internal buffer line by line to a system printer or a file.

Overview of the Print Message Functions

The following functions are provided by the API print services:

PRTI The *print initialize* function initializes the print environment.

PRTL The *print next line* function formats the next line to be printed. The calling application may choose to actually print the result line in the buffer.

PRTT The *print terminate* function terminates the formatting of messages for the printer. The resources used for creating the print lines are released.

Refer to “Chapter 9. DSLAPI Functions” on page 97 for a description of each of these functions.

Journal Services

With the API journal services, you can read and write records from the MERVA ESA journal. All journal records have a key. The journal key consists of the following parts:

- A journal record type identifier (JRNRID). The record types used by MERVA are listed in *MERVA for ESA Concepts and Components*. You can use your own types.
- A time stamp (JRNKDAT2, JRNKTIM2, and JRNKFRC2), which is generated by MERVA.
- Segment information for segmented records (JRNKSEG and JRNKSEGS), which is generated by MERVA.
- A user-key extension (JRNKUFLD or JRNKUSER). This field can be used by the application.

The journal-key structures, JRNKEY and JRN2KEY, are defined by the copybook DSLAPIJK.

Overview of Journal Functions

The following functions are used for journal record management:

JRLG	The *journal get* function reads the journal record with a key equal to the journal key from the MERVA ESA journal and puts it in the buffer you supply. If the key does not exist, the record with the next higher key is returned, so you can provide an incomplete or *generic* key.
JRLN	The *journal get next* function reads the journal record with the next higher journal key from the MERVA ESA journal and puts it in the buffer. Use this function to read the journal sequentially.
JRLP	The *journal put* function adds a record from the buffer to the MERVA ESA journal. You provide the record type ID and the user-key extension, MERVA generates the time stamp.
JRNG	This is similar to JRLG but supports only journal records of up to 32KB.
JRNN	This is similar to JRLN but supports only journal records of up to 32KB.
JRNP	This is similar to JRLP but supports only journal records of up to 32KB.

Refer to "Chapter 9. DSLAPI Functions" on page 97 for a description of each of these functions.

Defining a Journal Record Buffer

When using the functions JRLG, JRLN, and JRLP, you must supply the following parameters:

- A buffer, which does not contain a buffer prefix
- The length of the buffer

When using the functions JRNG, JRNN, and JRNP, you must supply a buffer with a standard MERVA buffer prefix for the journal record. See the description of small buffers in "External Buffers" on page 5.

Note: The sum of the maximum record length and an additional 50 bytes for the journal key must not be larger than the maximum record length specified in both the journal cluster definition and the JRNBUF parameter of the MERVA ESA customizing table (DSLPRM).

User File Services

The user file services allow you to read records from the MERVA ESA user file. The user file is described in *MERVA for ESA Concepts and Components*.

Use of this API service is automatically recorded in the MERVA ESA journal by DSLAPI. The journaled user ID is taken from the APIUID parameter in the MERVA ESA parameter module, DSLPRM.

If `EXDSP=NO` is specified in DSLPRM, then access to the user file by application programs is suppressed.

Overview of the User File Functions

The following functions are provided by the API user file services:

USRG The *user file get* function reads the user file record for the specified user identification.

USRN The *user file get next* reads the user file record following the record for the given user identification. Use this service to read sequentially through the user file.

Refer to "Chapter 9. DSLAPI Functions" on page 97 for a description of each of these functions.

The User file functions require two parameters:

- A user ID.
- A buffer for the retrieved user file record. The buffer does not have a MERVA buffer prefix; its record structure, DSLUSRS, is defined by the copybook DSLAPIUS.

Operator Command Service

The *command execution* function lets your application program execute the same commands that can be entered from the online CMD panel; that is, all commands supported by the MERVA ESA Operator Command Service. The Operator Command Service is described further in *MERVA for ESA Concepts and Components*.

This capability is provided by the API function **CMD**, which requires two fixed-length parameters:

- A pointer to the storage area containing the command to be executed. This area is 120 bytes long.
- A pointer to the area where the response is to be stored. This area is 700 bytes long and does not have a MERVA buffer prefix. The response is always returned in this area as 10 lines of 70 characters, just as the response would appear on a terminal.

The result of the command execution is journaled by MERVA ESA. The operator user ID in the journal record is taken from the APIUID parameter of the MERVA ESA customizing table, DSLPRM.

Write-to-Operator Service

A write-to-operator (WTO) service is provided by the API. It lets unsolicited operator messages be written to the system console and added to the MERVA ESA display message table. The WTO function code is **WTO**.

The WTO function requires a single parameter: a buffer containing the operator message to be passed to the MERVA nucleus. This buffer does not contain a MERVA buffer prefix. Refer to "Chapter 9. DSLAPI Functions" on page 97 for a description of this function.

Chapter 3. Writing an API Program

This chapter describes various types of application programs you might want to write using the MERVA ESA API, and some things you should consider when writing them. The following types of application program are considered:

- Batch programs
- Nonconversational transactions
- Conversational transactions
- Pseudoconversational transactions

For more information about writing transactions, refer to the *CICS/ESA Application Programming Guide* or *IMS/ESA Application Programming: Design Guide*.

Writing a Batch Program

You write a batch API program just as you write any other batch program: as an independent program that will run in its own region, or partition, separate from MERVA ESA.

Some MERVA ESA API services (for exampleTOF and MFS) use MERVA direct services that can be used independently of an active MERVA ESA system. Other API services, in particular queue management services, invoke MERVA central services, which means that a MERVA ESA system must be active in another region. For these services, the DSLAPI program uses MERVA intertask communication to pass your request to MERVA. Direct and central MERVA services are discussed in *MERVA for ESA Concepts and Components*.

Before any API program can begin to use API services, it must first set up an API environment. It does this by calling the API initialization function INIT. Before terminating, it must close the API environment by issuing the API termination call TERM. Between the INIT and TERM calls, your program can invoke any of the API services. Some services require storage buffers that you must establish before the service is called. The parameters and storage areas required by each API function are described in “Chapter 9. DSLAPI Functions” on page 97.

Figure 1 on page 24 shows a very simple but complete COBOL batch API program that sends a MERVA operator command to MERVA, then prints out the result.

```
01     identification division.
02     program-id.    samp00y.
03     data division.
04     working storage section.
05     copy dslapiws.
06     77  cmdinp                pic x(120).
07     01  cmdresp.
08         02 resp-line           pic x(70) occurs 10 indexed by i.
09     procedure division.
10         move 'INIT' to intfunc
11         move 0 to intcwa
12         call 'dslapi' using intwstor
13         if intrc not = spaces then
14           display intrc ' ' intermsg
15           goback
16         end-if
17
18         move 'DP' to cmdinp
19         move 'CMD ' to intfunc
20         call 'dslapi' using intwstor, cmdinp, cmdresp
21         if intrc not = spaces then
22           display intrc ' ' intermsg
23         else
24           perform varying i from 1 by 1 until i > 10
25             display resp-line (i)
26           end-perform
27         end-if
28
29         move 'TERM' to intfunc
30         call 'dslapi' using intwstor
31         goback.
32     end program samp00y.
```

Figure 1. Sample COBOL Batch API Program

Description:

- The copy statement in line 5 declares the API interface working storage structure (INTWSTOR) by including the copybook that defines the structure (DSLAPIWS). All API programs must include this structure.
- The INTWSTOR variable INTFUNC must be assigned the code of the API function that is to be executed. In line 10, this variable is set to INIT, which is the initialization function code.
- In line 11, the variable INTCWA is set to 0 to indicate that the program runs in a batch environment.
- In line 12, the API program (DSLAPI) is invoked to do the initialization.
- Also defined in the INTWSTOR structure is API return information. In line 13, the return code (INTRC) is tested. A successful API call is indicated by a return code of blanks. Another field in INTWSTOR is INTERMSG which, in case of an error, may contain an explanatory MERVA ESA error message. In line 14, it is displayed together with the nonblank return code as an error diagnosis aid.
- In line 20, The API CMD function is used to send the DP (display programs) command to MERVA ESA. The API CMD function requires two parameters in addition to INTWSTOR: the command to be executed, and a buffer to hold the response from MERVA.
- In line 30, DSLAPI is called again to terminate the API before the program itself terminates.

The CMD function uses MERVA ESA central services, so a MERVA ESA system must be active when this program is run. If the CMD function is successful, the sample program generates output that looks like this:

```
DSL102I Display Programs
 Progname PID Y S P A Status LRC    Progname PID Y S P A Status LRC
 RTCOMM   1   6 N N Y ACTIVE  00    CONSOLE  2   9 Y Y Y ACTIVE  00
 BATCH    3   4 N N Y ACTIVE  00    TRANSACT 4   5 N N Y ACTIVE  00
 SYNPOINT 5   6 Y Y Y INACTV  00    MSGCOUNT 6   6 N N Y ACTIVE  00
 DSLNSFPP 7   2 N N Y ACTIVE  00    APPCSRV1 8   5 Y Y N INACTV  00
 MQISRV1  9   5 Y Y N INACTV  00    CICSSRV  10  5 Y Y Y ACTIVE  00
 SWIFTAUT 11  5 Y Y Y ACTIVE  00    SWIFTII  12  8 Y Y Y ACTIVE  00
 SWLOADSK 13  2 Y Y Y ACTIVE  00    TELEX    14  7 Y Y N INACTV  00
```

If MERVA ESA is not active, the CMD function fails, and the display statement on line 22 outputs:

```
02 DSL884I NIC04000 NIC=CMD RC=04
```

where 02 is the return code held in INTRC.

There are other examples of batch API programs among the sample programs distributed with MERVA ESA. These are described in “Appendix B. Sample Programs” on page 203.

Writing a Nonconversational Transaction

A *nonconversational transaction* is a batch program that is initiated asynchronously as the result of some event, and that runs in the background. In a MERVA ESA system, such a transaction is typically associated with a MERVA function, and is automatically started by MERVA when a message is routed to that function’s queue.

Input to the transaction is the MERVA terminal user control block (TUCB), the structure of which defined by the copybook DSLAPITU. This structure provides some data about the transaction. You read this data using the facilities of your transaction monitor (CICS or IMS).

Your transaction can use all API services, but as with batch programs your first call must be an INIT call to establish the API environment. If the transaction was automatically started by messages being routed to a queue, use API queue management services to retrieve the messages from the queue.

If the transaction is running in a CICS environment, you must use the API CICS interface routine when invoking DSLAPI. This is discussed in “Alternative API Entry Name” on page 9.

For more information on how to automatically start your transaction, refer to the chapter on coding MERVA ESA applications for automatic start in the *MERVA for ESA Customization Guide*.

An example CICS transaction for automatic start (DSLBA06) is distributed with MERVA ESA and described in “Sample Transaction for Automatic Start” on page 212.

Writing a Conversational Transaction

A transaction dialog typically needs to respond to inputs from a terminal, usually inputs entered by an end user. A *conversational transaction* implements a dialog as a single transaction that processes all steps of a dialog without relinquishing its resources between steps. While the program is waiting for input from the terminal it will be idle, but will continue to occupy main storage.

You can only write conversational transactions under CICS; under IMS, a dialog must be pseudoconversational.

Conversational transactions are no more difficult to write than nonconversational transactions. However, because conversational transactions are long lived and occupy resources for the whole period, even though for most of the time they are idle, you may prefer to consider a pseudoconversational approach.

Writing a Pseudoconversational Transaction

A *pseudoconversational transaction* implements each step of a dialog as a separate transaction. After each response to a terminal, the transaction terminates. The next input from the terminal user invokes the program anew, or might invoke a different program; the various steps of a dialog must not necessarily be processed by one program.

The main considerations when using the MERVA ESA API in pseudoconversational transactions are:

- Preserving data between the steps of the conversation
- Commiting and rolling back updated queue elements

The transaction must preserve data before it terminates so that the data can be restored by the transaction that implements the next step in the conversation. This data might be one or both of:

- The contents of the internal TOF buffer (if you are updating an existing queue element or creating a new element)
- The queue name and QSN of any queue elements you have retrieved

If you have retrieved a queue element but have not updated it, you do not need to save the contents of the internal TOF buffer; instead, you need only save the queue name and QSN in the SPA (IMS) or transient data area (CICS), then reread the element in the next dialog step. However, if you have updated a queue element or created a new message in the internal TOF buffer, then before the transaction terminates you must save the tokenized form of the message in one of the following ways:

- In a queue data set, as described in "Saving the Tokenized Form of a Message in a Queue Data Set" on page 27
- In your own database (IMS) or in temporary storage or transient data (CICS), as described in "Saving the Tokenized Form of a Message in Your Own Database (IMS) or in Temporary Storage or Transient Data (CICS)" on page 28

Note: IMS provides a scratchpad area (SPA) for passing data between steps. However, because the IMS SPA is not intended to hold such large amounts of data as the tokenized form of a message are likely to be, when using IMS you should instead write updates to a queue data set or to your own database between dialog steps.

Saving the Tokenized Form of a Message in a Queue Data Set

Between dialog steps, you can save the tokenized form of a message in a MERVA queue data set (QDS). Define a separate queue for holding only such uncommitted updates. Before terminating each dialog step, use API queue management services to write the updated tokenized form of a message to this temporary queue, and save the record's key (its QSN) in the SPA (IMS) or transient data (CICS). The next transaction can then retrieve the temporary element for the next update step in the dialog.

For example, consider a dialog with the following steps:

1. The terminal user selects a queue element to be updated.
2. The user updates field-A.
3. The user updates field-B.
4. The user indicates the updates should be committed.

Such a dialog could be implemented with three pseudoconversational transactions handling initialization, updates, and termination. Pseudocode for these transactions might look like this.

Initial transaction:

```
read terminal-user input (source queue name, qsn)
API INIT
API GETN read source queue name with lock (exclusive get)
API PUT  write source queue element to temporary queue
API TERM
save source queue name & QSN in SPA
save QSN of temporary queue element in SPA
display element at the terminal
write SPA for next transaction
terminate
```

Update transaction:

```
read SPA
read terminal-user input (field to be updated)
API INIT
API GETU read temporary queue element (lock is unnecessary)
API WRIT update TOF with new field
API REPL replace temporary queue element with updated TOF
API TERM
display updated element at the terminal
write SPA for next transaction
terminate
```

Termination transaction:

```
read SPA
read terminal-user input (commit updates, or rollback updates)
API INIT
if commit updates
    API GETU read temporary queue element (lock is unnecessary)
    API ROUB, or PUTB, temporary element to target queue with automatic
        delete of source queue element
else /* rollback the transaction */
    API FREE relinquish lock on source queue element
endif
API DELE delete temporary queue element
API TERM
terminate
```

Note the following:

- This dialog can be implemented as three separate programs or as three routines in one program.
- The dialog updates an existing MERVA ESA queue element. Therefore the initial transaction retrieves it with an exclusive get. The element is flagged *in-service* for the duration of the dialog.
- The actual update is carried out by the API ROUB or PUTB in the termination transaction. Until that point updates are applied to the temporary queue element, the original queue element, the "source" element, remains unchanged.
- There is no need to lock the temporary queue element because no users will have access to the temporary queue. The queue is used only by pseudoconversational transactions.
- After each update the temporary queue element is replaced, its QSN remains the same throughout the conversation.
- If the user decides to abandon the updates, the conversation is rolled back by relinquishing the lock on the source element, which has not been changed, and deleting the updated element from the temporary queue.
- If a new queue element is to be created instead of an existing element to be updated, similar logic could be used to save the new message in the temporary queue before committing it to its target queue.
- Because MERVA ESA manages the queue data set independently of CICS or IMS, when you finally commit a queue element update, this commit will not be synchronized with your IMS or CICS commit point.

Saving the Tokenized Form of a Message in Your Own Database (IMS) or in Temporary Storage or Transient Data (CICS)

Alternatives to preserving messages in a QDS are:

- Under CICS, preserving them in temporary storage or transient data.
- Under IMS, preserving them in your own IMS database, and at the end of the dialog either commit them (permanently apply them to the database) or roll them back (delete them from the database).

Two API functions are available to help you to save and restore the API environment:

SAVL Use this function to transfer the complete API environment, including the internal TOF buffer and queue buffer, to a buffer you supply.

REEN Use this function instead of the INIT function when restarting a pseudoconversational transaction, to initialize the API with the environment saved by the SAVL call.

These functions are described in "Chapter 9. DSLAPI Functions" on page 97.

For example, if the SAVL and REEN services were being used, the pseudocode for the update transaction in the previous section would look like this:

```
read SPA
read terminal-user input (field to be updated)
read saved API environment from CICS TS / IMS DB
API REEN initialize API with saved environment
API WRIT update TOF with new field
API SAVL the API environment
API TERM
```

```
                              write saved API environment to CICS TS / IMS DB
                              display updated element at the terminal
                              write SPA for next transaction
                              terminate
```

Chapter 4. Writing an MFS Exit Routine in C, COBOL, or PL/I

MFS exit routines play an important part in the processing of messages by MERVA ESA. Such routines are typically used to check or edit fields in messages, but can also be used for many other purposes. You write such a routine as a normal main program that is passed a parameter list using standard OS conventions. MFS exit routines are described in more detail in the *MERVA for ESA Customization Guide*.

In MERVA ESA, exit routines can be written in the low-level language Assembler, and in any of the following high-level languages (HLLs):

- C (for OS/390® and VSE)
- COBOL (for OS/390 and VSE)
- PL/I Version 2 (for OS/390 only); PL/I Version 1 is not supported

You indicate to MFS that your exit routine is written in an HLL by specifying LANG=COBOL, LANG=PLI, or LANG=C (as appropriate) in the MFS program table. For details, refer to the description of macro DSLMPT in the *MERVA for ESA Macro Reference*.

Your exit routine can use the MERVA API to invoke only MFS or TOF services (***the title refers to ″MFS exit routines″, but are also TOF exit routines. Should I drop reference to MFS in title?) , not queue management or other services. If your HLL exit routine uses MFS or TOF services, there is one crucial difference between it and the API program discussed in "Chapter 3. Writing an API Program" on page 23: your routine should *not* initialize an API environment. By its nature, an exit routine is embedded in the MERVA ESA environment, so when your exit receives control the API environment has already been established by MERVA. Consequently, do not begin your exit program with an API INIT call, and do not end it with a TERM call.

The parameter list passed to your HLL exit routine is set up by MERVA MFS and contains two parameters:

- The address of the API interface working storage, INTWSTOR (see "Interface Working Storage INTWSTOR" on page 81).
- The MFS parameter list, MFSL (see "MFS Parameter List" on page 95). This is the basic interface between MERVA ESA and an MFS exit routine, whether a HLL exit or a macro level exit. This structure is available to HLL exit routines in the copybook DSLMFSPL, and to Assembler exit routines via the macro DSLMFS.

Restrictions for Exit Routines Written in an HLL

When writing an HLL exit routine, remeber that it will be embedded in MERVA, hence the API interface working storage (INTWSTOR) that is passed to the exit routine represents the internal MERVA environment. When modifying this working storage, it is possible for you to modify the MERVA internal state in unpredictable and undesirable ways. In particular, do not use queue management services, because this will destroy MERVA's queue position.

If you want to access a MERVA queue, you must define your own API environment. You do this just as you would in any other API program: by defining your own interface working storage area (the INTWSTOR structure), and invoking the API INIT service to initialize it. You can then invoke any API services with this INTWSTOR. Your exit routine will thus have two INTWSTOR structures: one passed to it by MERVA ESA that allows only a restricted set of API calls, and another one (your own), with which you can invoke all API services. However, bear in mind that some exit routines might be invoked frequently, and that using queue services, CICS, or IMS services might reduce performance.

Apart from not using queue management services (***only QM services, or also other non-TOF and non-MFS services?), there are some other restrictions you need to consider when writing HLL exit routines:

- Your exit routines should be reentrant. If they dynamically allocate storage, they should free it before returning to MERVA. Similarly, if they open any files, or load any modules dynamically, they should close or free them before they return. Not adhering to this rule can reduce performance.
- Your exit routine cannot be linked to MFS; HLL exit routines are always loaded dynamically by MERVA (see the LINK parameter of the DSLMPT macro).
- An HLL exit routine program can have only one entry point.
- A COBOL exit routine cannot be called recursively. This means that your exit routine cannot invoke a MERVA ESA service that, as part of its processing, might invoke your exit routine again.

Writing an Exit Routine That Runs Under CICS and in Batch

One characteristic of an MFS exit routine is that, because MERVA ESA MFS services can be invoked in both an online as well as a batch environment, it must be programmed to run in both environments. This is not a problem if the online environment is IMS, because the interfaces in both cases are the same, but if you use the CICS online environment, your routine must be callable using both CICS and OS linkage conventions. One solution is to write two versions of the routine, but it it usually better (from a maintenance standpoint) to instead code it so that it determines at runtime with which linkage convention it has been called and sets up the parameter addresses accordingly.

In a C/370 CICS program you always have to set up the parameter addresses dynamically, but if your routine is written in COBOL or PL/I, then the way to do this differs depending on whether you use EXEC CICS calls in your routine; that is, on whether your routine is processed by the CICS pre-compiler. This is because with these languages you cannot prevent the pre-compiler inserting CICS parameters in the routine's procedure definition.

The following in an example of a COBOL routine that will be preprocessed by the CICS translator. The translator inserts DFHEIBLK and DFHCOMMAREA as the first two parameters after the USING like this:

```
procedure division using DFHEIBLK, DFHCOMMAREA, intwstor, mfsl.
```

The logic makes use of the fact that when the exit routine is called, MERVA ESA will have set INTFUNC in INTWSTOR to INIT:

```
...
linkage section.
01 dfhcommarea.
   03 ptr-to-intwstor   pointer.
   03 ptr-to-mfspl      pointer.
   copy dslapiws.
...
procedure division using intwstor, mfsl.
```

```
          set address of intwstor to address of DFHEIBLK
          if intfunc = 'INIT'
     *      this is the intwstor structure: batch environment..
            set address of mfsl to address of DFHCOMMAREA
          else
     *      the routine has been invoked in a CICS environment..
            set address of intwstor to ptr-to-intwstor
            set address of mfsl to ptr-to-mfspl
          end-if
      ...
```

You compile and link your HLL exit routines just as you would normally generate a CICS or batch application program. But it may be necessary to link your exit routine in two different ways: one for batch and one for the CICS environment. For example, if you have written a VS COBOL II exit routine, you link with a different COBOL library depending on whether the exit routine will run in batch or online. For batch you specify SYS1.COB2LIB in the SYSLIB concatenation, and for the CICS environment SYS1.COB2CICS. So you will have two different load modules for the one exit routine.

Similarly, with PL/I you need to generate two different load modules. In both cases you use the standard CICS JCL procedure, which causes the PL/I CICS interface module, DFHPL1OI, to be included in the load module. DFHPL1OI must be included in both the CICS and the batch version of your routine. However, DFHPL1OI contains CICS versions of the PL/I runtime modules; when generating the batch version of the exit routine, these must be replaced by the non-CICS versions from the PL/I library. One way you can do this is like this:

```
//CICS    EXEC DFHEITPL
//LKED.SYSLIB  DD  DSN=SYS1.PLIBASE,DISP=SHR
//             DD  DSN=SYS1.SIBMBASE,DISP=SHR
//             DD  DSN=MERVA.SDSLLODC,DISP=SHR
//             DD  DSN=MERVA.SDSLLODB,DISP=SHR
//             DD  DSN=CICS410.SDFHLOAD,DISP=SHR
//LKED.SYSLMOD DD  DSN=your load-library,DISP=SHR
//LKED.SYSLIN  DD  DSN=&&LOADSET,DISP=(OLD,DELETE)
//             DD  DDNAME=SYSIN
//             DD  DUMMY
//LKED.SYSIN DD  *
  INCLUDE SYSLIB(IBMBPIRA)
  REPLACE IBMBPIRA,IBMBPIRB,IBMBPIRC
  INCLUDE SYSLIB(DFHPL1OI)
  NAME DSLMS911(R)
```

For more information on compiling and linking your exit routine, refer to the programming guide for the corresponding language.

Field-Level Access for Exit Routines

Product-Sensitive Programming Interface

Mappings of MERVA ESA internal data structures are provided only by the Assembler macros described in the *MERVA for ESA Macro Reference*. An exit routine written in an HLL retrieves and sets fields in MERVA ESA internal data structures by using two field-level access functions provided by DSLAPI: FLDG and FLDP. These functions provide a structure-independent way of accessing these fields.

End of Product-Sensitive Programming Interface

Supported Fields

Product-Sensitive Programming Interface

Most fields in the following structures can be retrieved; many of them can also be set:

DSLCOM	MERVA ESA service communication area
DSLPRM	MERVA ESA customization parameter module
TUCB	Terminal user control block
DSLNIC	Nucleus intertask communication parameter list
DSLMFS PL	MFS parameter list
DSLMFS PS	MFS permanent storage
DSLMFS TS	MFS temporary storage
DSLMFS FLDREF	MFS field reference
DSLUSR	User file record
DSLFNTT	MERVA ESA function table

A high-level language mapping of the DSLMFS parameter list is provided by structure MFSL, copybook DSLMFSPL; it is included in the field level service to allow COBOL programmers access to bit indicators in the structure.

Refer to "Appendix D. Field-Level Access Fields" on page 269 for a list of all fields that can be accessed using field-level services. The name of each field is given, its data type, its length, a brief description of the field, and an indication whether the field may be modified.

To access any particular field you merely specify the field name, you do not provide the address of the structure containing the field. DSLAPI retrieves the address of the structure from anchors in the INTWSTOR structure.

End of Product-Sensitive Programming Interface

Field Values

Product-Sensitive Programming Interface

In addition to the field name, you provide a buffer, without a prefix, for the field's value. When retrieving a field, the value is placed in this buffer, left-justified. Similarly, when setting a field's value, you place the value in this buffer before invoking the FLDP function. The length of the buffer depends on the length of the field's value.

The value has one of the following forms depending on the data type of the specified field:

Data type	**Value**
character	A character string. The length of each string is given in "Appendix D. Field-Level Access Fields" on page 269.
binary	A 4-byte, fullword, binary value regardless of whether the field defines a 1-, 2-, or 4-byte signed or unsigned value.

bit	The character 0 or 1. The field name may define a 1-bit or multiple-bit pattern. When retrieving a bit or bit-pattern, if all bits defined by the pattern are on, a 1 is returned, otherwise a 0 is returned.
byte	Eight 0 or 1 characters.
packed	A packed-decimal value. The length of the value is given in “Appendix D. Field-Level Access Fields” on page 269.

When setting a field’s value, the value you supply must have the appropriate form. When setting a bit-pattern, a 1 will turn on all bits defined by the pattern; a 0 will turn all bits off.

Refer to “FLDG Get a MERVA Variable” on page 105 and “FLDP Set a MERVA Variable” on page 107 for the definition of the field-level access functions.

End of Product-Sensitive Programming Interface

MERVA Buffer Prefix Manipulation

In “Chapter 1. Introduction and Concepts” on page 3, the MERVA buffer prefix is discussed for buffers of up to 32KB. MERVA ESA also uses this standard buffer prefix internally for buffers larger than 32KB, however the format used by MERVA ESA cannot easily be used by application programs written in high-level languages. For a description of the format, refer to the chapter on the MERVA ESA buffer standard in the *MERVA for ESA Customization Guide*.

Because your HLL exit routines might need to manipulate such a buffer prefix, the following service programs have been defined that allow you to set and retrieve the buffer length and data length values regardless of their size:

DSLAPBSB	Set buffer length
DSLAPBGB	Get buffer length
DSLAPBSD	Set data length
DSLAPBGD	Get data length.

All of these programs use the same parameter syntax:

```
►►──DSLAPBxx──(──buffer──,──length──)──────────────────────────────►◄
```

The parameters are:

buffer	A buffer containing a MERVA buffer prefix.
length	A fullword binary variable. It contains either the length to be set into the buffer prefix (DSLAPBSB or DSLAPBSD), or the length value from the buffer prefix will be returned in it (DSLAPBGB or DSLAPBGD).

Notes:

1. These services are implemented using the standard OS call interface. The address of the buffer and the address of the length value are passed to the service program in a parameter list addressed by general purpose register 1.
2. The format of a MERVA buffer prefix is defined by the structure BUFFER_PREFIX in copybook DSLAPIBP.

3. If a buffer cannot be larger than 32KB, then you do not need to use these services; you can set and retrieve the length values directly using the BUFFER_PREFIX variables.
4. If the buffer can be larger than 32KB, then you must access the prefix using these services.
5. Under CICS, these routines must be linked to your application, but because these routines are very small, even if not running under CICS, you might want to statically link these routines to your application.

Chapter 5. The REXX Interface

In addition to COBOL, PL/I, C, and Assembler, MERVA ESA also lets you write API applications in REXX, which is particularly useful for experimenting with the API, prototyping, and writing quick, one-shot programs. Your REXX applications can also take advantage of the particular strengths of the REXX language, for example, its mathematical capabilities and its functions, and they can use any host command environments and function packages available in your system.The REXX API interface is supported under MVS and VSE, but cannot be used in MFS exit routines.

In addition to the information shown in this chapter:
- Sample REXX EXECs are shown in “Appendix B. Sample Programs” on page 203.
- Batch utilities written in REXX are shown in “Appendix C. Batch Utilities in REXX” on page 227.

Overview

The REXX support in MERVA ESA interfaces REXX to the MERVA API program DSLAPI. You use DSLAPI services in REXX very much as you use DSLAPI services in other languages.

The interface is implemented as a REXX host command processor. The host command processor has the same name as the MERVA ESA API interface: DSLAPI. You access a host command environment from a REXX EXEC using the REXX ADDRESS command:

```
    Address DSLAPI "READ"     /* DSLAPI TOF read function */
```

This causes REXX to pass the command 'READ' to the DSLAPI host command processor.

All MERVA ESA services supported by the DSLAPI program for conventional languages are also supported as REXX host commands.

To execute a MERVA REXX application you run the MERVA ESA program DSLAREXX passing to it the name of your REXX EXEC. DSLAREXX prepares a REXX language processor environment and then calls your REXX EXEC. You can invoke DSLAREXX as an MVS or VSE batch program, or from TSO or ISPF.

If you have the REXX/370 Compiler (5695-013) and Library (5695-014) installed, it is a good idea to compile your larger MERVA execs.

Variables

The variables in the DSLAPI parameter lists and structures that control the various API functions and that contain the results of API calls are available to the REXX EXEC; the variable names are the same in REXX and behave as described in “Chapter 8. Data Structures” on page 81. For example, the queue management GET function looks like this in REXX:

```
...
intqueue = 'L3ACKF'
intqsn   = 123
Address DSLAPI "GET"
If intrc ¬= ' ' | intbusy = 'BUSY' Then ...
...
```

This retrieves a message from the MERVA ESA queue L3ACKF using the DSLAPI GET function. After the GET the REXX variable INTRC contains the value from the INTRC field in the INTWSTOR structure, so the DSLAPI return code can be checked just as it would be using any of the other languages supported by the MERVA ESA API.

Note, however, that structures are not used by the REXX interface. For example, INTRC is part of the INTWSTOR structure but you cannot reference INTWSTOR in your REXX program. You can only use the individual fields in the INTWSTOR structure.

Here is a list of all the INTWSTOR variables that are available in REXX:

INTBQSN	Input	INTERMF3	Output	INTMSGID	Input
INTBQUE	Input	INTERMSG	Output	INTQSN	Input/Output
INTBUSY	Output	INTFRMID	Input	INTQUEUE	Input
INTDOUBL	Input/Output	INTFUNC	Input	INTRC	Input/Output
INTERMF1	Output	INTKEY1	Input/Output	INTSHUTD	Output
INTERMF2	Output	INTKEY2	Input/Output		

Some of them are only for input to DSLAPI (they are not used for passing results back to the REXX EXEC), some are only for output information (their values are not passed to DSLAPI), and the rest are used for both input and output.

Initializing Variables

The REXX convention for initializing variables applies: before a variable has been initialized, its value is its own name in uppercase. For example, if you use the queue management GET function and forget first to set the variable INTQUEUE, DSLAPI will attempt to read a message from the MERVA ESA queue with the name INTQUEUE, which probably does not exist.

However, if a variable that should have a numeric value has its own name as its value, MERVA ESA treats it as if it had the value zero. For example, if INTQSN were not initialized, it would have the value INTQSN, but MERVA ESA would recognize that the value and name were identical, and a GET would result in MERVA ESA reading from the beginning of the queue specified by INTQUEUE.

To trap uninitialized variables, it is considered good REXX programming practice to always set `Signal On Novalue`.

Fixed-Length Variables

REXX variables that are passed to DSLAPI fixed-length character fields, for example, the INTWSTOR variables above, are padded on the right with blanks if they are too short, or truncated on the right *without an error indication* if they are too long. So you can set a field to blanks by assigning a null-string to it, for example:

```
intkey1 = ''
```

Variable-Length Variables

When variable-length values are passed to DSLAPI, they have the length of the REXX variable. For example, to write a 4000-byte journal record you could set the journal record string like this:

```
jrnrcord = Left('a journal record consisting mostly of blanks',4000)
```

On the other hand

```
jrnrcord = ''          /* a null string */
```

defines a journal record of length zero (which is valid, it would consist merely of its key).

After a variable-length string has been returned to REXX from DSLAPI, you use the REXX LENGTH function to get its length. The REXX interface does not use buffer prefixes or separate length variables to indicate the length of strings.

The REXX interface itself does not restrict the size of variables; the only restrictions are those determined by your MERVA ESA parameters module DSLPRM and system storage constraints.

Case

The REXX interface does not modify the case of variables; the rules of REXX and DSLAPI apply. In the following example INTQUEUE is assigned an uppercase name (if variable l3de0 has not yet been defined) and INTBQUE lowercase.

```
intqueue = l3de0
intbque  = 'l1de0'
```

The DSLAPI program folds the values of the following variables to uppercase if they are specified in lowercase or mixed case:

- INTQUEUE
- INTBQUE
- INTFUNC
- INTMSGID
- INTFRMID

Other variable values are not folded to uppercase and must be specified as they appear in the MERVA ESA queue data set.

DSLAPI host commands can be specified in uppercase or lowercase.

Return Codes

After invoking the DSLAPI Host Command Environment, return codes are available from the Host Command Environment, and from DSLAPI.

Return Codes from the DSLAPI Host Command Environment

The standard REXX return code RC is set by the host command environment:

RC=0	No error: INTRC contains ' ', '08', or '09'.
RC=-1	INTRC contains '00', '01', or '02'.
RC=-2	A REXX variable was not numeric but should have been (for example, INTQSN, or TOFFDOC) or has an invalid value (for example, INTDOUBL).
RC=-3	Unknown host command.
RC=-8	A GETMAIN failed.
RC=-12	Internal storage overflow. This indicates an error in MERVA ESA code.

REXX Condition Traps

The host command processor sets the ERROR condition trap if the REXX RC variable is negative and in the range –1 to –7. The FAILURE condition trap is set if REXX RC contains any other negative value.

DSLAPI Return Codes

After control returns to the REXX EXEC, the following DSLAPI return values are available: INTRC, INTERMSG, INTSHUTD, and, depending on the DSLAPI function, INTBUSY, INTDOUBL, INTERMF1, INTERMF2, and INTERMF3.

It is not possible to receive a return code of INTRC='03', record truncated, in REXX.

The REXX Language Processor Environment

A REXX EXEC runs in a language processor environment. Program DSLAREXX, which initializes the MERVA ESA REXX API interface, establishes a language processor environment using the default values defined in your system's REXX parameter modules. See *TSO Extensions V2 REXX/MVS Reference* for a description of these parameter modules and how language processor environments are initialized.

DSLAPI Functions Supported by the REXX Interface

All DSLAPI functions defined in this guide are supported as host commands, except INIT, TERM, SAVE, SAVL, and REEN. INIT and TERM are carried out automatically by the interface so need not be invoked from the EXEC. If they are invoked they have no effect. SAVE, SAVL, and REEN are intended for use by pseudoconversational transactions; you cannot write pseudoconversational transactions in REXX.

When reading this section, it might be helpful to refer to:

- The DSLAPI function descriptions in "Chapter 9. DSLAPI Functions" on page 97
- The descriptions of the DSLAPI data structures in "Chapter 8. Data Structures" on page 81

Queue Management Services

The following variables used by DSLAPI queue management services are available in the REXX EXEC:

```
INTQSN    a numeric value converted internally to binary
INTQUEUE
INTBQSN   a numeric value converted internally to binary
INTBQUE
INTKEY1
INTKEY2
```

For example, the following program updates a message in the Telex data entry queue:

```
...
intqueue = 'TX2DE0'
intkey1 = 101                  /* specific key */
Address DSLAPI "GETK"          /* get queue element by key */
If intrc = ' '                 /* the element is now 'in-service' */
Then Do
   ...                         /*  .. update the element .. */
   Address DSLAPI "REPL"       /* replace the queue element */
   Address DSLAPI "FREE"       /* relinquish 'in-service' status */
End
...
```

TOF Services

The REXX variables for TOF services are the variables from the TOFPARM structure:

```
TOFFDDA    numeric             TOFFDOCA      numeric
TOFFDFG    numeric             TOFFDOC.n     numeric
TOFFDNAM                       TOFMODIF
TOFFDNL    numeric             TOFTSVRC
TOFFDOC    numeric             TOFTSVRS
```

In addition, the REXX variable TOFDATA is used for the TOF field data. This variable does *not* contain the 8-byte buffer prefix of the TOF field buffer. The TOFPARM structure is described in “Chapter 8. Data Structures” on page 81. The fields TOFREQ and TOFFDNA1 are not used.

The indexes for nested repeatable sequences are defined using the compound variable TOFFDOC.*n*, where *n* is a number from 1 to 9. When writing a field in a nested repeatable sequence, put the highest value of *n* you are using into the variable TOFFDOCA. You must also specify the RSEXT repeatable sequence extension modifier in TOFMODIF to indicate that you are using nested repeatable sequences.

This example updates a field of a message in the TOF:

```
...
toffdnam = 'ENLTXREF'           /* the TX2DE0 queue key    */
tofmodif = 'VFIRST'
Address DSLAPI "READ"           /* read field from the TOF */
Say 'fld ref of' toffdnam 'is' toffdnl toffdfg toffdoc toffdda
If Datatype(tofdata,'N')        /* is key numeric          */
Then Do
   tofdata = tofdata + 1        /* increment the key       */
   Address DSLAPI "WRIT"        /* and put it back into the TOF */
End
...
```

Message Format Services

The REXX variables you use with MFS services are based on the MSGSWIFT structure in copybook DSLAPIMS. The message string is moved to and from the REXX variable MSGSMSG. The variable contains no length prefix; the length of the message is the length of the variable.

The MFS parameter fields in the INTWSTOR structure are also used: INTMSGID and INTFRMID.

If you retrieve the MSGSWIFT prefix (with the GETS, GETM, or MPFG services), the following variables are set:

```
MSGACK              MSGADDR3             MSGDST            MSGUSER1
MSGADDR1            MSGADDR4             MSGMTYPE
MSGADDR2            MSGDBS               MSGNET
```

If you use the PUTS, PUTM, or MPFP services to move MSGSWIFT prefix fields into MERVA ESA, you must set the following variables:

```
MSGACK               MSGADDR2             MSGADDR4          MSGUSER1
MSGADDR1             MSGADDR3             MSGDBS
```

To retrieve a SWIFT II message from MERVA ESA you could use the MSGG command:

```
...
Address DSLAPI "GET"        /* get message into DSLAPI queue buffer */
If intrc ¬= ' ' | intbusy = 'BUSY' Then Signal Get_failed
intmsgid = ''               /* default msg type */
intfrmid = 'W'              /* SWIFT II format  */
Address DSLAPI "MSGG"       /* map queue buffer to msgsmsg */
If intrc ¬= ' ' Then Signal Msgg_failed
Say 'length of SWIFT II message is' Length(msgsmsg)
...
```

To import a message into MERVA ESA use the MSGP command:

```
...
intmsgid = 'TELEX'          /* msgsmsg contains a Telex */
intfrmid = 'P'              /* ... in the workstation based Telex format */
Address DSLAPI "MSGP"       /* map the Telex to the DSLAPI queue buffer */
...
```

Print Services

The API print services are:

PRTI Intializes the printing environment

PRTL Formats the message currently in the internal buffer and returns the message line by line in the variable PRTLINE

PRTT Terminates the printing environment

For example:

```
/* 1. PRTI - Initialize printing environment */

intfrmid = 'E'                          /* language ID is English    */
intmsgid = ' '                          /* default message type      */
Address DSLAPI "PRTI"
If intrc ¬= ' ' Then ...

/* 2. FLDP - Customize printing environment */
fldname  = 'TUCFRAMT'                   /* use top frame 0TOP        */
fldvalue = '0TOP    '
```

```
Address DSLAPI "FLDP"
If intrc ¬= ' ' Then ...

fldname  = 'TUCNAME'                    /* show function name        */
fldvalue = 'L1DE0   '
Address DSLAPI "FLDP"
If intrc ¬= ' ' Then ...

/* 3. GET - Read a queue element */
intqueue = 'L1DE0'                      /* queue name                */
intqsn   = 123                          /* queue sequence number     */
Address DSLAPI "GET"
If intrc ¬= ' ' Then ...

/* 4. WRIT - Write TOF field DSLSDYNO - running no. */
TOFDATA  = '00001'                      /* data to be written        */
TOFFDNAM = 'DSLSDYNO'                   /* name of the field         */

TOFMODIF = 'VFIRST'                     /* request modifier          */
Address DSLAPI "WRIT"
If intrc ¬= ' ' | toftsvrc ¬= 0 | toftsvrs ¬= 0 Then ...

/* 5. PRTL - Print message line by line */
intrc = ' '                             /* init PRTL rc              */
Do While intrc = ' '                    /* loop while PRTL rc = ' '  */
   Address DSLAPI "PRTL"
   If intrc = ' ' Then Say prtline
End

/* 6. PRTT - Terminate printing environment */
Address DSLAPI "PRTT"
If intrc ¬= ' ' Then ...
```

Journal Services

The following journal key variables are available in the REXX EXEC:

- JRNKDATE
- JRNKTIME
- JRNKUSER
- JRNRID

If you have defined journal segmentation in your MERVA ESA parameter module DSLPRM, then the segment number and count are also returned in variables JRNKSEG and JRNKSEGS.

The journal record is read into or from the REXX variable JRNRCORD; it does *not* contain the 8-byte buffer prefix.

Note: The JRNRID field in REXX is a 1-byte binary value. Use the REXX function D2C, or hexadecimal notation, to set it:

```
...
decimal_id = 12
jrnrid = D2c(decimal_id,1)         /* jrnrid has the value X'0C' */
Say 'jrnrid is' C2d(jrnrid)        /* says "jrnrid is 12"    */
Say "jrnrid is X'"C2x(jrnrid)"'"    /* says "jrnrid is X'0C'" */
...
```

In the following example, the time field will be padded with blanks. Following the JRLG call, the REXX journal key variables contain the key of the record read. These values are then not changed, and are used as key by the JRLN function, which retrieves the next sequential record into the REXX variable JRNRCORD:

```
...
jrnkdate = 990130
jrnktime = 1200
Address DSLAPI "JRLG"    /* get 1st journal record after 12 noon */
Address DSLAPI "JRLN"    /* gets the subsequent record */
Say 'read journal record with key' C2x(jrnrid) jrnkdate jrnktime jrnkuser
...
```

Following a JRLP or JRNP, the key of the record created is returned in the REXX journal key variables:

```
...
jrnkuser = ''            /* null, ie. blank user field */
jrnrid   = '99'x         /* journal id is '99'x */
jrnrcord = 'a private journal record written on' Date() 'at' Time()
Address DSLAPI "JRLP"    /* writes a record to the journal */
Say 'wrote journal record on' jrnkdate 'at' jrnktime
...
```

User File Services

The API USRG and USRN services require as input a userid in the variable USRKEY, and return the user file record in the variables defined in the DSLUSRS structure in copybook DSLAPIUS, as shown in "Chapter 8. Data Structures" on page 81. The variables USRUFTAB and USRUAMSG are returned as compound variables with a numeric suffix: USRUFTAB.*n* and USRUAMSG.*n*. The variables USRUFTAB.0 and USRUAMSG.0 contain the number of variables returned, in other words, the highest value of *n* you can specify. The fields in the user file pending area cannot be accessed.

```
...
usrkey = 'MASADM'
Address DSLAPI "USRG"
Say 'User' usrkey 'can use' usruftab.0 'functions:'
Do i = 1 To usruftab.0
   Say Format(i,2) usruftab.i
End
...
```

Command Service

The DSLAPI Command service uses the REXX variables:

```
CMDINP
CMDRESP
```

For example:

```
...
cmdinp = 'DQ filled'
Address DSLAPI "CMD"
If intrc = ' '
Then Do
   Do i = 0 To 9
      Say Substr(cmdresp,1 + i * 70,70)   /* display the response */
   End
End
...
```

WTO Service

To use the WTO service, place the operator message in the REXX variable WTOMSG:

```
...
wtomsg = 'this is my operator message'
Address DSLAPI "WTO"
...
```

Field-Level Access Services

Product-Sensitive Programming Interface

You can also use the field-level access services in REXX to retrieve MERVA-internal data. The REXX variables used are FLDNAME and FLDVALUE:

```
...
fldname = comtrata
Address DSLAPI "FLDG"
Say 'trace table is at A(' || C2x(Substr(fldvalue,1,4)) || ')'
...
```

End of Product-Sensitive Programming Interface

The SNAP Command

In addition to DSLAPI functions, the REXX interface provides a SNAP command, which writes the name and content of all accessible REXX variables to the output data set used by the REXX SAY command (usually SYSTSPRT). This can be useful when debugging an EXEC.

Only the first 20 characters of the name and the first 80 characters of the value are output. Nonprintable characters, for example, binary values, are output as a period (‘.’).

You invoke the SNAP command as you would any other host command. For example, if the MERVA ESA system is not active (remember CMD uses MERVA central services), the following program:

```
cmdinp = 'Du'
Address DSLAPI "CMD"
Address DSLAPI "SNAP"
```

results in the following being written to SYSTSPRT:

```
Snap of all REXX variables
  RC                    >-1<
  INTRC                 >02<
  CMDINP                >Du<
  INTERMSG              >DSL884I NIC04000 NIC=CMD RC=04 ...
  INTSHUTD              >INACTIVE<
  INTFUNC               >CMD <
End of snap of all REXX variables
```

Sample Programs Written in REXX

MERVA ESA provides the following sample programs and batch utilities in REXX:

DSLBAnnR These sample programs and batch utilities are described in “Appendix B. Sample Programs” on page 203 and “Appendix C. Batch Utilities in REXX” on page 227.

DSLSDxR These batch utilities are described in the *MERVA for ESA Operations Guide*.

Running a REXX EXEC under MVS

The interface is implemented as a batch program, DSLAREXX. To execute an EXEC you execute this program and pass the name of the REXX EXEC to be run in the EXEC PARM:

```
//MERVAREX EXEC  PGM=DSLAREXX,PARM='MYEXEC'
```

The REXX EXEC is loaded from the partitioned data set with DDname SYSEXEC. This EXEC can in its turn call other REXX EXECs, which will also be loaded from SYSEXEC.

If the parameter is omitted, the REXX EXEC will be loaded from SYSIN, but note that you cannot use the form `SYSIN DD *`. "JCL to Execute an Instream EXEC" on page 47 shows how to use an instream EXEC. An EXEC loaded from SYSIN cannot load other EXECs.

You can also run DSLAREXX from TSO using the TSO CALL command if the MERVA ESA library is in the search sequence of the MVS LOAD macro. The library could be in your logon STEPLIB or in the link library:

```
CALL 'MERVA.SDSLLODB(DSLAREXX)' 'MYEXEC'
```

REXX Input and Output Streams

Standard input and output streams can be used by REXX EXECs:

SYSTSIN	The PARSE EXTERNAL, or PULL command, will read a string from this DDname if the stack is empty. If SYSTSIN is empty, PULL returns a null string.
SYSTSPRT	SAY writes a string to the DDname SYSTSPRT. REXX Trace output is also written to SYSTSPRT.

Passing Parameters to the EXEC

Passing Parameters in the PARM Statement

You can pass a parameter string to the EXEC as the second parameter in the EXEC PARM:

```
//MERVAREX EXEC  PGM=DSLAREXX,PARM='MYEXEC,A PARAMETER STRING'
```

or under TSO:

```
CALL 'MERVA.SDSLLODB(DSLAREXX)' 'MYEXEC,A PARAMETER STRING'
```

The parameter string may contain blanks, but must not contain a comma. A comma is taken as the end of the string.

In the EXEC you access the parameters with PARSE ARG or just ARG:

```
...
Parse Arg arg1 arg2 arg3 .              /* get three args */
Say 'Arg1:' arg1
Say 'Arg2:' arg2
Say 'Arg3:' arg3
...
```

Passing Parameters via SYSTSIN

Another way to pass parameters to the EXEC is via SYSTSIN.

```
//SYSTSIN  DD *
              Parameter line 1 for MYEXEC
              Parameter line 2 for MYEXEC
              Parameter line 3 for MYEXEC
/*
```

The parameter lines may contain blanks and commas.

In the EXEC you access the parameters with PARSE PULL or just PULL:

```
...
Do i = 1 By 1                          /* read parameter lines into */
   Parse Pull systsin_data.i           /* stem var systsin_data.    */
   If systsin_data.i = '' Then Leave
End
systsin_data.0 = i - 1
Say 'Total supplied parameter lines:' systsin_data.0
...
```

JCL to Run EXECs from a PDS

You could use JCL like the following to execute a REXX EXEC from the PDS MERVA.REXX.EXEC. The EXEC uses PARSE PULL to read data from the REXX input stream SYSTSIN:

```
//REXXB    EXEC PGM=DSLAREXX,PARM='MYEXEC'
//STEPLIB  DD   DSN=MERVA.SDSLLODB,DISP=SHR
//SYSEXEC  DD   DSN=MERVA.REXX.EXEC,DISP=SHR
//SYSPRINT DD   DUMMY
//SYSTSPRT DD   SYSOUT=*                     REXX 'SAY' AND 'TRACE' OUTPUT
//SYSTSIN  DD   *                            REXX 'PULL' INPUT
this is input record 1
...
/*
//
```

"Appendix C. Batch Utilities in REXX" on page 227 contains more sample JCL statements.

JCL to Execute an Instream EXEC

The following JCL could be used to run an EXEC from SYSIN:

```
//COPY     EXEC PGM=IEBGENER
//SYSPRINT DD   SYSOUT=*
//SYSUT2   DD   DSN=&&EXEC,DISP=(NEW,PASS),UNIT=SYSDA,
//         SPACE=(3200,(400,50)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT1   DD   *,DLM=$$
/*   Rexx MERVA ESA API program   */
intqueue = 'L3ERROR'
intqsn   = 0
Address DSLAPI "GETN"
If intrc = ' '
Then Do
   ...
End
Exit
$$
//REXXB    EXEC PGM=DSLAREXX
//STEPLIB  DD   DSN=MERVA.SDSLLODB,DISP=SHR
//SYSIN    DD   DSN=&&EXEC,DISP=(OLD,DELETE)
//SYSTSIN  DD   DUMMY
//SYSPRINT DD   SYSOUT=*                DSLAREXX AND DSLAPI TRACES
//SYSTSPRT DD   SYSOUT=*                REXX 'SAY' AND 'TRACE' OUTPUT
//
```

Running a REXX EXEC under VSE

The interface is implemented as a batch program, DSLAREXX. To execute an EXEC you execute this program and pass the name of the REXX EXEC to be run in the EXEC PARM:

```
//EXEC DSLAREXX,SIZE=300K,PARM='MYEXEC'
```

The REXX EXEC is loaded from the active LIBDEF sublibrary chain and must have been catalogued with extension .PROC. This EXEC can in its turn call other REXX EXECs, which will also be loaded from the active LIBDEF sublibrary chain.

REXX Input and Output Streams

Standard input and output streams can be used by REXX EXECs:

SYSIPT	The PARSE EXTERNAL, or PULL command, will read a string from SYSIPT if the stack is empty. If SYSIPT is empty, PULL returns a null string.
SYSLST	SAY writes a string to SYSLST. REXX Trace output is also written to SYSLST.

Passing Parameters to the EXEC

Passing Parameters in the PARM Statement

You can pass a parameter string to the EXEC as the second parameter in the EXEC PARM:

```
//EXEC DSLAREXX,SIZE=300K,PARM='MYEXEC,A PARAMETER STRING'
```

The parameter string may contain blanks, but must not contain a comma. A comma is taken as the end of the string.

In the EXEC you access the parameters with PARSE ARG or just ARG:

```
...
Parse Arg arg1 arg2 arg3 .              /* get three args */
Say 'Arg1:' arg1
Say 'Arg2:' arg2
Say 'Arg3:' arg3
...
```

Passing Parameters via SYSIPT

Another way to pass parameters to the EXEC is via SYSIPT:

```
//EXEC DSLAREXX,SIZE=300K,PARM='MYEXEC'
Parameter line 1 for MYEXEC
Parameter line 2 for MYEXEC
Parameter line 3 for MYEXEC
/*
```

The parameter lines may contain blanks and commas.

In the EXEC you access the parameters with PARSE PULL or just PULL:

```
...
Do i = 1 By 1                        /* read parameter lines into */
   Parse Pull sysipt_data.i          /* stem var sysipt_data.     */
   If sysipt_data.i = '' Then Leave
End
sysipt_data.0 = i - 1
Say 'Total supplied parameter lines:' sysipt_data.0
...
```

JCL to Run EXECs

You could use JCL like the following to execute a REXX EXEC from the sublibrary MERVA.PROCS. The EXEC uses PARSE PULL to read data from the REXX input stream SYSIPT:

```
// JOB REXXBAT1 MERVA ESA V410 REXX EXEC CALLS
// DLBL MERVA,'MERVA.PRODUCT.LIBRARY',99/365,SD
// EXTENT ,MERV41
LIBDEF *,SEARCH=(MERVA.LIBS,MERVA.PROCS)
// OPTION NODUMP
// EXEC DSLAREXX,SIZE=300K,PARM='MYEXEC,L1DE0'
this is input record 1
...
/*
//
```

Chapter 6. Advanced Topics

This chapter discusses some aspects of MERVA ESA that you need to consider if you write a MERVA application that is to be linked to the MERVA nucleus, DSLNUC, or if you want to use MERVA ESA macro level services in an API program.

Some MERVA ESA system fields are also described.

Applications Linked to DSLNUC

Assembler application programs linked to the MERVA ESA nucleus (DSLNUC) can also use the MERVA ESA Application Programming Interface. But note that application programs linked to DSLNUC cannot use the user file service functions USRG and USRN.

An application program linked to DSLNUC receives the address of the MERVA ESA service communication area (DSLCOM) in register 12. Before making the DSLAPI initialization call this address must be stored into the INTCWA parameter of INTWSTOR so that DSLAPI has access to the DSLNUC environment. Then all MERVA ESA requests (except USRN and USRG) can be issued by DSLAPI as direct calls as required for applications linked to DSLNUC. Interregion communication is not used.

When DSLAPI is called it checks whether INTCWA contains an address that points to a service communication area. If it does, the provided area is used, otherwise DSLAPI builds its own DSLCOM.

Refer to *MERVA for ESA Concepts and Components* for more information on applications linked to DSLNUC.

Locating the TOF

If you want to manipulate a message in the API internal TOF using MERVA ESA macro services, you can find the address of the TOF in the API interface working storage field INTTOFA. After each API TOF service call you should reload this address because the internal TOF can be dynamically relocated.

Having located the TOF you can use the MERVA macro level TOF service, DSLTSV, to manipulate it. Do not change it in any other way.

Locating the Internal Queue Buffer

If you want to address the API internal queue buffer, you can find its address in the API interface working storage field INTQBUFA. After each API queue management call you should reload this address because the internal buffer may have been relocated.

Having located the Queue Buffer you can process it using the MERVA macro level Queue Management service. Do not change the Queue buffer in any other way.

Customizing the API

Customization of MERVA ESA is described in the *MERVA for ESA Customization Guide*.

DSLPRM Parameters

The following parameters in the MERVA ESA parameters module, DSLPRM, concern DSLAPI and are briefly discussed here:

- APISMSG
- APIUID
- EXDSP.

Refer also to *MERVA for ESA Macro Reference* for more information on macro DSLPARM.

APISMSG

This DSLPRM parameter defines the size of the largest message in net format you want to process using the API MFS services GETM, GETS, PUTM, and PUTS. The size you specify must include 8 bytes for the MERVA buffer prefix. The maximum value you can specify is 32KB minus 4 bytes.

Note: The buffer size DSLAPI uses is 376 bytes longer than APISMSG. This is the size of the MSGSWIFT_PREFIX structure (copybook DSLAPIMP).

APISMSG is not used by the MSGG and MSGP services. You are recommended to use MSGG and MSGP instead of GETS, GETM, PUTS, and PUTM.

APIUID

This DSLPRM parameter defines the user ID used when the CMD, USRG, or USRN API service is journaled.

EXDSP

This DSLPRM parameter can be used to suppress user file access by application programs. If EXDSP=NO has been specified, you cannot use the API USRG and USRN functions.

Runtime Environment Settings

With the following fields you can customize the API runtime environment. Set them with the FLDP function. The fields have the type *BIT*; to switch the bit on, use the character ‘1’ as input data, to switch the bit off, use the character ‘0’ as input data.

APICQBIN

Normally the key fields INTKEY1 and INTKEY2 are treated as character fields, for example, trailing blanks are removed. Switch on APICQBIN if you do not want keys to be manipulated, for example, when your keys contain binary data.

Initially, the bit is switched off.

APICQDIR

This flag is applicable only to queue management using DB2 for MVS. Set this flag to enable direct DB2 queue management calls.

Initially, the bit is switched off and the queue management calls are directed via intertask communication to the MERVA ESA nucleus (recommended).

See “Queue Management Using DB2” on page 57 for details.

APICQLAZ

Set this flag to switch on deferred queue management write requests (‘lazy’). Your application must provide restart logic when it sets APICQLAZ.

Initially, the bit is switched off; all queue management requests are executed immediately.

APICQMIT

This flag is applicable only to direct DB2 queue management. When this flag is on, all queue management requests are committed immediately. When this flag is off, queue management requests are not committed. It is the responsibility of the application to commit or rollback database changes then.

Initially, the bit is switched on. If you want to switch off the flag, you must do this before the very first queue management request.

APICQWRB

This flag can be used to switch off the use of the DOUBLE (write-back) indicator. When the flag is switched off, the indicator need not be updated in the QDS after a get function, therefore such a request can be executed faster.

Initially, the bit is on and the DOUBLE indicator is written when a message is accessed exclusively (GETN, GETK, and GETC functions).

APICMCLR

Set this flag to clear the internal TOF totally before the (next) message is mapped into the TOF (MSGP, PUTM, and PUTS functions).

Initially, the bit is switched off; the internal TOF is not cleared of system fields (the fields with nesting level 0) when a message is mapped into the TOF.

APICMCHK

The MERVA ESA MFS message checking can be enabled or disabled before a message is mapped from the internal buffer to an external format (functions MSGG, GETM, and GETS), or from an external format to the internal buffer (functions MSGP, PUTM, and PUTS). Switch off APICMCHK if you do not want messages to be checked.

Initially, the bit is switched on.

MERVA Fields in Messages

A message in the MERVA internal format contains some MERVA control fields in addition to the message itself. These fields are accessible just like any other fields using API TOF services. MERVA fields are at nesting identifier 0 in the TOF. The following MERVA control fields are described in the following sections:

- MSGTRACE
- Exit fields
- DSLMSG
- UMR (unique message reference)

Additionally, some MERVA system fields that are not in the TOF can be read as if they were in the TOF at level 0. An example is the date and time in various formats.

All fields known to MERVA ESA are defined in the Field Definition Table; refer to this table for a list of all MERVA system fields.

MSGTRACE

The MSGTRACE field is intended to record the stages through which a message passes in MERVA ESA. For example, each time a message is routed from one queue to another, MERVA records the event with an entry in the MSGTRACE field. An 'entry' means simply that a new data area is added to the field.

If you write an application to manipulate MERVA messages, your program should also write a MSGTRACE entry. For an example of how to create and write a MSGTRACE entry refer to sample programs DSLBA04 in "DSLBA04x" on page 208 and DSLBA10R in "DSLBA10R - Update a Queue Element" on page 210.

Refer to the copybook DSLFDTTC in the MERVA ESA distributed material for the definition of the MSGTRACE field structure, that is, its subfields.

Message Exit Fields

Exit fields exist at each nesting level in a message and define the message identifier of the message at the next, embedded, level. Even a message at the highest level, nesting level 1, is embedded in the MERVA system, level 0, so the type of any message can be found by reading the exit field with nesting identifier 0. Exit fields are defined in an MCB using the DSLLEXIT macro; the default name of a MERVA ESA exit field is DSLEXIT.

However, at nesting level 0 other exit fields are also used. If a SWIFT message is at nesting level 1, its message type is contained in the standard exit field, DSLEXIT, at level 0. If a Telex Link message is at level 1, its message identifier is in exit field ENLEXIT at level 0. Which exit field at level 0 you use is identified by a second exit field, NLEXIT: NLEXIT contains the name of an Exit field.

If a Telex Link message at level 1 contains a SWIFT message, then all three exit fields exist at level 0: NLEXIT contains ENLEXIT, the field ENLEXIT identifies the Telex cover MCB, TCOV, which is used to map Telex messages that contain a SWIFT message, and DSLEXIT identifies the SWIFT message, for example, S100.

If you used MERVA to create a SWIFT message in a Telex, you can extract the SWIFT message from the Telex using the API MFS MSGG or GETM function and specifying the message type from the DSLEXIT field as the INTMSGID value.

Since the SWIFT message type is contained in the standard exit field, DSLEXIT, the same effect can be obtained simply by specifying the standard MERVA cover MCB, 0COV, in the INTMSGID parameter. To understand this you will need to look at the MCB code. The 0COV message type is defined in the MCB DSL0COV.

DSLMSG

The field DSLMSG contains error messages from message checking. Following importation of a message into MERVA ESA by MFS, this field will contain any error messages generated by MFS field checking exit routines. Each message occupies one data area in the field. The first three data areas are returned in the INTWSTOR fields INTERMF1-3.

UMR - Unique Message Reference

You can configure MERVA ESA to assign a unique message reference (UMR) to a message processed by MERVA queue management by specifying UMR=YES in the MERVA ESA parameter module DSLPRM. Refer to *MERVA for ESA Concepts and Components* for a discussion on the UMR.

The UMR can be accessed using API TOF services by retrieving the field DSLUMR. The length of the UMR field is 28 bytes, the subfields that make up the UMR are defined in the Field Definition table, copybook DSLFDTTC, in the MERVA ESA distributed material.

You must not change a UMR assigned by queue management or remove a UMR from a message. However, when you create a new message by duplicating an existing message, for example, by using API GET and PUT, the UMR will also be copied. So that a new UMR can be assigned to this new message the existing UMR must be deleted from the internal TOF buffer.

After the GET and before the PUT you would use the DSLAPI TOF service EMPT to delete the field:

```
INTFUNC  = 'EMPT';
TOFFDNAM = 'DSLUMR';
TOFMODIF = 'DELFN';
TOFFDNL  = 0;
TOFFDFG  = 1;
TOFFDOC  = 1;
TOFFDDA  = 1;
CALL 'DSLAPI' USING INTWSTOR TOFPARM
```

Then, when the message is stored into the MERVA ESA queue data set and no UMR exists in the message, a new UMR is assigned by MERVA ESA queue management. As a new UMR is not created until the message is stored into the MERVA ESA queue data set, this UMR is not available for the application during processing of the message.

If the application requires that the UMR is assigned in advance to be used when processing the message, the field DSLUMRGT can be read. Use the DSLAPI TOF service READ to read the field DSLUMRGT to extract the current UMR or, if it does not exist, create a new one. The UMR is presented in the TOF buffer.

```
INTFUNC  = 'READ';
TOFFDNAM = 'DSLUMRGT';
TOFMODIF = ' ';
TOFFDNL  = 0;
TOFFDFG  = 1;
TOFFDOC  = 1;
TOFFDDA  = 1;
CALL 'DSLAPI' USING INTWSTOR TOFPARM BUFFER
```

Read the field DSLUMRNW to force a new UMR to be created even if a UMR exists in the message.

External Line Format

MERVA ESA allows the processing of messages in external line format (ELF) in addition to the traditional tokenized format (TOF). The use of the external line format can save CPU time because the mapping is less costly and the resulting message length in the MERVA ESA queue buffer is usually smaller due to a reduced control block overhead in the internal buffer.

On the other hand storing messages in external line format has some disadvantages:

- Messages cannot be checked.
- Single fields, for example field 20 of a SWIFT message, cannot be directly accessed via API calls.
- Display and printing of messages is in noprompt format only.
- Routing using message text fields is not possible in a standard way.

Conversion between the ELF and TOF formats can be done using standard API calls. The following description assumes that the message is in the internal queue buffer, for example after a GETN call. After conversion the message can be processed further, for example routed to target queues using the ROU call.

Conversion from tokenized format to external line format

1. Use MSGG to map the message from the internal buffer to external line format in an external buffer. The parameter INTMSGID should be set to blank; the parameter INTFRMID should be set to the required line format, for example 'W' in the case of SWIFT messages.
2. Use MSGP to map from the external buffer back into the internal buffer with INTMSGID = '0ELF' and INTFRMID = '0'.
3. Use WRIT to write the message type, right-padded to 8 characters together with the format ID, for example 'S100ƀƀƀƀW', to TOF field DSLTYPE at nesting identifier 1.
4. The internal buffer now contains the message in external line format.

Conversion from external line format to tokenized format

1. Use MSGG to map from the internal buffer to the external line format in an external buffer with INTMSGID = '0ELF' and INTFRMID = '0'.
2. Use READ to read the TOF field DSLTYPE at nesting identifier 1 and save the first 8 bytes of the result as *dsltype1* and the ninth character as *dsltype2*.
3. Use MSGP to map from external buffer back into the internal buffer. The field INTMSGID should be set to *dsltype1*, or left blank to force an automatic message type determination by MERVA ESA. The field INTFRMID should be set to *dsltype2* or to an appropriate line format identification, for example, use 'W' for SWIFT messages.
4. The internal buffer now contains the message in TOF format.

It is also possible to have both formats in the internal queue buffer at the same time. In this case it is the responsibility of the application to make sure that both message formats are synchronized, that is, contain the identical message.

Add the external line format to a tokenized format

1. Use MSGG to map the message from the internal buffer to external line format in an external buffer. The parameter INTMSGID should be set to blank; the parameter INTFRMID should be set to the required line format, for example 'W' in the case of SWIFT messages.
2. Use WRIT to write the data in external line format to the TOF field DSLELF at nesting identifier 1.
3. Use WRIT to write the message type, right-padded to 8 characters together with the format ID, for example 'S100ƀƀƀƀW', to the TOF field DSLTYPE at nesting identifier 1.
4. The internal buffer now contains the message in TOF format and in external line format.

Add the tokenized format to a message in external line format

1. Use MSGG to map from the internal buffer to the external line format with INTMSGID = ‘0ELF’ and INTFRMID = ‘0’.
2. Use READ to read the TOF field DSLTYPE at nesting identifier 1 and save the first 8 bytes of the result as *dsltype1* and the ninth character as *dsltype2*.
3. Use MSGP to map back from the external line format into the internal buffer. The field INTMSGID should be set to *dsltype1* or left blank to force an automatic message type determination by MERVA ESA. The field INTFRMID should be set to *dsltype2* or to an appropriate line format identification, for example, use ‘W’ for SWIFT messages.
4. Use WRIT to write the data in external line format to the TOF field DSLELF at nesting identifier 1.
5. Use WRIT to write the message type, right-padded to 8 characters together with the format ID, for example ‘S100ƀƀƀƀW’, to the TOF field DSLTYPE at nesting identifier 1.
6. The internal buffer now contains the message in TOF format and in external line format.

Queue Management Using DB2

Whether MERVA ESA runs with queue management using VSAM or with queue management using DB2 is completely transparent to MERVA ESA application programs.

Normally all MERVA ESA queue management requests are handled by the central MERVA ESA queue management. When you run MERVA ESA under MVS with queue management using DB2, your API programs can specify that their queue management requests should be executed directly. To do so, they must switch on the API customization flag APICQDIR before the very first queue management request (see “APICQDIR” on page 52). DSLAPI then initializes and terminates the queue management itself without going through DSLNUC.

Additionally your applications can then also specify that they want to control the commit and rollback themselves (see “APICQMIT” on page 53).

Chapter 7. Auxiliary API Services

This chapter describes additional MERVA ESA services that help you to process SWIFT messages in your applications:

- SWIFT Field services

  These services provide a simple interface for reading and writing fields in SWIFT messages.
- SWIFT message conversion

  This service provides a simple interface for converting SWIFT messages between the SWIFT I and SWIFT II formats.
- EDIFACT message conversion

  This service allows you to build EDIFACT messages from SWIFT messages and to extract SWIFT messages from EDIFACT messages.

  However, you are recommended to use the standard conversion programsprovided by MERVA ESA, the batch programs DSLSDI and DSLSDO, and the transactions DSLCESTR and DSLCSETR. These are discussed in the chapter on Message Processing in *MERVA for ESA Concepts and Components*.

Field Services

As SWIFT-type messages have variable-length lines (fields), MERVA ESAprovides two additional programs that convert variable-length fields to fixed-length fields. You can then read and write fixed-length fields from your application program.

The programs DSLAPFFS and DSLAPFTS enable you to read and write a SWIFT-type message line by line.

- DSLAPFFS read fields from SWIFT
- DSLAPFTS write fields to SWIFT.

The Field Services working storage structure, FLDWSTOR, is defined by the copybook DSLAPFWS, and described in the following section.

The Field Services Working Storage FLDWSTOR

Table 1 shows the structure of the field services working storage (FLDWSTOR).

Table 1. Structure of the Field Working Storage

Label	Length (Bytes)	Description
FLDFUNC	4	Function: INIT INIB DATA DAII
FLDRC	2	Return code: space=No error 01=Function 02=Buffer size (DSLAPFTS MSGSMSG) (DSLAPFFS FLDSFLD) 09=End (DSLAPFFS)
FLDSFLD	1016	Field buffer
	2	
	72	Save area

You can copy the field working storage definition to yourapplication program using the copybook DSLAPFWS.

DSLAPFFS Read Fields from a SWIFT-Type Message

This program provides two functions:

- An initialization function (INIT), which positions an offset pointer to the first field or message line
- A data function (DATA), which retrieves the next field or message line.

The Initialization Function INIT

The INIT function of DSLAPFFS positions an offset pointer to the first field or message line in the message buffer MSGSWIFT.

The offset pointer is saved in the MSGSWIFT field MSGSMSG+2 and used by the DATA function.

```
►►──INIT──(──FLDWSTOR──,──MSGSWIFT──)──────────────────────────────────►◄
```

Parameters:

FLDWSTOR
The Field Services working storage. The FLDWSTOR structure is defined by the copybook DSLAPFWS.

MSGSWIFT
The variable length buffer containing the SWIFT message. The buffer format is defined by the MSGSWIFT structure, copybook DSLAPIMS.

Return Codes: The INIT function has no return codes.

The Data Retrieval Function DATA

The DATA function of DSLAPFFS retrieves the next field or message line from MSGSWIFT and puts it in the FLDWSTOR variable FLDSFLD. The function adds blanks at the end of the field or message line to fill FLDSFLD and increments the offset pointer to the next field or message line.

```
►►──DATA──(──FLDWSTOR──,──MSGSWIFT──)──────────────────────────────────►◄
```

Parameters:

FLDWSTOR
The Field Services working storage. The FLDWSTOR structure is defined by the copybook DSLAPFWS.

MSGSWIFT
The variable length buffer containing the SWIFT message. The buffer format is defined by the MSGSWIFT structure, copybook DSLAPIMS.

Usage Notes:

1. The maximum field length is 1016 bytes.
2. The end of a field or message line is indicated by a carriage return/line feed (CRLF) character sequence, except lines that end with '-' in SWIFT I or '}' in SWIFT II.
3. The field separator CRLF is removed from the end of the field or message line.

Return Codes:

FLDRC = space
The function retrieved the next field or message line successfully.

FLDRC = 01
Invalid function. Function is not INIT or DATA.

FLDRC = 02
The function cannot transfer the next field or line to FLDSFLD because it is longer than 1016 bytes.

FLDRC = 09
The function has found the end of the message.

DSLAPFTS Write Fields to a SWIFT-Type Message

This program provides two functions:

- An initialization function (INIT), which initializes the message buffer for a fixed length
- An initialization function (INIB), which initializes the message buffer for a customizable length
- A data function (DATA), which adds the field FLDSFLD and a CR/LF field separator to the message buffer
- A data function (DAII), which adds the field FLDSFLD, without a field separator, to the message buffer.

The Initialization Function INIT

The INIT function of DSLAPFTS fills MSGSWIFT with blanks and sets the length fields to the default maximum message length (4088 bytes). The length fields can be overwritten afterwards by using the auxiliary buffer prefix manipulation routines.

```
►►──INIT──(──FLDWSTOR──,──MSGSWIFT──)──────────────────────────────────────►◄
```

Parameters:

FLDWSTOR
The Field Services working storage. The FLDWSTOR structure is defined by the copybook DSLAPFWS.

MSGSWIFT
The variable length buffer containing the SWIFT message. The buffer format is defined by the MSGSWIFT structure, copybook DSLAPIMS.

Usage Notes: You can change the maximum message length.

Return Codes: The INIT function has no return codes.

The Initialization Function INIB

The INIB function of DSLAPFTS fills MSGSWIFT with blanks and sets the length fields to the values defined in DSLPARM (APISMSG parameter).

```
►►──INIB──(──FLDWSTOR──,──MSGSWIFT──,──INTWSTOR──)─────────────────────────►◄
```

Parameters:

FLDWSTOR
The Field Services working storage. The FLDWSTOR structure is defined by the copybook DSLAPFWS.

MSGSWIFT
The variable length buffer containing the SWIFT message. The buffer format is defined by the MSGSWIFT structure, copybook DSLAPIMS.

INTWSTOR
The interface working storage. The interface working storage is defined by the INTWSTOR structure, copybook DSLAPIWS. Prior to using the INTWSTOR working storage, it must be initialized by calling the DSLAPI with the INIT function.

Usage Notes: You can change the maximum message length.

Return Codes: The INIB function has no return codes.

The Data Insertion Function DATA

The DATA function of DSLAPFTS adds the field FLDSFLD to MSGSWIFT and updates the length fields.

```
►►──DATA──(──FLDWSTOR──,──MSGSWIFT──)──────────────────────────────────────►◄
```

Parameters:

FLDWSTOR
The Field Services working storage. The FLDWSTOR structure is defined by the copybook DSLAPFWS.

MSGSWIFT
The variable length buffer containing the SWIFT message. The buffer format is defined by the MSGSWIFT structure, copybook DSLAPIMS.

Usage Notes: The function removes any fill characters (blanks) from the end of the field. It then adds a field separator CRLF to the end of the field data except:
- When the first and only character is '-' indicating a SWIFT I trailer.
- When the last character is '}' indicating SWIFT II.

Return Codes:

FLDRC = space
The function has added the field or message line successfully.

FLDRC = 01
Invalid function. Function is not INIT, INIB, DATA, or DATI.

FLDRC = 02
There is insufficient free space available in MSGSWIFT for the function to transfer the data stored in FLDSFLD.

The Data Insertion Function DAII

The DAII function of DSLAPFTS adds the field FLDSFLD to MSGSWIFT and updates the length fields.

Note: The function removes any fill characters (blanks) from the end of the field. Separators are not added.

```
►►──DAII──(──FLDWSTOR──,──MSGSWIFT──)────────────────────────────────►◄
```

Parameters:

FLDWSTOR
The Field Services working storage. The FLDWSTOR structure is defined by the copybook DSLAPFWS.

MSGSWIFT
The variable length buffer containing the SWIFT message. The buffer format is defined by the MSGSWIFT structure, copybook DSLAPIMS.

Return Codes:

FLDRC = space
The function has added the field or message line successfully.

FLDRC = 01
Invalid function. Function is not INIT, INIB, DATA, or DATI.

FLDRC = 02
There is insufficient free space available in MSGSWIFT for the function to transfer the data stored in FLDSFLD.

SWIFT Message Conversion Services

MERVA ESA provides two programs that you can call from your application program to map messages between SWIFT I and SWIFT II formats:

- DSLAP1T2 to map SWIFT I banking messages to SWIFT II format
- DSLAP2T1 to map SWIFT II banking messages to SWIFT I format.

Working Storage Areas for the Message Services

Each of the message mapping programs requires a working storage area. The structures are identical but the variable names are different:

- M12WSTOR for the DSLAP1T2 program. The structure is defined by the copybook DSLAP1WS. Table 2 shows the structure of M12WSTOR.

Table 2. Structure of the M12WSTOR Working Storage

Label	Length (Bytes)	Description
M12RC	2	Return code: space=No error 01=Formal error (Message type not supported) 02=System error (Undefined message, DSLAPFFS, DSLAPFTS)
M12ERMSG	131	Error Message
	891	DSLAP1T2 working storage

- M21WSTOR for the DSLAP2T1 program. The structure is defined by the copybook DSLAP2WS. See Table 3.

Table 3. Structure of the M21WSTOR Working Storage

Label	Length (Bytes)	Description
M21RC	2	Return code: space=No error 01=Formal error (Message type not supported) 02=System error (Undefined message, DSLAPFFS, DSLAPFTS)
M21ERMSG	131	Error message
	891	DSLAP2T1 working storage

How to Call the DSLAP1T2 Program

The program DSLAP1T2 maps a SWIFT banking message from SWIFT II format to SWIFT I format:

```
►►──DSLAP1T2──(──M12WSTOR──,──MS1SWIFT──,──MS2SWIFT──)──────────────────────►◄
```

Parameters

M12WSTOR
: The Message Conversion Services working storage. This data structure is defined by the copybook DSLAP1WS.

MS1SWIFT
: The variable length buffer containing the message in SWIFT I format. The buffer format is defined by the MSGSWIFT structure, copybook DSLAPIMS.

MS2SWIFT
: The variable length buffer containing the message in SWIFT II format. The buffer format is defined by the MSGSWIFT structure, copybook DSLAPIMS.

Return Codes

M12RC = space
: The function mapped the message successfully.

M12RC = 01
: The function cannot map the message. Additional information is contained in field M12ERMSG:

- SYSTEM MESSAGE -> MIGRATION NOT SUPPORTED
- MESSAGE WITH MULTIPLE DEST. -> NOT SUPPORTED BY SWIFT-II.

M12RC = 02
: The function cannot map the message. Additional information is contained in field M12ERMSG:

- NO SWIFT-I MESSAGE - NO MIGRATION POSSIBLE
- UNFORMATTED SWIFT-I MESSAGE - NO MIGRATION POSSIBLE.

How to Call the DSLAP2T1 Program

The program DSLAP2T1 maps the following types of SWIFT message from SWIFT II format to SWIFT I format:

- SWIFT Banking Messages
- Non-Delivery Warning (010)
- Delivery Notification (011).

The following trailer components are supported in the conversion process:

- PDE, possible duplicate emission
- MAC, authentication.

All other trailers are ignored.

```
►►──DSLAP2T1──(──M21WSTOR──,──MS1SWIFT──,──MS2SWIFT──)──────────────────────►◄
```

Parameters

M21WSTOR
: The Message Conversion Services working storage. This data structure is defined by the copybook DSLAP2WS.

MS1SWIFT
The variable length buffer containing the message in SWIFT I format. The buffer format is defined by the MSGSWIFT structure, copybook DSLAPIMS.

MS2SWIFT
The variable length buffer containing the message in SWIFT II format. The buffer format is defined by the MSGSWIFT structure, copybook DSLAPIMS.

Return Codes

M21RC = space
The function has mapped the message successfully.

M21RC = 01
The function cannot map the message. Additional information is contained in field M21ERMSG:

- NO FINANCIAL MESSAGE - NO MIGRATION POSSIBLE
- NO FINANCIAL/BANKING MESSAGE - NO MIGRATION POSSIBLE.

M21RC = 02
The function cannot map the message. Additional information is contained in field M21ERMSG:

- NO SWIFT-II MESSAGE - NO MIGRATION POSSIBLE
- UNFORMATTED SWIFT-II MESSAGE - NO MIGRATION POSSIBLE.

EDIFACT Message Conversion Services

MERVA ESA provides three programs that you can call from your application program to enable you to map messages between EDIFACT and the SWIFT message types 105 and 106.

Note: Instead of writing your own program to do this conversion, however, you are recommended to use the conversion programs provided by MERVA ESA. These are the programs DSLCES1, DSLCES2, and DSLCSE1, and the transactions DSLCESTR and DSLCSETR.

These programs, and the concepts of EDIFACT message conversion, are discussed in the chapter on Message Processing in *MERVA for ESA Concepts and Components*.

The three programs are:

- DSLCES1 converts an EDIFACT message into multiple SWIFT messages, and puts all these messages into a MERVA ESA queue.

  DSLCES1 has built in RESTART recovery, see Figure 3 on page 69.
- DSLCES2 maps an EDIFACT message from an input buffer and converts into multiple SWIFT messages and puts one of these messages into the internal queue buffer.
- DSLCSE1 gets a group of SWIFT messages from a MERVA ESA queue, converts them into an EDIFACT message and puts it into an output buffer.

The Conversion Program DSLCES1

DSLCES1 converts an EDIFACT message into multiple SWIFT messages, and puts all these messages into an intermediate queue, and routes them according to the routing table of the intermediate queue. DSLCES1 has built in RESTART recovery; this is described in Figure 3 on page 69.

There are two functions you can use:

PUT The PUT function:

1. Maps the EDIFACT message into SWIFT 105 or 106 messages
2. Writes all these messages and a RESTART message to the intermediate queue
3. Routes all the SWIFT messages to the target queues.

Note: After each SWIFT message has been built, but before it is put into the intermediate queue, the user exit DSLMU240 is called to let the user inspect or modify the SWIFT message.

DELR Deletes the RESTART message from the intermediate queue.

Figure 2 shows how to call the DSLAPI EDIFACT conversion service program DSLCES1.

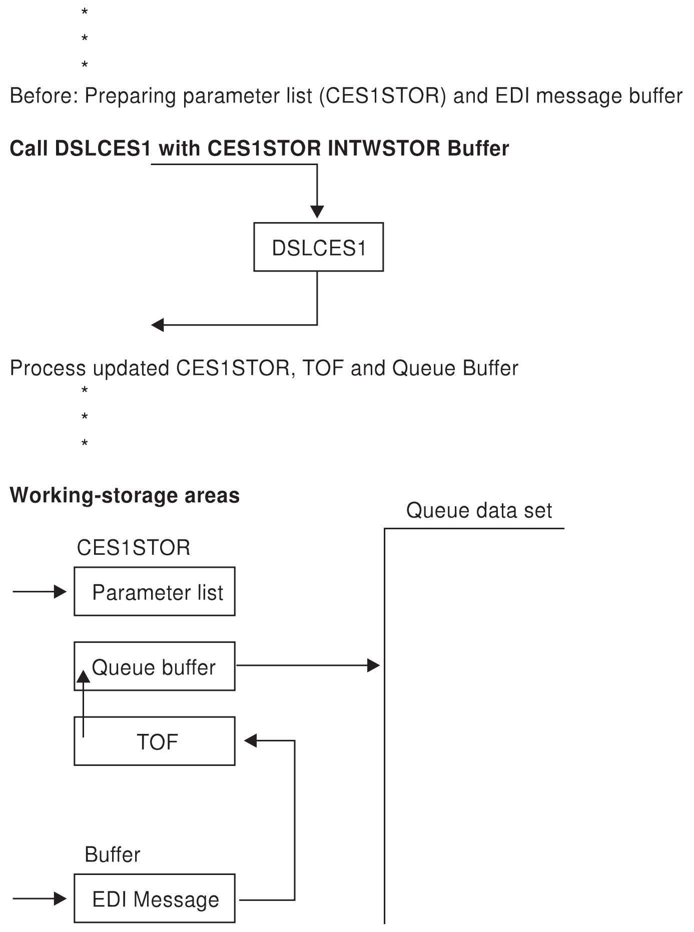

*
*
*
Before: Preparing parameter list (CES1STOR) and EDI message buffer
Call DSLCES1 with CES1STOR INTWSTOR Buffer
DSLCES1
Process updated CES1STOR, TOF and Queue Buffer
*
*
*
Working-storage areas
Queue data set
CES1STOR
Parameter list
Queue buffer
TOF
Buffer
EDI Message

Figure 2. Calling the DSLCES1 Program

Note: For EDIFACT messages in queue format (CES1IF='Q') the EDIFACT message is passed to DSLCES1 in the internal queue of DSLAPI. In this case no input buffer is required.

For EDIFACT messages in net format (CES1IF='N') the EDIFACT message is passed to DSLCES1 in the input buffer.

The Conversion Working Storage CES1STOR

Table 4 shows the structure of the parameter list of the CES1STOR working storage. The structure is defined in copybook DSLCES1S (Assembler), DSLAPCBL (COBOL), DSLAPPLI (PL/I), and DSLAPC (C).

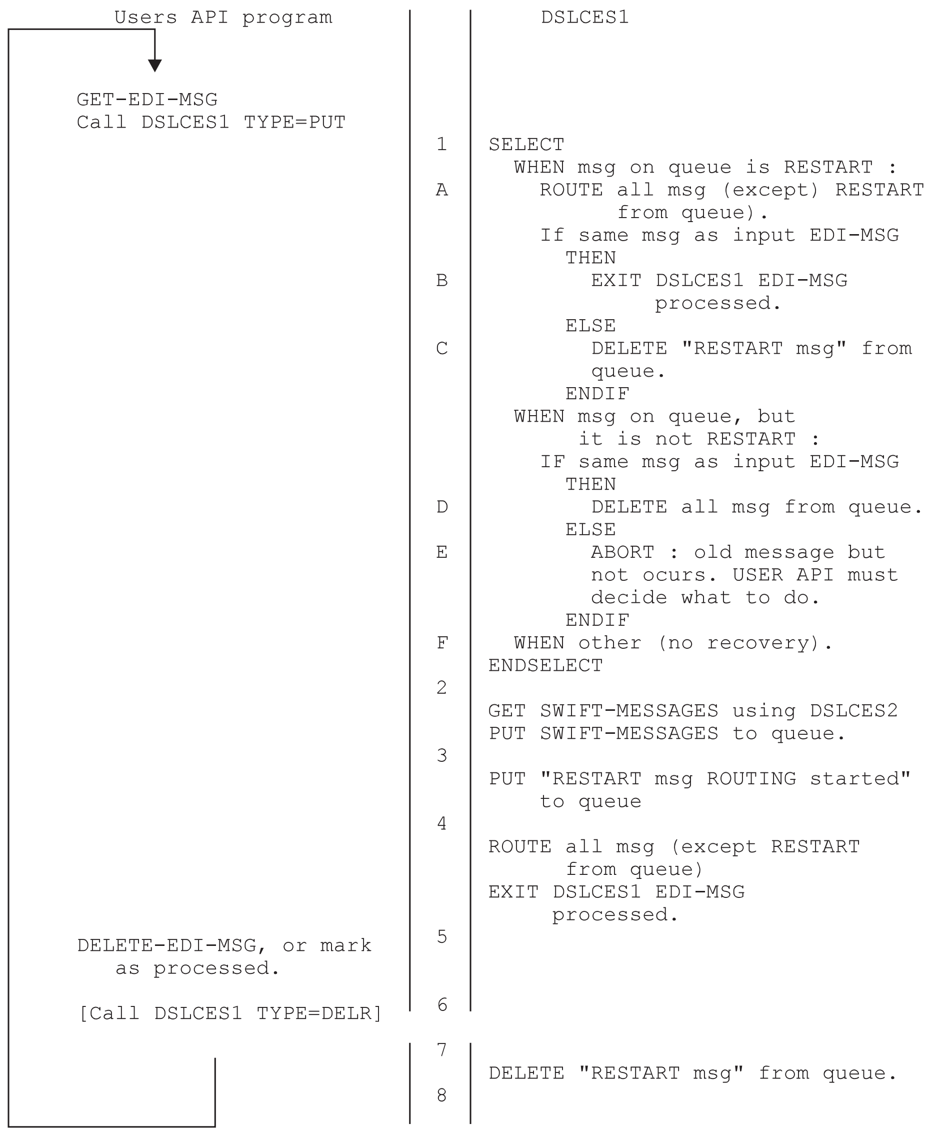
Table 4. Structure of the DSLCES1 Working Storage

Label	Length (Bytes)	Type	Description
			INPUT FIELDS
CES1FUNC	4	C	Function: PUT = Convert EDIFACT message to SWIFT messages, PUT SWIFT messages to queue and ROUTE them. DELR = Delete RESTART message from queue.
CES1IF	1	C	Input Format Q (queue) or N (net)
CES1AORD	8	C	ACCEPT or DROP messages with non-critical errors
CES1Q	8	C	Intermediate Queue
	3		
			OUTPUT FIELDS
CES1RC	2	C	Return Code: =Okay (Normal Completion (spaces)) 01=Formal Error 02=Processing Error
CES1EMSG	131	C	Error Message
	3		
CES1NMSG	2	B	Number of SWIFT Messages created (1-9)
CES1CEMS	131	C	Checking message. Checking is made if the intermediate queue has CHECK=YES
CES1HOME	11	C	Returned Home ID
CES1MT	3	C	Returned Message Type of SWIFT
CES1CORR	11	C	Message (105/106)
CES1PRI	1	C	Returned Correspondents ID
CES1SW20	16	C	Returned Priority
CES1SW21	16	C	Returned Transaction Reference
CES1SW12	3	C	Returned Related Reference
			Returned Message Number
CES1WORK	3742	C	Working storage reserved for DSLCES1

B = Binary value
C = Character			

Restart Recovery of DSLCES1

Figure 3 on page 69 shows how the API EDIFACT conversion service program DSLCES1 handles RESTART recovery.



```
     Users API program                     DSLCES1

  GET-EDI-MSG
  Call DSLCES1 TYPE=PUT
                                 1   SELECT
                                       WHEN msg on queue is RESTART :
                                 A       ROUTE all msg (except) RESTART
                                               from queue).
                                         If same msg as input EDI-MSG
                                           THEN
                                 B           EXIT DSLCES1 EDI-MSG
                                                  processed.
                                           ELSE
                                 C           DELETE "RESTART msg" from
                                             queue.
                                           ENDIF
                                       WHEN msg on queue, but
                                            it is not RESTART :
                                         IF same msg as input EDI-MSG
                                           THEN
                                 D           DELETE all msg from queue.
                                           ELSE
                                 E           ABORT : old message but
                                             not ocurs. USER API must
                                             decide what to do.
                                           ENDIF
                                 F     WHEN other (no recovery).
                                     ENDSELECT
                                 2
                                     GET SWIFT-MESSAGES using DSLCES2
                                     PUT SWIFT-MESSAGES to queue.
                                 3
                                     PUT "RESTART msg ROUTING started"
                                         to queue
                                 4
                                     ROUTE all msg (except RESTART
                                           from queue)
                                     EXIT DSLCES1 EDI-MSG
                                          processed.
                                 5
  DELETE-EDI-MSG, or mark
     as processed.
                                 6
  [Call DSLCES1 TYPE=DELR]
                                 7
                                     DELETE "RESTART msg" from queue.
                                 8
```

Figure 3. Restart Recovery of DSLCES1

Previous job crashes in step	Recovery made in step
Before step 1	No recovery is made, processing had not been started
Between steps 1-2	All ready in a recovery, so the recovery type will depend on the crash, before this one
Between steps 2-3	Left with SWIFT messages in intermediate queue. Recovery will be made by step D.
Between steps 4-5	Started routing message to target queues. Recovery will be made by step a. Followed by step B.
Between steps 6-7	Started routing message to target queues. Recovery will be made by step A. Followed by step C.
After step 8	No recovery is made, processing has completed.

Figure 4. Restart Recovery Explanation

Note: In the above recovery explanation **E** shows the program DSLCES1 failing because there are messages in the intermediate queue but these are not the same as the message being converted. One solution would be for the users API program to delete all the messages from the intermediate queue.

Calling the Program DSLCES1

The following example shows how to call DSLCES1:

```
CES1FUNC := 'PUT '
CES1IF   := 'N'
CES1AORD := 'ACCEPT  '
CES1Q    := 'L1CES1  '
 ...
Call DSLCES1 with CES1STOR INTWSTOR BUFFER
 ...
CES1RC    = space|01|02
```

The return codes (CES1RC) show:

CES1RC = space
The call is successful.

CES1RC = 01 The call has failed, with a formal error. Additional information is contained in fields CES1EMSG.

CES1RC = 02 The call has failed, an error was detected during conversion. Additional information is contained in fields CES1EMSG.

The Conversion Program DSLCES2

There are two functions you can use when mapping EDIFACT messages to SWIFT messages:

PUTF The PUTF function:

1. Maps the EDIFACT message into a SWIFT 105 or 106 messages
2. Extracts the first section of EDIFACT data
3. Puts it into field SW77.

The SWIFT message is placed into the internal queue.

Note: The user exit DSLMU242 is called to extract the SWIFT fields required in MT 105/106 from the EDIFACT message.

PUTN The PUTN function:

1. Extracts the next section of EDIFACT data
2. Replaces the field SW77 in the internal queue with the new data
3. Increments the field SW27 in the internal queue.

Figure 5 shows how to call the DSLAPI EDIFACT conversion service program DSLCES2.

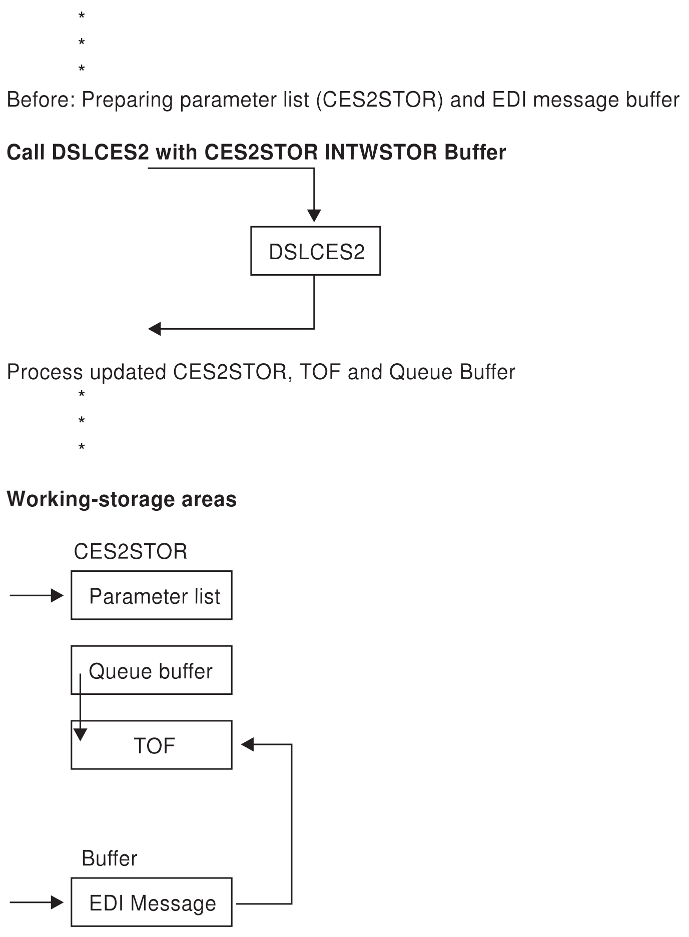

*
*
*
Before: Preparing parameter list (CES2STOR) and EDI message buffer
Call DSLCES2 with CES2STOR INTWSTOR Buffer
DSLCES2
Process updated CES2STOR, TOF and Queue Buffer
*
*
*
Working-storage areas
CES2STOR
Parameter list
Queue buffer
TOF
Buffer
EDI Message

Figure 5. Calling the DSLCES2 Program

Note: Unlike DSLCES1 and DSLCSE1 the EDIFACT message is passed to DSLCES2 in the input buffer, even when it is in queue format (CES2IF='Q'), this is because the SWIFT messages are passed back in the internal queue buffer.

The Conversion Working Storage CES2STOR

Table 5 on page 72 shows the structure of the parameter list of the CES2STOR working storage. The structure is defined in copybook DSLCES2S (Assembler), DSLAPCBL (COBOL), DSLAPPLI (PL/I), and DSLAPC (C).

DSLCES2

Table 5. Structure of the DSLCES2 Working Storage

Label	Length (Bytes)	Type	Description
			INPUT FIELDS
CES2FUNC	4	C	Function: PUTF = Put the first message in EDIFACT buffer to the internal queue. PUTN = Put the next message in EDIFACT buffer to the internal queue.
CES2IF	1	C	Input Format Q (queue) or N (net)
CES2CK	1	C	Checking: Y=Message checking, N=No Message checking
	2		
			OUTPUT FIELDS
CES2RC	2	C	Return Code: =Okay (Normal Completion (spaces)) 00=Okay (Messages to follow) 01=Formal Error 02=Processing Error
CES2EMSG	131	C	Error Message
	3		
CES2LMSG	2	B	Last Message Processed
CES2NMSG	2	B	Number of SWIFT Messages created (1-9)
CES2CEMR	2	B	EDIFACT MSG : MFS Reason Code (CES2CK=Y only)
CES2CEMS	131	C	EDIFACT MSG : Error Message (CES2CK=Y only)
	1		
CES2CSMR	2	B	SWIFT MSG : MFS Reason Code (CES2CK=Y only)
CES2CSMS	131	C	SWIFT MSG : Error Message (CES2CK=Y only)
CES2HOME	11	C	Returned Home ID
CES2MT	3	C	Returned Message Type of SWIFT Message (105/106)
CES2CORR	11	C	Returned Correspondents ID
CES2PRI	1	C	Returned Priority
CES2SW20	16	C	Returned Transaction Reference
CES2SW21	16	C	Returned Related Reference
CES2SW12	3	C	Returned Message Number
CES2WORK	1572	C	Working storage reserved for DSLCES2

B = Binary value
C = Character			

The Put EDIFACT Message Function PUTF

The PUTF function does the following:

1. Extracts the SWIFT fields from the EDIFACTmessage using the user exit of DSLCES2.
2. Formats the required SWIFTmessage (105/106) in the internal TOF.
3. Adds the extracted SWIFT fields to the internal TOF.
4. Takes the first section of the EDIFACT data and places it in field SW77.
5. The internal TOF is passed through to the internal queue.
6. If checking is requested (CES2CK=Y), the whole EDIFACT message is checked and the result of this checking is placed in CES2CEMR and CES2CEMS, the SWIFTmessage produced is checked and the result of this checking is placed in CES2CSMR and CES2CSMS.

The Put EDIFACT Message Function PUTN

The PUTN function does the following:

1. Takes the next section of EDIFACT data.
2. Replaces the field SW77, in the internal queue, with the new data.

3. Increments the field SW27 in the internal queue.
4. If checking is requested (CES2CK=Y), the SWIFT message produced is checked and the result of this checking is placed in CES2CSMR and CES2CSMS. The whole EDIFACT message is not checked and CES2CEMS and CES2CEMR are left unchanged.

Note: The PUTN function relies on the fact that the internal queue still contains the SWIFT message produced by the previous call to DSLCES2.

Calling the Program DSLCES2

The following example shows how to call DSLCES2:

```
CES2FUNC := 'PUTF'
CES2IF   := 'N'
CES2CK   := 'Y'
 ...
Call DSLCES2 with CES2STOR INTWSTOR Buffer
 ...
CES2RC    = space|00|01|02
```

The return codes (CES2RC) show:

CES2RC = space
The call is successful, and there are no more SWIFT messages.

CES2RC = 00
The call is successful, but there are more SWIFT messages.

CES2RC = 01
The call has failed, with a formal error. Additional information is contained in fields CES2EMSG.

CES2RC = 02
The call has failed, an error was detected during conversion. Additional information is contained in fields CES2EMSG.

The Conversion Program DSLCSE1

DSLCSE1 gets a group of SWIFT messages from a MERVA ESA queue, converts them into an EDIFACT message, and this message is put into an output EDIFACTbuffer. The program DSLCSE1 is based around an audit list, which is both built and used by DSLCSE1. The audit list describes the status of a group of SWIFT messages making up an EDIFACT message. Table 6 shows the structure of the audit list used by DSLCSE1. The structure is defined in copybook DSLCSE1A (Assembler), DSLAPCBL (COBOL), DSLAPPLI (PL/I), and DSLAPC (C).

Table 6. Structure of the Audit List

Label	Length (Bytes)	Type	Description
AUDITBH			Audit Buffer Header
AUDITBL	2	B	Maximum Buffer Length
	2		
AUDITDL	2	B	Data Length plus 4
	2		
			Audit Record Header Data

Table 6. Structure of the Audit List (continued)

Label	Length (Bytes)	Type	Description
AUDDATE	8	C	Date (YY:MM:DD) the audit list was produced/updated
AUDTIME	8	C	Time (HH:MM:SS) the audit list was produced/updated
AUDUSER	16	C	Reserved for the user. For example, a comment
AUDSTAT	1	C	Audit list status C (Complete) S (Skeleton)
AUDFUNC	4	C	The function (CSE1FUNC) causing the audit list production/update
AUDQUEUE	8	C	The queue the audit list was produced for
AUDHOME	11	C	The LT of the receiver (Home ID)
AUDCORR	11	C	The LT of the sender (Corr ID)
AUDSW21	16	C	Related Reference of the EDIFACT message
	1	C	Reserved
AUDNMSG	2	B	Number of SWIFT messages in the EDIFACT message
	2	C	Reserved
			Audit Record one per SWIFT message
AUDQSN	4	B	Queue sequence number
AUDMNUM	2	B	Message number in the sequence
			The following fields are blank in a skeleton audit list (GET only)
AUDOSS	4	C	Output Session Number
AUDOSN	6	C	Output Sequence Number
AUDODATE	6	C	Output Date from SWIFT (YYMMDD)
AUDOTIME	4	C	Output Time from SWIFT (HHMM)
AUDMSTAT	1	C	SWIFT Message Usage Status. U (used for EDIFACT) D (duplicate) (not decided yet (space))
AUDPSTAT	1	C	PUTB, ROUB and DELE Status. S (move, route, delete successful) F (move, route, delete failed) (move, route, delete not attempted (space))
	4	C	Reserved

B = Binary value
C = Character			

The functions provided for SWIFT to EDIFACT conversion are:

GET Builds a skeleton audit list for an EDIFACT message. The EDIFACT message that the audit list is built for is identified by putting one part of the EDIFACT message (that is, one of the SWIFT messages) into the DSLAPI internal queue before the DSLCSE1 TYPE=GET is made.

If all SWIFT parts of the EDIFACT message are present in the MERVA ESA queue, the SWIFT messages are converted into the EDIFACT message, this message is put into an output EDIFACT buffer, and the audit list is completed.

Notes:

1. The user exit DSLMU241 is called after each SWIFT message is read from the queue to allow for inspection and modification.
2. If all SWIFT parts of the EDIFACT message are not present in the MERVA ESA queue, only a skeleton audit list will be built, and DSLCSE1 will not read any messages from the queue.

If all SWIFT parts of the EDIFACT message are present in the MERVA ESA queue, but the conversion fails while building the EDIFACT message, DSLCSE1 will fill in the non-skeleton data for only the SWIFT message already read.

LIST Builds the audit list for an EDIFACT message. The EDIFACT message that the audit list is built for is identified by putting one part of the EDIFACT message (that is, one of the SWIFT messages) into the DSLAPI internal queue before the DSLCSE1 TYPE=LIST is made.

Note: To build the audit list all the SWIFT messages making up the EDIFACTmessage will be read from the queue by DSLCSE1.

PUTB The PUTB function moves all the SWIFT messages in the audit list using a back reference, that is, the SWIFT messages are moved and deleted using the API automatic delete mechanism.

ROUB The ROUB function routes all the SWIFT messages in the audit list using a back reference, that is, the SWIFT messages are routed and deleted using the API automatic delete mechanism.

DELE The DELE function deletes all the SWIFT messages in the audit list.

Notes:

1. The functions PUTB, ROUB, and DELE will not operate on a skeleton audit list. If you want to use these functions after an unsuccessful GET, you must first perform the LIST function.
2. The contents of the API internal queue buffer will be changed by a call to DSLCSE1.

Figure 6 on page 76 shows how to call the DSLAPI EDIFACT conversion service program DSLCSE1.

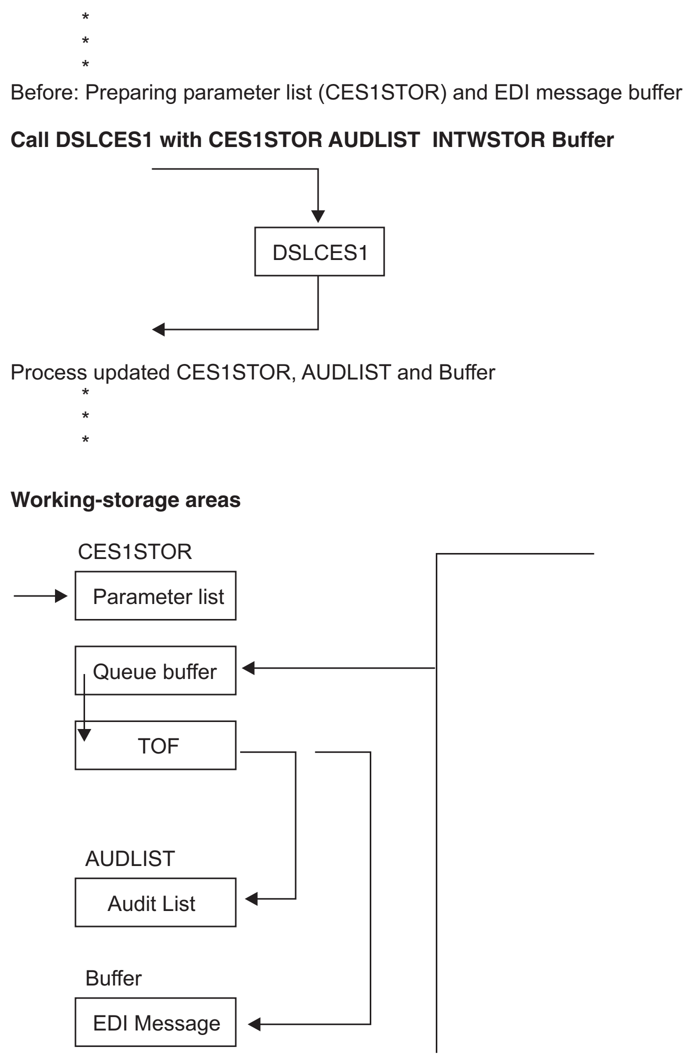

*
*
*
Before: Preparing parameter list (CES1STOR) and EDI message buffer
Call DSLCES1 with CES1STOR AUDLIST INTWSTOR Buffer
DSLCES1
Process updated CES1STOR, AUDLIST and Buffer
*
*
*
Working-storage areas
CES1STOR
Parameter list
Queue buffer
TOF
AUDLIST
Audit List
Buffer
EDI Message

Figure 6. Calling the DSLCSE1 Program

Note: For EDIFACT messages returned in queue format (CSE1FORM='Q') the EDIFACT message is returned from DSLCSE1 in the internal queue of DSLAPI. In this case no output buffer is required.

For EDIFACT messages returned in net format (CSE1FORM='N') the EDIFACT message is returned from DSLCSE1 in the output buffer.

The Conversion Working Storage CSE1STOR

Table 7 on page 77 shows the structure of the parameter list of the CSE1STOR working storage. The structure is defined in copybook DSLCSE1S (Assembler), DSLAPCBL (COBOL), DSLAPPLI (PL/I), and DSLAPC (C).

Table 7. Structure of the DSLCSE1 Working Storage

Label	Length (Bytes)	Type	Description
			INPUT FIELDS

CSE1FUNC	4	C	`Function: GET  = Convert SWIFT messages to an EDIFACT message, build an Audit List.`
			`LIST = Build an Audit List.`
			`PUTB = Put and delete SWIFT messages in Audit list.`
			`ROUB = Route and delete SWIFT messages in Audit list.`
			`DELE = Delete SWIFT messages in Audit list.`

CSE1IQ	8	C	Input Queue
CSE1FORM	1	C	Format of EDIFACT message produced. Q or N.
CSE1MID	8	C	Message Identification
CSE1LF	1	C	Line Format
CSE1PART	1	C	Parts deleted/routed/moved (DELE/PUTB/ROUB only) A (all), D (Duplicate) or U (used in EDIFACT)
CSE1OQ	8	C	Output Queue (PUTB only).
CSE1TCOD	4	C	Trace code to be added to MSGTRACE (PUTB, ROUB only) For PUTB no trace record is written if CSE1OQ=CSE1IQ
	1	C	Reserved
			OUTPUT FIELDS

CSE1RC	2	C	`Return Code:  =Okay (Normal Completion (spaces))`
			`01=Formal Error`
			`02=Processing Error`

CSE1EMSG	131	C	Error Message
CSE1WORK	1879	C	Working storage reserved for DSLCSE1
C = Character			

Calling the Program DSLCSE1

The following example shows how to call DSLCSE1:

```
CSE1FUNC := 'GET '
CSE1MID  := 'EDIFACT '
CSE1LF   := 'Z'
CSE1Q    := 'D1CSE1  '
CSE1FORM := 'N'
 ...
Call DSLCSE1 with CSE1STOR AUDLIST INTWSTOR Buffer
 ...
CSE1RC    = space|01|02
```

The return codes (CSE1RC) show:

CSE1RC = space
The call is successful.

CSE1RC = 01 The call has failed, with a formal error. Additional information is contained in field CSE1EMSG.

CSE1RC = 02 The call has failed, an error was detected during conversion. Additional information is contained in fields CSE1EMSG.

Part 2. DSLAPI Data Structures and Functions

This part is a reference section that describes:

- The data structures used by DSLAPI
- Each DSLAPI function that can be called from an application program.

Chapter 8. Data Structures

This chapter describes the data structures used by API services.

In each of the supported languages except for C and REXX the structure is defined by the specified copybook or include-file. For C all API structures are defined in the C header file DSLAPC. REXX does not require data declaration.

Structure	Page	Copybook
The interface working storage area, INTWSTOR	81	DSLAPIWS
The TOF access parameters, TOFPARM	85	DSLAPITP
The MFS message buffer, MSGSWIFT	88	DSLAPIMS
The MFS message prefix, MSGSWIFT_PREFIX	88	DSLAPIMP
SWIFT message headers	90	DSLAPIMH
The journal key, JRNKEY	92	DSLAPIJK
The terminal user control block (TUCB) (Assembler: macro DSLMFS MF=TUCB)	93	DSLAPITU
The User File record, DSLUSRS (Assembler: macro DSLUSR)	94	DSLAPIUS
The MFS parameter list, MFSL (Assembler: macro DSLMFS MF=L)	95	DSLMFSPL

Refer to the description of the individual API services for details on which fields you use and how.

The copybooks are distributed in a separate library for each language as follows:

Assembler	MERVA.SDSLMAC0
COBOL	MERVA.SDSLMAC1
PL/I	MERVA.SDSLMAC2
C/370	MERVA.SDSLMAC3

Interface Working Storage INTWSTOR

The interface working storage, INTWSTOR, is 4096 bytes long and consists of two parts:

- The parameters that DSLAPI uses to communicate with the calling program. The parameter list is 512 bytes long.
- Working storage used by DSLAPI internally. This area is 3584 bytes long.

The interface working storage is provided by the application program. DSLAPI allocates other buffers separately, for example, the internal TOF and queue buffers. During processing their size may change.

Table 8 shows the structure of the interface working storage.

Table 8. Structure of the DSLAPI Interface Working Storage

Label	Offset (Hex.)	Length	Type	Description
INTFUNC	000	4	C (1)	The name of the DSLAPI function to be called. All functions are described in "Chapter 9. DSLAPI Functions" on page 97.
INTQUEUE	004	8	C (1)	Queue Management. Queue name.
INTQSN	00C	4	B	Queue Management. The unique queue sequence number (QSN) of a queue element.
INTKEY1	010	24	C	Queue Management. The first symbolic key of a queue element.
INTKEY2	028	24	C	Queue Management. The second symbolic key, if used.
INTBQUE	040	8	C (1)	Queue Management. The name of the MERVA queue containing the queue element that is to be automatically deleted.
INTBQSN	048	4	B	Queue Management. The queue sequence number (QSN) of the queue element that is to be automatically deleted.
INTDOUBL	04C	6	C	Queue Management. Indicates whether the queue element has been processed previously. Can contain "DOUBLE" or blanks.
INTBUSY	052	4	C	Indicates whether the queue element is *in-service*, that is, being processed by another task. Can contain "BUSY" or blanks.
INTSHUTD	056	8	C	Indicates the state of the MERVA ESA system. The value can be: • SHUTDOWN, the operator has entered the SHUtdown command. • INACTIVE, the MERVA ESA system is not ready. • Blanks, MERVA ESA is active.
INTRC	05E	2	C	The DSLAPI return code: see the relevant section of each function description in "Chapter 9. DSLAPI Functions" on page 97.
INTCWA	060	4	A	The address of the MERVA ESA service communication area DSLCOM or 0.
INTEISTG	064	4	A	The address of the CICS DFHEISTG (set by DSLAPCIC).
INTEIB	068	4	A	The address of the CICS DFHEIBLK (set by DSLAPCIC).
INTMSGID	06C	8	C (1)	MFS services. Message identification.

Table 8. Structure of the DSLAPI Interface Working Storage (continued)

Label	Offset (Hex.)	Length	Type	Description
INTFRMID	074	1	C (1)	MFS services. Message format identifier. The following codes are used by MERVA ESA: **K** Telex test-key; **N** Workstation based Telex (V3.2 or less); **P** Workstation based Telex (V3.3 or higher); **Q** Internal queue format; **S** SWIFT I; **T** Telex Link; **W** SWIFT II; **X** SWIFT I noprompt; **Y** SWIFT II noprompt
INTERMSG	075	131	C	May contain an explanatory message if INTRC contains a nonblank return code.
INTERMF1	0F8	79	C	MFS services. Following an MFS error, INTERMF1-3 can contain the first three data areas from the DSLMSG field in the message.
INTERMF2	147	79	C	DSLMSG data area 2.
INTERMF3	196	79	C	DSLMSG data area 3.
	1E5	3	C	Reserved.
INTTOFA	1E8	4	A	The storage address of the TOF. Refer to "Locating the TOF" on page 51.
INTQBUFA	1EC	4	A	The storage address of the internal queue buffer. Refer to "Locating the Internal Queue Buffer" on page 51.
	1F0	12		Reserved.
INTSIZE	1FC	4	B	The total size of API internal storage.
	200	3584		The remainder of INTWSTOR is used by DSLAPI internally.

A = Address
B = Binary value
C = Uppercase Character, (1) can be in lowercase				

Notes:

1. When you specify a function, DSLAPI sets the unused fields for:
   - Characters to blanks
   - Numbers or addresses to X'00'.
2. MERVA ESA requires that the interface working storage INTWSTOR be aligned on a doubleword boundary, that is, that its address be a multiple of eight.

Figure 7 shows which INTWSTOR fields you must specify for each function, and which fields may contain an answer after using a function.

	FUNC	QUEUE	QSN	KEY1	KEY2	BQUE	BQSN	SHUTD	DOUBL	BUSY	RC	CWA	EISTG	EIB	MSGID	FRMID	ERMSG	ERMFn
CMD	M							A			A		C	C			A	
DELE	M	M	M	A	A			A			A		C	C			A	
EMPT	M										A		C	C			A	
FLDG	M										A		C	C			A	
FLDP	M										A		C	C			A	
FREE	M	M	M	A	A			A			A		C	C			A	
GEKU	M	M	A	M	A			A	A	A	A		C	C			A	
GET	M	M	M	A	A			A	A	A	A		C	C			A	
GETC	M	M	M	A	A			A	A		A		C	C			A	
GETK	M	M	A	M	A			A	A	A	A		C	C			A	
GETM	M	M									A		C	C	M	M	A	A
GETN	M	M	M	A	A			A	A		A		C	C			A	
GETS	M										A		C	C			A	A
GETU	M	M	M	A	A			A	A	A	A		C	C			A	
INIT	M							A			A	M	C			C	A	
JRLG	M							A			A		C	C			A	
JRLN	M							A			A		C	C			A	
JRLP	M							A			A		C	C			A	
JRNG	M							A			A		C	C			A	
JRNN	M							A			A		C	C			A	
JRNP	M							A			A		C	C			A	
MPFG	M										A		C	C			A	A
MPFP	M										A		C	C			A	A
MSGG	M	M									A		C	C	M	M	A	A
MSGP	M	M									A		C	C	M	M	A	A
PRTI	M										A				M	M	A	A
PRTL	M										A						A	A
PRTT	M										A						A	A
PUT	M	M	A	A	A			A			A		C	C			A	
PUTB	M	M	A	A	A	M	M	A			A		C	C			A	
PUTM	M	M									A		C	C	M	M	A	A
PUTR	M	M	M	M	M			A	M	M	A		C	C			A	

M = Mandatory
A = Answer
C = CICS only, set by DSLAPCIC

Figure 7. Relationship between Interface Working Storage and Its Functions (Part 1 of 2)

	FUNC	QUEUE	QSN	KEY1	KEY2	BQUE	BQSN	SHUTD	DOUBL	BUSY	RC	CWA	EISTG	EIB	MSGID	FRMID	ERMSG	ERMFn
PUTS	M										A		C	C			A	A
QLF	M	M	M	M	M			A		A	A		C	C			A	
QLL	M	M	M	M	M			A		A	A		C	C			A	
QLN	M	M	M	M	M			A		A	A		C	C			A	
QLP	M	M	M	M	M			A		A	A		C	C			A	
READ	M										A		C	C			A	
REEN	M							A			A	M	C			C	A	
REPL	M	M	M	A	A			A			A		C	C			A	
ROU	M	M	A	A	A			A			A		C	C			A	
ROUB	M	M	A	A	A	M	M	A			A		C	C			A	
ROUD	M	M	M	A	A			A			A		C	C			A	
ROUN	M	M	M	A	A			A			A		C	C			A	
SAVE	M							A			A		C			C	A	
TERM	M																	
USRG	M							A			A		C	C			A	
USRN	M							A			A		C	C			A	
WRIT	M										A		C	C			A	
WTO	M							A			A		C	C			A	

M = Mandatory
A = Answer
C = CICS only, set by DSLAPCIC

Figure 7. Relationship between Interface Working Storage and Its Functions (Part 2 of 2)

TOF Access Parameters TOFPARM

The TOFPARM structure is used by the API TOF services READ, WRIT, and EMPT. It contains the field reference of the field being processed and the return code and reason code from the MERVA TOF supervisor.

Table 9 shows the structure of TOFPARM. The TOF access parameters are described in detail in “Field Reference” on page 86.

Table 9. Structure of the TOF Access Parameters

Label	Offset (Hex.)	Length	Type	Description
TOFREQ	000	4	C	Contains the most recent API TOF request.

Table 9. Structure of the TOF Access Parameters (continued)

Label	Offset (Hex.)	Length	Type	Description
TOFMODIF	004	40	C	Blanks, or a string containing one or more request modifiers, separated by commas or blanks.
TOFFDNAM	02C	8	C	The name of the field.
TOFFDNA1	034	8	C	Reserved.
TOFFDNL	03C	2	B	Nesting identifier (index): A binary number between 0 and 255.
TOFFDFG	03E	2	B	Field group index: A binary number between 1 and 255.
TOFFDOC	040	2	B	Repeatable sequence index: A binary number between 1 and 32767.
TOFFDDA	042	2	B	Data area index: A binary number between 1 and 32767.
TOFTSVRC	044	2	B	The return code from the TOF supervisor.
TOFTSVRS	046	2	B	The reason code from the TOF supervisor.
TOFFDOCA	048	2	B	The number of nested repeatable sequence indexes used in the fields TOFFDOC1 to TOFFDOC9.
TOFFDOC1	04A	2	B	The repeatable sequence index for the first level of nested repeatable sequences: A binary number between 1 and 32767.
TOFFDOC2	04C	2	B	The repeatable sequence index for the second level.
TOFFDOC3	04E	2	B	The repeatable sequence index for the third level.
TOFFDOC4	050	2	B	The repeatable sequence index for the fourth level.
TOFFDOC5	052	2	B	The repeatable sequence index for the fifth level.
TOFFDOC6	054	2	B	The repeatable sequence index for the sixth level.
TOFFDOC7	056	2	B	The repeatable sequence index for the seventh level.
TOFFDOC8	058	2	B	The repeatable sequence index for the eighth level.
TOFFDOC9	05A	2	B	The repeatable sequence index for the ninth level.

B = Binary value
C = Character				

Field Reference

Each field in a message can be precisely identified by a field name and four levels of indexing:

- Message nesting identifier
- Field group index
- One or more repeatable sequence indexes
- Data area index.

In MERVA ESA this is called a field reference.

When you access a field in a message you identify it by setting the field reference variables in the TOFPARM structure. You can also specify request modifiers to modify the way the TOF supervisor positions to a field. After the TOF access the TOFPARM field reference reflects the current position in the TOF.

Message Nesting Identifier TOFFDNL

Nesting identifier 0 is generated when the TOF is initialized. Nesting identifier 1 is generated when a message is initialized. Subsequent nesting identifiers are generated when a new message is initialized nested, or embedded, in an existing message. SWIFT message type 192 is an example of a message containing embedded messages.

If a nesting identifier between 1 and 255 is used, it must exist in the TOF. Fields with TOF nesting identifier 0 can always be written.

Field Group Index TOFFDFG

A field name can occur more than once in a message. These occurrences are identified by a different field group index. For SWIFT I messages, group index 1 is reserved for the first line of the header; group index 2 is reserved for the second and the third lines of the header. For SWIFT II messages, the basic header is in group index 1, the application header in group index 2, and the user header in group index 3. Message text fields start with group index 5. The trailer is always in group 255.

Starting with MERVA ESA V4.1 the group index is indicated in all MCBs for SWIFT messages with the label GRNnnn and the DSLLGRP operand GRPNUM=nnn, where nnn is the actual group index, for example:

```
GRN005   DSLLGRP   GRPNUM=5
```

Repeatable Sequence Index TOFFDOC

The repeatable sequence index TOFFDOC identifies one sequence of fields in a repeated sequence of fields. A SWIFT example is message type 412.

Repeatable sequences can be nested; a repeatable field sequence is nested when it is defined within a sequence of fields that itself is repeated. An example of a message containing nested repeatable sequences is SWIFT message type 801. To uniquely identify a field in a nested repeatable sequence you need an index for each nesting level. The TOFPARM can define up to nine levels of repeatable-sequence indexes in the fields TOFFDOC1 to TOFFDOC9. The field TOFFDOCA defines the number of these indexes that are actually being used. TOFFDOC1-9 are only used if RSEXT is specified in the modifier field TOFMODIF. In this case TOFFDOC is ignored.

Data Area Index TOFFDDA

A data area index identifies a specific data area of a field consisting of multiple data areas. Some fields have an option area besides the data areas.

Request Modifiers

The modifier group consists of a string of 40 characters, containing modifiers separated by commas or blanks. A complete list of all modifiers is given in the description of the DSLTSV macro in *MERVA for ESA Macro Reference* . Both one-time modifiers, which affect TOF positioning (DSLTSV parameter MODIF), and function modifiers, which affect the way the TOF service works (DSLTSV parameter FMODIF), can be specified. Some of the more important modifiers are:

OPTION	The option is processed rather than the field data.

VFIRST	The first field with the given name is processed.
FIRSTDA	The first data area of a field is processed.
NEXTDA	The next data area of a field is processed.
LASTDA	The last data area of a field is processed.
EDIT	Function modifier to show that the data has to be edited. The MFS edit routine number is specified in the Field Definition Table DSLFDTT.
CHECK	Function modifier to show that the data has to be checked. The MFS checking routine number is specified in the Field Definition Table DSLFDTT.
OPTLIST	Function modifier to show that the option list has to be read; only for FUNCTION=READ.
DELDA	Function modifier to show that a specific data area of a field has to be deleted; only for FUNCTION=EMPT.
DELFN	Function modifier to show that a field has to be deleted; only for FUNCTION=EMPT.
DELAD	Function modifier to show that all data areas of a field have to be deleted; the field itself still exists in the TOF. This modifier is only for FUNCTION=EMPT.
RSEXT	One-time modifier, which indicates that the RS extension in TOFFDOCA, TOFFDOC1 to TOFFDOC9, is to be used for TOF access. This modifier is necessary if messages with nested repeatable sequence are being processed.

TOF Reason Codes

Refer to the *MERVA for ESA Messages and Codes* manual for a list of possible DSLTSV reason codes and their meanings.

Message Buffer MSGSWIFT

The MSGSWIFT buffer is used by the MFS functions GETS, GETM, PUTS, and PUTM for the mapping of messages (not just SWIFT messages) smaller than 32000 bytes. The buffer is defined by the copybook DSLAPIMS. MSGSWIFT consists of two parts:

- A header, the MSGSWIFT_PREFIX (defined separately by copybook DSLAPIMP).
- A buffer, MSGSMSG, with the standard MERVA 8-byte buffer prefix, containing a message in net format.

The structures MS1IH, MS1OH, MS2BH, MS2AI, and MS2AO, in copybook DSLAPIMH, define SWIFT message headers in both SWIFT I and SWIFT II format.

MSGSWIFT Prefix

The MSGSWIFT_PREFIX structure, copybook DSLAPIMP, is part of the MSGSWIFT structure used by the MFS services GETS, GETM, PUTS, and PUTM. The MFS services MPFG and MPFP can be used toread and write the prefix alone.

Not all fields in the structure are used. For PUTM, PUTS, and MPFP, only the MSGACK field, and the user fields MSGDBS, MSGUSER1, and MSGADDR1-4 are added to the message being built in the internal queue buffer. Figure 10 shows the layout of the MSGSWIFT prefix.

Table 10. Structure of the MSGSWIFT Prefix

Label	Offset (Hex.)	Length	Type	Description
MSGLGTH	000	2	B	Length of prefix
MSGOS	002	2	A	Reserved
MSGTRAN	004	8	C	Not used
	00C	1		Not used (IMS)
MSGRPGM	00D	8	C	Not used
MSGSPGM	015	8	C	Not used
MSGMTYPE	01D	1	C	Type of message (SWIFT I or SWIFT II) **F** = Unformatted message **O** = Output message **I** = Input message
MSGRC	01E	2	C	Not used
				User Information
MSGDBS	020	24	C	User field (example: Key SSA in data base)
MSGUSER1	038	24	C	User field
MSGADDR1	050	35	C	User field (example: Header address)
MSGADDR2	073	35	C	User field
MSGADDR3	096	35	C	User field
MSGADDR4	0B9	35	C	User field
MSGQUEUE	0DC	8	C	Not used
MSGLTERM	0E4	8	C	Not used
MSGQSN	0EC	4	B	Not used
MSGKEY1	0F0	24	C	Not used
MSGKEY2	108	24	C	Not used
MSGNET	120	8	C	The message identifier
MSGDST	128	9	C	SWIFT Master Destination
MSGACK	131	69	C	ACK/NAK (SWIFT, TELEX) (see below)
	176	2		Reserved

A = Address
B = Binary value
C = Character				

SWIFT ACK/NAK Structure

If a message is being retrieved, the MSGACK field in the MSGSWIFT prefix contains the message's acknowledgement. But note that a SWIFT II acknowledgement is truncated. To read a SWIFT II acknowledgement use the TOF READ service.

The structure of the SWIFT I acknowledgement is defined as part of the MSGSWIFT_PREFIX structure, copybook DSLAPIMP, and is shown in Table 11 on page 90 The offset is the offset from the beginning of the MSGSWIFT structure.

Table 11. Structure of the SWIFT I Acknowledgement

Label	Offset (Hex.)	Length	Description
MSGAACK	131	3	Type: ACK or NAK
	134	1	
MSGAINTI	135	4	Input time
	139	1	
MSGASRN	13A	19	System reference number (SRN)
	14D	1	
MSGAERR	14E	12	...

The structure of the SWIFT II acknowledgement is shown in Table 12. A copybook mapping is not provided.

Table 12. Structure of the SWIFT II Acknowledgement

Label	Offset (Hex.)	Length	Description
MSGA2BH	00	6	Block ID, Appl.ID, APDU ID
MSGA2LTA	06	12	Logical Terminal
MSGA2SN	12	4	Session number
MSGA2ISN	16	6	Sequence number
	1C	9	
MSGA2TIM	25	10	Date and time of ACK
	2F	6	
MSGA2ACC	35	1	Acceptance/Rejection
	36	6	
MSGA2ERR	3C	3	Error code
MSG2ALIN	3F	3	
	42	6	
MSG2AMUR	48	14	MUR
	56	2	

Message Buffer MSGSMSG

The message buffer MSGSMSG holds the actual message. Mappings for the fixed parts of SWIFT messages MS1IH, MS1OH, MS2BH, MS2AI, and MS2AO, as shown in Table 13 on page 91, are provided in the copybook DSLAPIMH. In Table 13 on page 91, offsets are given from the start of text following the MSGSMSG 8-byte buffer prefix.

Table 13. Structure of the SWIFT Message

Label	Offset (Hex.)	Length	Description
S.W.I.F.T. User Handbook: SWIFT I Input Header			
	00	2	
MSGISDST	02	12	Destination of sending bank
	0E	1	
MSGIISN	0F	5	Input sequence number (ISN)
	14	2	
MSGIMT	16	3	Message type
	19	1	
MSGIPR	1A	2	Message priority
	1C	2	
MSGIRDST	1E	11	Destination of receiving bank
	29	2	
MSGITEXT	2B		Text ...
S.W.I.F.T. User Handbook: SWIFT I Output Header			
	00	2	
MSGOOUTI	02	4	Output time
	06	1	
MSGOORN	07	19	Output reference number (ORN)
	1A	2	
MSGOINTI	1C	4	Input time
	20	1	
MSGOSRN	21	19	System reference number
	34	2	
MSGOMT	36	3	Message type
	39	1	
MSGOPR	3A	2	Message priority
	2C	2	
MSGOTEXT	2E		Message text ...
S.W.I.F.T. User Handbook: SWIFT II Basic Header			
MS2BHBID	00	3	Block identifier
MS2BHAID	03	1	Application identifier
MS2BHMID	04	2	Message/APDU identifier
MS2BHLTA	06	12	LT address
MS2BHSES	12	4	Session number (0)
MS2BHSEQ	16	6	Sequence number
MS2BHEND	1C	1	Block

S.W.I.F.T. User Handbook: SWIFT II Application Header, FIN Input			
	00	29	Basic Header
MS2AIBID	1D	3	Block identifier
MS2AIID	20	1	Input/output identifier
MS2AIMTY	21	3	Message type
MS2AIDST	24	12	Destination address
MS2AIMPR	30	1	Message priority
MS2AIIDM	31	1	Delivery monitoring (optional)
MS2AIIOP	32	3	Obsolescence period (optional)
MS2AIEND	35	1	Block

S.W.I.F.T. User Handbook: SWIFT II Application Header, FIN Output			
	00	29	Basic Header
MS2AOBID	1D	3	Block identifier
MS2AOID	20	1	Input/output identifier
MS2AOMTY	21	3	Message type
MS2AOITI	24	4	Input time (HHMM)
MS2AOTOR	28	28	Input MIR
MS2AOODA	44	6	Output date (YYMMDD)
MS2AOOTI	4A	4	Output time (HHMM)
MS2AOMPR	4E	1	Message priority
MS2AOEND	4F	1	Block

Journal Key JRNKEY

Table 14 shows the structure of the new journal key used with four-digit year format. Only the first part, the date and the time stamp, is changed, the total length of the journal key header is unchanged.

Table 14. Structure of the Journal Key with Four-Digit Year

Label	Offset (Hex.)	Length	Type	Description
JRNRID	00	1	X	Record identifier
JRNKDAT2	01	8	C	Record key: Date (YYYYMMDD)
JRNKTIM2	09	6	C	Time (HHMMSS)
JRNKFRC2	0F	3	X	Fractional part (PPP)
JRNKSEG	12	3	C	Segment number
	15	1	C	(/)
JRNKSEGS	16	3	C	Number of segments
JRNKUSER	19	25	C	User extension

X = Hexadecimal
C = Character				

Table 15 shows the structure of the old journal key.

Table 15. Structure of the Journal Key with Two-Digit Year

Label	Offset (Hex.)	Length	Type	Description
JRNRID	00	1	X	Record identifier
JRNKDATE	01	6	C	Record key: Date (YYMMDD)
	07	1	C	(/)
JRNKTIME	08	10	C	Time (HHMMSSPPNN)
JRNKSEG	12	3	C	Segment number
	15	1	C	(/)
JRNKSEGS	16	3	C	Number of segments
JRNKUSER	19	25	C	User extension

X = Hexadecimal
C = Character				

Terminal User Control Block (TUCB)

If you program a transaction and the transaction is initiated by MERVA ESA, it will be passed a Terminal User and Control Block. Table 16 shows some fields of the structure of the TUCB.

Table 16. Structure of the Interface Terminal and User Control Block TUCB

Label	Offset (Hex.)	Length	Type	Description
	00	6		Filler
TUCBTRAN	06	8	C	Transaction Name
	0E	10		Filler
TUCNAME	18	8	C	Function Name
	20	80		Filler
TUCLTE1	70	8	C	Logical Terminal Name
	78	392		Filler
C = Character				

User File Record Buffer

The user file record buffer is used to hold user file records retrieved by the functions USRG and USRN. The fields of the authorized part of the user file record are shown in Table 17.

Table 17. Structure of the User File Record

Label	Offset (Hex.)	Length	Type	Description
	000	8	C	MERVA buffer prefix
USRUKEY	008	8	C	Key (User ID)
USRUSCPW	010	8		Password (scrambled)
USRUNAME	018	18	C	User name
USRUORID	02A	34	C	Origin ID
USRUDATE	04C	8	C	Date of last update
USRUTIME	054	8	C	Time of last update
USRUUUID	05C	8	C	Update user ID
USRUDATP	064	8	C	Date of last password change
USRUTIMP	06C	8	C	Time of last password change
USRUPFKS	074	8	C	PF-key setname
USRULID	07C	1	C	Language ID
USRUNLIF	07D	1	C	No-prompt line format
USRUDNW	07E	1	C	Default network
	07F	1		Reserved
USRUFTAB (18)	080	8	C	Allowed functions
USRUAMSG (24)	110	8	C	Message types assigned to user
USRUNOCM	1D0	60	C	Commands forbidden to user
USRUUFLM	20C	8	C	FLM administrator
	214	17		Reserved
USRUDATS	225	8	C	Date of last sign-on
USRUIMRX	22D	1	B	Traffic Reconciliation user class
USRUGRP	22E	8	C	Group ID
USRUUSON	236	1	B	Number of rejected sign-ons
USRUUTYP	237	1	C	User type
USRUUDTA	238	48	C	User data
USRUUDTB	268	48	C	User data
	298	636		Reserved

B = Binary
C = Uppercase character				

MFS Parameter List

Table 18 shows the structure of the MFS parameter list.

Table 18. Structure of the MFS Parameter List

Label	Offset (Hex.)	Length	Type	Description
MFSLTYP	00	1	X	MFS Function
MFSLMED	01	1	X	Medium
MFSLOPT1	02	1	X	Option-code
MFSLOPT2	03	1	X	Option-code
	04	1		
MFSLRET	05	1	X	The return code you set when you return control to MERVA ESA
MFSLREAS	06	2	B	The reason code that you return to MERVA ESA and that describes more precisely any errors. Possible values are described in the structure MFSLREAS, copybook DSLMREAS.
MFSLMODN	08	2	B	Exit no., or operator message no. (TYPE=ERRMSG) The module number of the exit (MFS exits are identified by number). If you are writing several similar exit routines, it might be easier to write a single exit routine to process them all. MFSLMODN indicates for which particular exit your routine has been invoked.
MFSLMILF	0A	1	C	Line format
MFSLWORK	0B	1	X	Indicators
MFSLCOMA	0C	4	A	Address of MERVA ESA communication area
	10	4		Reserved
MFSLPERM	14	4	A	Address of MFS permanent storage
MFSLTEMP	18	4	A	Address MFS of temporary storage
MFSLENVA	1C	4	A	Address of environment string
MFSLMSG	20	4	A	Address of message ID, load module name
MFSLTOF	24	4	A	Address of TOF
MFSLFLD	28	4	A	Address of field reference
MFSLIBUF	2C	4	A	A pointer to the input buffer, which has a standard MERVA buffer prefix, and in which data to be processed by your exit routine is passed from MERVA ESA. For example, a checking exit routine would receives the name of the field to be checked in this buffer.
MFSLOBUF	2C	4	A	A pointer to the output buffer, which has a standard MERVA buffer prefix, and in which your routine passes data back to MERVA ESA.

A = Address
B = Binary
C = Character
X = Hexadecimal				

Chapter 9. DSLAPI Functions

The API functions supported by the DSLAPI program are described in this chapter in alphabetical order. The following information is provided for each function:

- Purpose of the function
- The DSLAPI parameter list for the function
- Return Codes
- Notes explaining how the function is used
- Examples of how to code the function.

The parameter lists are described in a language-independent way, for example:

```
►►──EMPT──(──INTWSTOR──,──TOFPARM──)─────────────────────────────────────►◄
```

In Assembler you must construct the parameter list yourself:

```
         MVC   INTFUNC,=C'EMPT'    The API service
         LA    R1,INTWSTOR
         ST    R1,PARMLIST
         LA    R1,TOFPARM
         ST    R1,PARMLIST+4
         OI    PARMLIST+4,X'80'    end-of-list indicator
         LA    R1,PARMLIST
         L     R15,=V(DSLAPI)
         BALR  R14,R15
```

In C/370 the call might look like this:

```
      memcpy(ws.INTFUNC,"EMPT",4);
      DSLAPI(&ws, &tofpl);
```

In COBOL, after setting the TOFPARM values, you could invoke this function like this:

```
      move 'empt' to intfunc of api-ws.
      call dslapi using api-ws, tof-parameters.
```

or in PL/I like this:

```
      api_ws.intfunc = 'empt';
      call dslapi(api_ws, tof_parameters);
```

The API REXX interface does not use parameter lists:

```
      toffdnam = ...             /* set tofparm values */
      Address DSLAPI "EMPT"      /* invoke the function */
```

CMD Execute a MERVA Command

The CMD function passes a MERVA ESA operator command to MERVA and returns the command response. Any commands that can be input at the MERVA ESA Command function panel can be executed using this API service.

The Queue Test commands MOVE, DELETE, DELX, COPY, or FREE, and the journal command JRN cannot be executed using the CMD function; these commands should be implemented using the DSLAPI queue management and journal functions.

```
►►──CMD──(──INTWSTOR──,──command──,──response──)─────────────────────────────►◄
```

INTWSTOR
: API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

command
: The *command* buffer is a fixed-length buffer 120 bytes long and contains the command string to be passed to MERVA.

response
: A 700-byte fixed-length buffer. DSLAPI places the command response into this buffer. The response contains 10 lines of data; each line is 70 bytes long.

Usage Notes

- If the response is longer than 10 lines, you should resend the command to get the next group of lines. Repeat this until the last lines have been returned. This is what you do when you use a command from the MERVA ESA Command function panel.

  You can determine when all lines have been returned only by inspecting the data in the response buffer.
- Resend only display commands with continuation information. These are: DF, DICB, DM, DNS, DP, DQ, DQSORTED, DU, DL, and DLA.
- If the response is shorter than 10 lines, the remaining lines are blank.

Return Codes

INTRC = spaces
: The call was successful.

INTRC = 02
: The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
77  cmdinp                 pic x(120).
01  cmdresp.
    02 resp-line           pic x(70) occurs 10 indexed by i.
...
    move 'DM LAST' to cmdinp
    move 'CMD ' to intfunc
    call dslapi using intwstor, cmdinp, cmdresp
    if intrc = spaces then
      perform varying i from 1 by 1 until i > 10
        display '  ' resp-line (i)
      end-perform
...
```

Here is an example in C/370:

```
#include "dslapc.h"
 ...
  struct INTWSTOR ws;
  char cmdinp[120];
  char cmdresp[10][70];
 ...
    memcpy(cmdinp,"DM LAST",7);
    memcpy(ws.INTFUNC,"CMD ",4);
    DSLAPI(&ws,cmdinp,cmdresp);
    if (memcmp(ws.INTRC,"  ",2) != 0) {
 ...
```

Here is an example in REXX:

```
...
cmdinp  = 'DM LAST'
cmdresp = ''                            /* be tidy */
Address DSLAPI "CMD"
If intrc = ' '
Then Do
   Do i = 0 To 9
      Say Substr(cmdresp,1 + i * 70,70)    /* display the response  */
   End
End
Else
   Say 'MERVA API command CMD failed with intrc' intrc'.'
...
```

DELE Delete a Queue Element

The DELE function deletes a queue element from a MERVA ESA queue.

```
►►──DELE──(──INTWSTOR──)──────────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR must be set:

INTQUEUE
The name of the MERVA ESA queue containing the queue element to be deleted.

INTQSN
The queue sequence number (QSN) of the queue element to be deleted.

Usage Notes

DELE following ROUB or PUTB

After a ROUB or PUTB (route or put with automatic delete) a specific delete is superfluous. However, when the DSLAPI call immediately preceding the delete was a ROUB or PUTB in which the queue element to be automatically deleted, identified by INTBQUE and INTBQSN, is also the element specified in the DELE, DSLAPI returns with INTRC=spaces. This is to maintain compatibility with previous versions.

Normally, when a nonexistent queue element is deleted, DSLAPI returns INTRC=02.

Return Codes

INTRC = spaces
The call was successful.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
77  next-qsn                pic s9(8) binary.
77  our-queue               pic x(8).
...
    move next-qsn to intqsn
    move our-queue to intqueue
    move 'dele' to intfunc
    call dslapi using intwstor
    if intrc not = spaces then
...
```

Here is an example in C/370:

```
...
 struct INTWSTOR ws;
...
   ws.INTQSN = 2;
   memset(ws.INTQUEUE,' ',sizeof ws.INTQUEUE);
   memcpy(ws.INTQUEUE,"L1DO0",5);
   memcpy(ws.INTFUNC,"DELE",4);
   DSLAPI(&ws);
   if (memcmp(ws.INTRC,"  ",2) == 0) {
...
```

Here is an example in REXX:

```
...
intqueue = 'L1DO0'
intqsn   = 5
Address DSLAPI "DELE"
If intrc = ' '
Then Do
   Say 'MERVA API command DELE was successful.'
   Say 'Queue name :' intqueue
   Say 'QSN ...... :' intqsn
End
Else
   Say 'MERVA API command DELE failed with intrc' intrc'.'
...
```

EMPT Empty a Field (TOF)

The EMPT function deletes a field, a data area, or all data areas of a field in the internal queue buffer, through the internal TOF, using the field-reference structure. The field TOFMODIF must be set with the appropriate modifier.

```
►►──EMPT──(──INTWSTOR──,──TOFPARM──)─────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

TOFPARM
The field reference of the field to be deleted, or of the field containing the data area or areas to be deleted.

Usage Notes

- The modifiers you will normally use are:

DELFN	Deletes the field.
DELDA	Deletes the specified data area.
DELAD	Deletes all data areas of the field.
DELDAGR	Deletes all data areas with a data area index greater than the specified data area.

  See “TOF Access Parameters TOFPARM” on page 85 and the description of macro DSLTSV in the *MERVA for ESA Macro Reference* for more details about field references and request modifiers.
- After TOF access calls (like EMPT) the input field-reference data might have been changed by the TOF supervisor to its output parameters. Note especially that after an unsuccessful EMPT the returned TOFFDNAM value is unpredictable, that is, does not contain the failing TOFFDNAM parameter.

Return Codes

INTRC = spaces
The call was successful.

Note: This return code only suggests that control was successfully passed to the TOF supervisor. The TSV return code TOFTSVRC and TSV reason code TOFTSVRS in the TOFPARM structure show the actual result.

INTRC = 01
The function has failed for one of the following reasons:
- Failure to map from the internal queue buffer to the internal TOF
- Failure to map from the internal TOF to the internal queue buffer.

Additional information is contained in field INTERMSG.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
copy dslapitp.
...
    move 'SW73' to toffdnam
    move 1 to toffdnl
    move 9 to toffdfg
    move 2 to toffdoca
    move 3 to toffdnoc(1)
    move 1 to toffdnoc(2)
    move 4 to toffdda
    move 'DELDAGR RSEXT' to tofmodif
    move 'empt' to intfunc
    call dslapi using intwstor tofparm
    if intrc not = spaces then
...
```

Here is an example in REXX:

```
...
/* GETC and FREE a queue element */
intqueue = 'L1DE0'                    /* queue name              */
intqsn   = 123                        /* queue sequence number   */
Address DSLAPI "GETC"
If intrc ¬= ' ' Then ...
Address DSLAPI "FREE"
If intrc ¬= ' ' Then ...

/* EMPTy field SW73 */
toffdnam  = 'SW73'                    /* name of the field       */
toffdnl   = 1                         /* nesting level index     */
toffdfg   = 9                         /* field group index       */
toffdoca  = 2                         /* no. of nested rep. seqs */
toffdoc.1 = 3                         /* rep. sequence index 1   */
toffdoc.2 = 1                         /* rep. sequence index 2   */
toffdda   = 4                         /* data area index         */
tofmodif  = 'DELDAGR RSEXT'           /* request modifier        */
Address DSLAPI "EMPT"
If intrc = ' ' & toftsvrc = 0 & toftsvrs = 0
Then
   Nop                                /* ok, continue            */
Else
   ...                                /* EMPT failed             */

/* PUT queue element to another queue */
intqueue = 'L2DE0'                    /* target queue name       */
Address DSLAPI "PUT"
...
```

The code fragments show how to delete some data areas from the field with the name SW73. This field occurs in a S.W.I.F.T message type 801. According to the MERVA ESA convention, the field is on nesting identifier (toffdnl) 1 and has the group index (toffdfg) 9.

The field is in a nested repeatable sequence on the second level; therefore two occurrence numbers fully qualify the field. In the above example the two numbers are 3 and 1, which are set into the array of occurrence numbers (toffdnoc). The array can hold up to nine numbers, therefore you have to specify that only the first and the second array element is used (toffdoca). Because this field is used in a nested repeatable sequence you must also specify the modifier RSEXT. The other modifier DELDAGR indicates that all data areas after data area 4 (toffdda) are to be deleted. The first four data areas are not deleted.

FLDG Get a MERVA Variable

Product-Sensitive Programming Interface

This function retrieves the value of a specified field of a MERVA internal structure.

```
►►──FLDG──(──INTWSTOR──,──field-name──,──buffer──)────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

field-name
A 40-byte fixed-length buffer containing the name, left-justified, of the field to be retrieved. Refer to “Field-Level Access for Exit Routines” on page 33 for more information on this service.

buffer A fixed-length buffer in which the value of the field to be retrieved is returned left justified. The length of the buffer is the length of the value you are retrieving.

The value can be in one of the following forms depending on the data type of the specified field:

- Character string
- 4-byte, fullword binary value
- The characters ‘0’ or ‘1’
- Eight ‘0’ or ‘1’ characters
- A packed-decimal value.

Refer to “Field-Level Access for Exit Routines” on page 33 for more information.

Usage Notes

The names of all fields, their type, and their lengths, are listed in “Appendix D. Field-Level Access Fields” on page 269.

Return Codes

INTRC = spaces
The call was successful.

INTRC = 02
The call failed for one of the following reasons:

- The specified name is not known or not supported
- The structure containing the field is not addressable.

Additional information is contained in field INTERMSG.

Examples

Here is an example in COBOL:

```
...
77  field-name             pic x(40).
77  field-value            pic x(8).
...
    move 'NPNAME' to field-name
    move spaces to field-value
    move 'fldg' to intfunc
    call dslapi using intwstor field-name field-value
    if intrc = spaces then
      if field-value(1:8) = 'MERVAESA' then
...
```

Here is an example in REXX:

```
...
fldname  = 'NPNAME'                 /* MERVA name              */
fldvalue = ''                       /* be tidy                 */
Address DSLAPI "FLDG"
If intrc = ' '
Then Do
   If fldvalue = 'MERVAESA' Then
 ...
```

End of Product-Sensitive Programming Interface

FLDP Set a MERVA Variable

Product-Sensitive Programming Interface

This function moves a value to the specified field of a MERVA internal structure.

```
►►──FLDP──(──INTWSTOR──,──fieldname──,──buffer──)────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

field-name
A 40-byte fixed-length buffer containing the name, left-justified, of the field to be modified. Refer to "Field-Level Access for Exit Routines" on page 33 for more information on which names are valid.

buffer A fixed-length buffer containing the value to be written to the field, left-justified. The length of the buffer is the length of the field value.

The value must have one of the following forms depending on the data type of the specified field:

- Character string
- 4-byte, fullword binary value
- The characters '0' or '1'
- Eight '0' or '1' characters
- A packed-decimal value.

Refer to "Field-Level Access for Exit Routines" on page 33 for more information.

Usage Notes

The names of all fields, their type, and their lengths, are listed in "Appendix D. Field-Level Access Fields" on page 269.

Return Codes

INTRC = spaces
The call was successful.

INTRC = 02
The call failed for one of the following reasons:

- The specified name is not known or not supported
- The specified name may not be modified
- The structure containing the field is not addressable.

Additional information is contained in field INTERMSG.

Examples

Here is an example in COBOL:

```
...
copy dslapiws.
77  field-name              pic x(8).
77  field-value             pic x(1).
...
    move 'COMTRAMF' to field-name
    move '1'        to field-value
    move 'fldp' to intfunc
    call dslapi using intwstor field-name field-value
    if intrc not = spaces then
...
```

Here is an example in REXX:

```
...
fldname  = 'COMTRAMF'                /* field name              */
fldvalue = 1                         /* field value             */
Address DSLAPI "FLDP"
If intrc = ' '
Then Do
   Say 'MERVA API command FLDP was successful.'
   Say 'The field' fldname 'has now the value' fldvalue'.'
   If fldvalue = 0 Then Say 'The MERVA TRACE MFS flag is reset.'
                   Else Say 'The MERVA TRACE MFS flag is set.'
End
Else
...
```

End of Product-Sensitive Programming Interface

FREE Free a Queue Element

The FREE function resets the *in-service* indicator of a queue element in a MERVA ESA queue.

FREE also posts an ECB and starts a transaction, when defined for the associated queue.

```
▶▶—FREE—(—INTWSTOR—)——————————————————————————————————————▶◀
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR are used:

INTQUEUE
The name of the MERVA ESA queue or function containing the queue element to be reset.

INTQSN
The queue sequence number (QSN) of the queue element to be reset.

Return Codes

INTRC = spaces
The call was successful. Additional information is contained in fields INTKEY1 and INTKEY2.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
77  saveqsn            pic s9(8) binary.
77  queue-name         pic x(8).
...
    move 'FREE' to intfunc
    move saveqsn to intqsn
    move queue-name to intqueue
    call dslapi using intwstor
...
```

Here is an example in C/370:

```
  ...
 #include "dslapc.h"
 int main() {
   struct INTWSTOR ws;
   int saveqsn;
   char queue[8];
  ...
       memcpy(ws.INTFUNC,"FREE",4);
       memcpy(ws.INTQUEUE,queue,8);
       ws.INTQSN = saveqsn;
       DSLAPI(&ws);
  ...
```

Here is an example in REXX:

```
...
intqueue = queue_name                  /* queue name              */
intqsn   = saveqsn                     /* queue sequence number   */
Address DSLAPI "FREE"
If intrc = ' '
Then Do
...
```

GEKU Get a Queue Element by Key Unconditionally

The GEKU function retrieves the queue element with the specified symbolic key from the MERVA ESA queue and puts it in the internal queue buffer.

The function is unconditional because it retrieves a queue element regardless of its *in-service* status. Therefore the function should only be used for read-only retrieval.

The GEKU function retrieves queue elements only from a queue in nohold status.

```
►►──GEKU──(──INTWSTOR──)──────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR must be set:

INTQUEUE
The name of the MERVA ESA queue or function containing the queue element to be retrieved.

INTKEY1
The symbolic key to be used to retrieve an element based on the queues first key.

INTKEY2
The symbolic key to be used to retrieve an element based on the queues second key. This key is ignored if INTKEY1 is not blank.

Usage Notes

If more than one queue element matches a specified key value, GEKU will always return the first one (the one with the lowest QSN). You can use the QLF and QLN function to loop through all queue elements matching a specified key 1 and/or key 2 value.

Return Codes

INTRC = spaces
The call was successful. Additional information is contained in fields INTQSN, INTKEY2, INTBUSY, and INTDOUBL.

INTRC = 01
INTQUEUE is not defined. Additional information is contained in fields INTERMSG.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

INTRC = 09
No queue element exists that matches INTKEY1 or INTKEY2.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
77  dslapi                 pic x(8) value 'DSLAPI'.
...
    move spaces to intkey1
    move ' Statistics Summary' to intkey2
    move 'tx2tlc' to intqueue
    move 'geku' to intfunc
    call dslapi using intwstor
    if intrc not = spaces then
...
```

Here is an example in REXX:

```
...
intqueue = 'TX2TLC'                   /* queue name             */
intkey1  = ' '                        /* key 1                  */
intkey2  = ' Statistics summary'      /* key 2                  */
Address DSLAPI "GEKU"

If intrc = ' '
Then Do
   Say 'MERVA API command GEKU was successful.'
   Say 'Queue name :' intqueue
   Say 'QSN ...... :' intqsn
   Say 'Key 1 .... :' intkey1
   Say 'Key 2 .... :' intkey2
End
Else
   Say 'MERVA API command GEKU failed with intrc' intrc'.'
...
```

GET Get a Queue Element Unconditionally

The GET function retrieves the specified queue element and puts it in the internal queue buffer.

The function is unconditional because it retrieves a queue element regardless of its *in-service* status. Therefore the function should only be used for read-only retrieval.

```
►►──GET──(──INTWSTOR──)──────────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR must be set:

INTQUEUE
The name of the MERVA ESA queue containing the required queue element.

INTQSN
The queue sequence number (QSN) of the required queue element. If a QSN of 0 is specified, the first element in the queue is returned.

Usage Notes

Return Codes

INTRC = spaces
The call was successful. Additional information is contained in fields INTKEY1, INTKEY2, INTBUSY, and INTDOUBL.

INTRC = 01
INTQUEUE is not defined. Additional information is contained in field INTERMSG.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
77  dslapi                 pic x(8) value 'DSLAPI'.
...
    move 0 to intqsn
    move 'L2FORMS' to intqueue
    move 'get' to intfunc
    call dslapi using intwstor
    if intrc not = spaces then
...
```

Here is an example in REXX:

```
...
intqueue = 'L2FORMS'                   /* queue name              */
intqsn   = 0                           /* queue sequence number   */
Address DSLAPI "GET"
If intrc = ' '
Then Do
...
```

GETC Get a Queue Element Conditionally

The GETC function retrieves the specified queue element and puts it in the internal queue buffer.

The function is conditional because it retrieves a queue element only when it is not *in-service*. It flags the queue element *in-service* and sets the DOUBLE (write-back) indicator.

```
►►──GETC──(──INTWSTOR──)──────────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR must be set:

INTQUEUE
The name of the MERVA ESA queue containing the required queue element.

INTQSN
The queue sequence number (QSN) of the required queue element.

Return Codes

INTRC = spaces
The call was successful. Additional information is contained in fields INTKEY1, INTKEY2, and INTDOUBL.

INTRC = 01
INTQUEUE is not defined. Additional information is contained in field INTERMSG.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

INTRC = 08
The queue element exists but is *in-service*.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
77  dslapi                 pic x(8) value 'DSLAPI'.
...
    move current-qsn to intqsn
    move 'L2FORMS' to intqueue
    move 'getc' to intfunc
    call dslapi using intwstor
    if intrc not = spaces then
...
```

GETK Get a Queue Element by Key

The GETK function retrieves a queue element by key, if it is not flagged *in-service*, from the MERVA ESA queue and puts it in the internal queue buffer.

The function flags the queue element *in-service* and sets the DOUBLE (write-back) indicator.

When the first symbolic key is blank, the second symbolic key is used.

```
►►──GETK──(──INTWSTOR──)───────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR must be set:

INTQUEUE
The name of the MERVA ESA queue or function containing the queue element.

INTKEY1
The symbolic key to be used to retrieve an element based on the queues first key.

INTKEY2
The symbolic key to be used to retrieve an element based on the queues second key. This key is ignored if INTKEY1 is not blank.

Usage Notes

If more than one queue element matches a specified key value, GETK will always return the first one (the one with the lowest QSN). If that queue element is *in-service*, INTRC=09 will be returned. You can use the QLF and QLN function to loop through all queue elements matching a specified key 1 and/or key 2 value.

Return Codes

INTRC = spaces
The call was successful. Additional information is contained in fields INTQSN, INTKEY1, INTKEY2, and INTDOUBL.

INTRC = 01
INTQUEUE is not defined. Additional information is contained in fields INTERMSG.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

INTRC = 09
No queue element matches the specified key INTEK1 or INTKEY2 value, or the first matching queue is *in-service*. Additional information is contained in field INTBUSY.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
77  dslapi                 pic x(8) value 'DSLAPI'.
...
    move trn to intkey1
    move 'l2ack' to intqueue
    move 'getk' to intfunc
    call dslapi using intwstor
    if intrc not = spaces then
...
```

Here is an example in REXX:

```
...
intqueue = 'L2ACK'                     /* queue name              */
intkey1  = trn                         /* key 1                   */
Address DSLAPI "GETK"
If intrc = ' '
Then Do
...
```

GETM Get Message (MFS)

The GETM function maps a message from MERVA internal format to an external format. The message in the API internal queue buffer is mapped through the internal TOF to the MSGSWIFT buffer using the message identifier specified in the INTWSTOR field INTMSGID, and the format identifier specified in INTFRMID.

Notes:

1. The message identifier of the message is written to the field MSGNET in the MSGSWIFT prefix.
2. You should prefer the MSGG function to the GETM function.

```
►►──GETM──(──INTWSTOR──,──MSGSWIFT──)──────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR are used:

INTMSGID
INTMSGID identifies an MCB. If INTMSGID is blank, the message identifier is taken from the exit field in the message.

INTFRMID
INTFRMID identifies the line format in the MCB. If INTFRMID is blank, the first line format in the MCB is used.

MSGSWIFT
A variable-length buffer defined by the MSGSWIFT structure, copybook DSLAPIMS. The buffer size is defined by the APISMSG parameter in the MERVA ESA parameter module DSLPRM.

Return Codes

INTRC = spaces
The call was successful. Additional information is contained in the MSGSWIFT prefix fields MSGMTYPE, MSGNET, MSGDEST (SWIFT Link), MSGACK, and User Fields.

INTRC = 00
MFS has detected checking errors. Additional information is contained in fields INTERMSG, INTERMF1, INTERMF2, and INTERMF3.

INTRC = 01
The call has failed. Additional information is contained in fields INTERMSG, INTERMF1, INTERMF2, and INTERMF3.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
copy dslapims.
    03  filler              pic x(12280).
...
    move 'GETM' to intfunc
    move 'N' to intfrmid
    move spaces to intmsgid
    call dslapi using intwstor, msgswift
...
```

Here is an example in C/370:

```
...
#include "dslapc.h"
...
  struct INTWSTOR ws;
  struct {
     struct MSGSWIFT hdr;
     char buffer[12280];
     } ms;
...
  memcpy(ws.INTFUNC,"GETM",4);
  memcpy(ws.INTFRMID,"W",1);              /* SWIFT II format id */
  memcpy(ws.INTMSGID,"        ",8);       /* use internal msg.id */
  DSLAPI(&ws,&ms);
  if (memcmp(ws.INTRC,"  ",2) == 0) {
...
```

GETN Get Next Queue Element

The GETN function retrieves the next queue element with a QSN higher than the specified QSN and without an *in-service* flag from the MERVA ESA queue and puts it in the internal queue buffer.

The function flags the queue element *in-service* and sets the DOUBLE (write-back) indicator.

```
►►──GETN──(──INTWSTOR──)──────────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR must be set:

INTQUEUE
The name of the MERVA ESA queue or function containing the queue element to be retrieved.

INTQSN
The queue sequence number (QSN) of the queue element prior to the element to be retrieved.

Return Codes

INTRC = spaces
The call was successful. Additional information is contained in fields INTQSN, INTKEY1, INTKEY2, and INTDOUBL.

INTRC = 01
INTQUEUE is not defined. Additional information is contained in field INTERMSG.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

INTRC = 09
Either the INTQUEUE is empty or no queue element with a QSN higher than INTQSN and not flagged *in-service* exists.

Examples

Here is an example in C/370:

```
...
 #include "dslapc.h"
 struct INTWSTOR ws;
 char queue[2][8];
 int prevQSN;
...
 ws.INTQSN = prevQSN;
 memcpy(ws.INTFUNC,"GETN",4);
 memcpy(ws.INTQUEUE,queue[0],8);
 DSLAPI(&ws);
 if (memcmp(ws.INTRC,"  ",2) != 0) {
...
```

Here is an example in REXX:

```
...
intqueue = 'L1DE0'                    /* queue name               */
intqsn   = 0                          /* QSN - start at top       */
getn_rc  = ' '                        /* init GETN rc             */
Say 'Queue name :' intqueue
Do While getn_rc = ' '                /* loop while GETN rc = ' ' */
   Address DSLAPI "GETN"
   getn_rc = intrc                    /* save GETN rc             */
   If getn_rc = ' '
   Then Do
      Say ' '
      Say 'MERVA API command GETN was successful.'
      Say 'QSN ...... :' intqsn
      Say 'Key 1 .... :' intkey1
      Say 'Key 2 .... :' intkey2
      Address DSLAPI "FREE"           /* .. good practice         */
      If intrc ¬= ' ' Then Say 'But FREE failed with intrc' intrc'.'
   End
   Else Do
      Say ' '
      Say 'MERVA API command GETN failed with intrc' getn_rc'.'
   End
End
...
```

GETS Get SWIFT Message (MFS)

The GETS function maps a S.W.I.F.T message from MERVA internal format to an external format. The message in the API internal queue buffer is mapped through the internal TOF to the MSGSWIFT buffer.

The message is mapped into SWIFT I format. Specify INTFRMID='W' to have the message mapped according to the SWIFT II format.

The function reads user header fields from the TOF on nesting identifier 0 to the MSGSWIFT prefix. The MSGSWIFT prefix is described in Table 10 on page 89.

Note: You should prefer the MSGG function to the GETS function.

```
►►──GETS──(──INTWSTOR──,──MSGSWIFT──)──────────────────────────────────────►◄
```

INTWSTOR

API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameter in INTWSTOR is used:

INTFRMID

If set to 'W', the message is mapped into SWIFT II format. Otherwise SWIFT I format is used.

MSGSWIFT

A variable length buffer defined by the MSGSWIFT structure, copybook DSLAPIMS.

The buffer must be at least as big as the APISMSG specification in the MERVA ESA parameter module DSLPRM.

Return Codes

INTRC = spaces

The call was successful. Additional information is contained in the MSGSWIFT prefix fields MSGMTYPE, MSGNET, MSGDEST, MSGACK, and User Fields.

INTRC = 00

The MFS has detected checking errors. Additional information is contained in fields INTERMSG, INTERMF1, INTERMF2, and INTERMF3.

INTRC = 01

The call has failed. Additional information is contained in fields INTERMSG, INTERMF1, INTERMF2, and INTERMF3.

Examples

Here is an example in COBOL:

```
...
working-storage section.
copy dslapiws.
copy dslapims.
    03  msg-data            pic x(12280).
...
    move 'GETS' to intfunc
    move 'W' to intfrmid
    call dslapi using intwstor, msgswift
    if intrc not = spaces then
...
```

Here is an example in C/370:

```
...
  #include "dslapc.h"
  struct INTWSTOR ws;
  struct {
     struct MSGSWIFT hdr;
     char buffer[12280];
     } ms;
...
 ...
  memcpy(ws.INTFUNC,"GETS",4);
  memcpy(ws.INTFRMID,"W",1);       /* SWIFT II format id */
  DSLAPI(&ws,&ms);
  if (memcmp(ws.INTRC,"  ",2) != 0) {
 ...
```

GETU Get Next Queue Element Unconditionally

The GETU function retrieves the next queue element with a QSN higher than the specified QSN from the MERVA ESA queue and puts it in the internal queue buffer.

The function is unconditional because it retrieves a queue element regardless of its *in-service* status. Therefore the function should only be used for read-only retrieval.

The GETU function also ignores the hold status of a queue. That means the function retrieves queue elements, even if the queue is in hold status.

```
►►──GETU──(──INTWSTOR──)──────────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR must be set:

INTQUEUE
The name of the MERVA ESA queue or function containing the queue element.

INTQSN
The queue sequence number (QSN) of the queue element prior to the element to be retrieved.

Return Codes

INTRC = spaces
The call was successful. Additional information is contained in fields INTQSN, INTKEY1, INTKEY2, INTBUSY, and INTDOUBL.

INTRC = 01
INTQUEUE is not defined. Additional information is contained in field INTERMSG.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

INTRC = 09
Either INTQUEUE is empty or no queue element with a QSN higher than INTQSN exists.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
77  dslapi                 pic x(8) value 'DSLAPI'.
...
    move current-qsn to intqsn
    move 'L1PR1' to intqueue
    move 'getu' to intfunc
    call dslapi using intwstor
    if intrc = spaces then
      move intqsn to current-qsn
  ...
```

Here is an example in REXX:

```
...
intqueue = 'L1PR1'                      /* queue name              */
intqsn   = current_qsn                  /* queue sequence number   */
Address DSLAPI "GETU"
If intrc = ' '
Then
   current_qsn = intqsn                 /* save QSN                */
Else
   ...
```

INIT Initialize the API Environment

The INIT function initializes the operation of the DSLAPI interface including:

- INTWSTOR
- The Message Format Service
- Nucleus Intertask/Interregion Communication (NIC)
- Internal TOF and queue buffers.

```
►►──INIT──(──INTWSTOR──)──────────────────────────────────────────────────►◄
```

INTWSTOR API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameter in INTWSTOR is used:

INTCWA Before calling the INIT function, this address must be set to zero, except for the following special case: Your application is linked to the MERVA ESA nucleus, in which case INTCWA must be set to the address of the service communication area DSLCOM (contents of register 12).

When using the calling interface DSLAPCIC under CICS you do not need to set INTCWA to zero, it is set for you by DSLAPCIC.

Return Codes

INTRC = spaces
DSLAPI has initialized successfully. Additional information is contained in field INTSHUTD (INACTIVE or SHUTDOWN).

Note: If INTSHUTD contains the value INACTIVE, then an API program will not be able to use MERVA ESA central services: queue management, journal, user file, and command execution services. These functions will be rejected with return code 02.

INTRC = 02
DSLAPI is not initialized. Additional information is contained in field INTERMSG.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
77  dslapi                pic x(8) value 'DSLAPI'.
...
    move 'INIT' to intfunc
    set intcwa to null
    call dslapi using intwstor
...
```

Here is an example in C/370:

```
...
 #include "dslapc.h"
 struct INTWSTOR ws;
...
 memcpy(ws.INTFUNC,"INIT",4);
 ws.INTCWA = NULL;
 DSLAPI(&ws);
 if (memcmp(ws.INTRC,"  ",2) != 0) {
   print_error_codes(&ws);
...
```

Here is a PL/I example:

```
...
%include dslapiws;
...
allocate intwstor;
INTFUNC = 'INIT';
unspec(INTCWA) = 0;
call dslapi (intwstor);
if INTRC ¬= '  ' then do;
  call error_handling;
...
```

JRLG Get a Journal Record

The JRLG function reads the journal record having a key equal to the specified key from the MERVA ESA journal and puts it in the specified buffer.

If there is no record with the specified key, the record with the next higher key, if any, is returned. An incomplete, or generic, key is thus supported.

This function can be used instead of JRNG and supports journal records of any length.

```
►►──JRLG──(──INTWSTOR──,──JRNKEY──,──buffer──,──length──)──────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

JRNKEY
The key of the journal record to be read. The JRNKEY structure is defined by the copybook DSLAPIJK. The JRNRID field of the key is ignored.

If a record is returned, its key including the JRNRID field is placed in JRNKEY, otherwise JRNKEY is not changed.

buffer A storage area with no buffer prefix. The retrieved journal record is returned in this buffer. A truncated record is returned if the buffer is too small.

length A 31-bit binary value containing the length of the buffer. On return, this field contains the length of the journal record found. If the journal record is larger than the buffer, the required length is returned in *length*. You should allocate a larger buffer and repeat the JRLG call.

If no record was found, the value is set to zero.

Usage Notes

- If you do not know the size of a journal record, specify a length of zero. The required size is returned in the *length* parameter.
- If you only want the first part of a journal record, you do not need to supply a buffer large enough for the complete record.

Return Codes

INTRC = spaces
The call was successful.

INTRC = 02
The call failed. Additional information is contained in INTERMSG and INTSHUTD.

INTRC = 03
A record was found but it is larger than *length*. The necessary buffer size, that is, the length of the found record, is returned in *length*. The *buffer* contains the truncated journal record.

INTRC = 09
No record was returned. Either the journal is empty, or it contains no record with a key equal to or greater than JRNKEY.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
copy dslapijk.
77  jrnrcord                pic x(100).
77  jrn-size                pic s9(8) binary.
procedure division.
...
    move '19990301' to jrnkdat2
    move '0800'     to jrnktim2
    move 'JRLG'     to intfunc
    move length of jrnrcord to jrn-size
    call 'dslapi' using intwstor, jrn2key, jrnrcord, jrn-size
    if intrc = '03' then
      display 'journal record was truncated'
...
```

Here is an example in REXX:

```
...
jrnkdate = '19990301'                 /* journal date and time   */
jrnktime = '0800'
jrnkuser = ' '                        /* be tidy                 */
Address DSLAPI "JRLG"

If intrc = ' '
Then Do
   Say 'MERVA API command JRLG was successful.'
   Say 'journal id     :' "'"C2x(jrnrid)"'x"
   Say 'journal date   :' jrnkdate
   Say 'journal time   :' jrnktime
   Say 'user extension :' jrnkuser
   Say 'journal data   :' Strip(jrnrcord,'T')
End
Else
   Say 'MERVA API command JRLG failed with intrc' intrc'.'
...
```

JRLN Get Next Journal Record

The JRLN function reads the journal record having the next higher key than JRNKEY from the MERVA ESA journal and puts it in the specified buffer. Use this function to read the journal sequentially.

This function can be used instead of JRNN and supports journal records of any length.

```
►►—JRLN—(—INTWSTOR—,—JRNKEY—,—buffer—,—length—)——————————————————————►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

JRNKEY
The key of a journal record. The journal record with the next higher key is read. The JRNKEY structure is defined by the copybook DSLAPIJK. The JRNRID field of the key is ignored.

If a record is returned, its key including the JRNRID field is placed in JRNKEY, otherwise JRNKEY is not changed.

buffer A storage area of up to 2MB with no buffer prefix. The journal record is returned in this buffer. A truncated record is returned if the buffer is too small.

length A 31-bit binary value containing the length of the buffer. On return, this field contains the length of the returned journal record. If no record was found, the value is set to zero. If a record was found but it is longer than the buffer, its length is returned in *length*. You should allocate a larger buffer and repeat the JRLN call.

Usage Notes

If you only want the first part of a journal record, you do not need to supply a buffer large enough for the complete record.

Return Codes

INTRC = spaces
The call was successful.

INTRC = 02
The call failed. Additional information is contained in INTERMSG and INTSHUTD.

INTRC = 03
A record was found but it is larger than *length*. The necessary buffer size, that is, the length of the found record, is returned in *length*. The *buffer* contains the truncated journal record.

INTRC = 09
No record was returned. Either the journal is empty, or it contains no record with a key greater than JRNKEY. The *length* parameter is set to 0.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
copy dslapijk.
77  jrn-length            pic s9(8) binary.
77  jrn-data              pic x(4000).
...
    move 'jrlg' to intfunc
    call dslapi using intwstor jrn2key jrn-data jrn-length
    perform until intrc not = spaces or jrnkdat2 not = start-date
      display 'read journal record ' ...
      move 'jrln' to intfunc
      call dslapi using intwstor jrn2key jrn-data jrn-length
    end-perform
...
```

Here is an example in REXX:

```
...
jrnkdate = '19990404'               /* journal date and time    */
jrnktime = '1200'
jrnkseg  = ' '                      /* be tidy                  */
jrnksegs = ' '                      /*    "                     */
jrnkuser = ' '                      /*    "                     */

jrln_rc  = ' '                      /* init JRLN rc             */
Do While jrln_rc = ' '              /* loop while JRLN rc = ' ' */
   Address DSLAPI "JRLN"
   jrln_rc = intrc                  /* save JRLN rc             */
   If jrln_rc = ' '
   Then Do
      Say ' '
      Say 'MERVA API command JRLN was successful.'
      Say 'journal id .......... :' "'"C2x(jrnrid)"'x"
      Say 'journal date ........ :' jrnkdate
      Say 'journal time ........ :' jrnktime
      Say 'segment number ...... :' jrnkseg
      Say 'segment count ....... :' jrnksegs
      Say 'user extension ...... :' jrnkuser
      Say 'length of data ...... :' Length(jrnrcord)
      Say 'journal data  ....... :' Strip(jrnrcord,'T')
   End
End
...
```

JRLP Put a Journal Record

The JRLP function adds the journal record from the specified buffer, in the specified length, to the MERVA ESA journal. The journal record identifier (JRNRID) and user-key extension (JRNKUSER) from the JRNKEY structure are combined with the system date and time to form the journal key.

On return, the generated key is returned in JRNKEY.

This function can be used instead of JRNP and supports journal records of any length.

```
►►—JRLP—(—INTWSTOR—,—JRNKEY—,—buffer—,—length—)——————————————————————►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

JRNKEY
The key of the journal record to be written. The JRNKEY structure is defined by the copybook DSLAPIJK. Only the JRNRID and JRNKUSER fields are used, the JRNKDATE and JRNKTIME fields are filled by the system.

If the record is successfully written, its complete key including the generated date and time is placed in JRNKEY, otherwise JRNKEY is not changed.

buffer A storage area of up to the size specified in the MAXBUF parameter of the MERVA ESA parameter module, DSLPRM, with no buffer prefix, containing the journal record excluding the key.

length A 31-bit binary value containing the length of the journal record.

Usage Notes

If you have not defined segmented journal records in the JRNBUF parameter of your MERVA ESA parameter module DSLPRM, then the largest record you can write is the size of the journal cluster minus 54.

Return Codes

INTRC = spaces
The call was successful.

INTRC = 02
The call failed. Additional information is contained in INTERMSG and INTSHUTD.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
copy dslapijk.
77  jrn-length             pic s9(8) binary.
77  jrn-data               pic x(4000).
77  journal-record         pic x(100).
...
    move x'77' to jrnrid
    move 'a special journal key' to jrnkuser
    move journal-record to jrn-data
    move length of journal-record to jrn-length
    move 'jrlp' to intfunc
    call dslapi using intwstor jrnkey jrn-data jrn-length
...
```

Here is an example in REXX:

```
...
jrnrid   = '77'x
jrnkuser = ' a special journal key'   /* first byte should be ' ' */
jrnrcord = 'a privat journal record'
Address DSLAPI "JRLP"
If intrc = ' '
Then
   ...
```

JRNG Get a Journal Record

The JRNG function reads the journal record with a key equal to or greater than the specified key from the MERVA ESA journal and puts it in the specified buffer. The buffer must be smaller than 32KB (32,768 bytes).

Note: You should prefer the JRLG function to the JRNG function.

```
►►—JRNG—(—INTWSTOR—,—JRNKEY—,—buffer—)——————————————————————————►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

JRNKEY
The key of the journal record to be read. The JRNKEY structure is defined by the copybook DSLAPIJK. The JRNRID field of the key is ignored.

If a record is returned, its key, including the JRNRID field, is placed in JRNKEY, otherwise JRNKEY is not changed.

buffer
A buffer of up to 32KB containing a MERVA buffer prefix. You must set the buffer-size field in the buffer prefix.

Return Codes

INTRC = spaces
The call was successful. The key of the retrieved record is in parameter JRNKEY.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

INTRC = 09
Either the journal is empty or no journal record with a key equal to or greater than JRNKEY exists.

Examples

Here is an example in PL/I:

```
...
dcl dslapi entry options(assembler);
%include dslapiws;
%include dslapijk;
dcl 1 jrnrcord,
      2 pfx, %include dslapibp;
      2 data  char(100);
...
jrnkdate = '970301';
jrnktime = '1433';
jrnrcord.pfx.bufsize = length(jrnrcord.data)+8;
intfunc = 'JRNG';
call dslapi (intwstor,jrnkey,jrnrcord);
if intrc ¬= '  ' then do;
...
```

JRNN Get Next Journal Record

The JRNN function reads the journal record with a key greater than the specified key from the MERVA ESA journal and puts it in the specified buffer. The buffer must be smaller than 32KB. This function can be used to read the journal data set sequentially.

Note: You should prefer the JRLN function to the JRNN function.

```
►►──JRNN──(──INTWSTOR──,──JRNKEY──,──buffer──)──────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

JRNKEY
The key of the journal record to be read. The JRNKEY structure is defined by the copybook DSLAPIJK. The JRNRID field of the key is ignored.

If a record is returned, its key, including the JRNRID field, is placed in JRNKEY, otherwise JRNKEY is not changed.

buffer A buffer of up to 32KB containing a MERVA buffer prefix. You must set the buffer-size field in the buffer prefix.

Return Codes

INTRC = spaces
The call was successful. The key of the retrieved record is in parameter JRNKEY.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

INTRC = 09
Either the journal is empty or no journal record with a key greater than JRNKEY exists.

Examples

Here is a PL/I example:

```
...
dcl dslapi entry options(assembler,inter);
%include dslapiws;
%include dslapijk;
dcl ws like intwstor automatic;
dcl jk like jrnkey automatic;
dcl 1 jrnbuffer,
    %include dslapibp;
    2 jrndata char(100);
...
jrnkdate = DATE();
jrnktime = '0800';
bufsize = storage(jrnbuffer);
ws.intfunc='jrnn';
call dslapi(ws,jk,jrnbuffer);
...
```

JRNP Put a Journal Record

The JRNP function adds a record with the specified journal key record identifier (JRNRID) and user-key extension (JRNKUSER), from the record buffer and puts it in the MERVA ESA journal.

Note: You should prefer the JRLP function to the JRNP function.

```
►►—JRNP—(—INTWSTOR—,—JRNKEY—,—buffer—)————————————————————————►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

JRNKEY
The key of the journal record to be written. The JRNKEY structure is defined by the copybook DSLAPIJK. You must set the JRNRID field and the JRNKUSER field. The date and time fields are generated by MERVA ESA.

If a record is returned, its key, including the JRNRID field, is placed in JRNKEY, otherwise JRNKEY is not changed.

buffer A buffer of up to 32KB containing a MERVA buffer prefix. You must set the buffer-size field in the buffer prefix.

Note: Do not set the data-size field, DSLAPI determines the length by searching the buffer for the last non-blank character. So you should pad the buffer with blanks.

Return Codes

INTRC = spaces
The call was successful. The key of the retrieved record is in parameter JRNKEY.

INTRC = 01
The journal record buffer is empty. Additional information is contained in field INTERMSG.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
copy dslapijk.
01  jrnbuf.
    copy dslapibp.
    03 jrn-data         pic x(5000).
...
    move x'77' to jrnrid
    move 'a special journal record' to jrnkuser
    move special-buffer to jrn-data
    move length of jrnbuf to bufsize of jrnbuf
    move 'jrnp' to intfunc
    call dslapi using intwstor jrnkey jrnbuf
...
```

MPFG Get Message Prefix (MFS)

The MPFG function extracts the MERVA MSGSWIFT prefix structure from the message in the API internal queue buffer. This structure, MSGSWIFT_PREFIX, is defined by the DSLAPIMP copybook.

Only the following fields in the structure are obtained:

MSGMTYPE	Message form: I/O/F
MSGDBS	User field
MSGUSER1	User field
MSGADDR1..4	User field
MSGNET	Message identifier
MSGDST	S.W.I.F.T master destination (S.W.I.F.T only)
MSGACK	Message acknowledgment

All other fields in the structure are undefined.

```
►►──MPFG──(──INTWSTOR──,──MSGSWIFT_PREFIX──)─────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

MSGSWIFT_PREFIX
A storage area defined by the MSGSWIFT_PREFIX structure, copybook DSLAPIMP. The message prefix fields are returned in this buffer.

Return Codes

INTRC = spaces
The call was successful.

INTRC = 01
The call has failed. Additional information is contained in fields INTERMSG, INTERMF1, INTERMF2, and INTERMF3.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
01  msg-prefix.
    copy dslapimp.
...
    move qsn to intqsn
    move 'L3do0' to intqueue
    move 'get' to intfunc
    call dslapi using intwstor
...
    move 'mpfg' to intfunc
    call dslapi using intwstor msg-prefix
    if intrc = spaces then
       display 'message type is ' msgnet
...
```

Here is an example in REXX:

```
...
/* GETN and FREE a queue element */
intqueue = 'L1DE0'                   /* queue name              */
intqsn   = 0                         /* queue sequence number   */
Address DSLAPI "GETN"                /* .. sets actual QSN      */
If intrc ¬= ' ' Then ...
Address DSLAPI "FREE"
If intrc ¬= ' ' Then ...

Address DSLAPI "MPFG"
If intrc = ' '
Then Do
   Say 'MERVA API command MPFG was successful.'
   Say 'Message acknowledgment . (MSGACK)   :' msgack
   Say 'User field ............ (MSGADDR1) :' msgaddr1
   Say 'User field ............ (MSGADDR2) :' msgaddr2
   Say 'User field ............ (MSGADDR3) :' msgaddr3
   Say 'User field ............ (MSGADDR4) :' msgaddr4
   Say 'User field ............ (MSGDBS)   :' msgdbs
   Say 'SWIFT master destination (MSGDST)   :' msgdst
   Say 'Message form (I,O, or F) (MSGMTYPE) :' msgmtype
   Say 'Message identifier ..... (MSGNET)   :' msgnet
   Say 'User field ............ (MSGUSER1) :' msguser1
End
...
```

MPFP Put Message Prefix (MFS)

The MPFP function moves the MERVA MSGSWIFT prefix structure into the message in the API internal queue buffer. This structure, MSGSWIFT_PREFIX, is defined by the DSLAPIMP copybook.

Only the following fields in the structure are moved:

MSGDBS	User field
MSGUSER1	User field
MSGADDR1..4	User field
MSGACK	Message acknowledgment

All other fields in the structure are ignored.

```
►►──MPFP──(──INTWSTOR──,──MSGSWIFT_PREFIX──)─────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

MSGSWIFT_PREFIX
A storage area defined by the MSGSWIFT_PREFIX structure, copybook DSLAPIMP, containing the message prefix fields.

Return Codes

INTRC = spaces
The call was successful.

INTRC = 01
The call has failed. Additional information is contained in fields INTERMSG, INTERMF1, INTERMF2, and INTERMF3.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
01  msg-prefix.
    copy dslapimp.
...
    move 'some user data' to msguser1
    move 'mpfp' to intfunc
    call dslapi using intwstor msg-prefix
...
    move 'put' to intfunc
    call dslapi using intwstor
...
```

Here is an example in REXX:

```
...
/* GETN and FREE a queue element */
intqueue = 'L1DE0'                     /* queue name              */
intqsn   = 0                           /* queue sequence number   */
Address DSLAPI "GETN"                  /* .. sets actual QSN      */
If intrc ¬= ' ' Then ...
Address DSLAPI "FREE"
If intrc ¬= ' ' Then ...

msguser1 = 'Some user data'
msgack   = ''                          /* init all other DSLAPIMP */
msgaddr1 = ''                          /* fields! Otherwise REXX  */
msgaddr2 = ''                          /* would assume the field  */
msgaddr3 = ''                          /* name in uppercase as    */
msgaddr4 = ''                          /* value, e.g. msgaddr1 =  */
msgdbs   = ''                          /* 'MSGADDR1' ..           */
Address DSLAPI "MPFP"

Select
   When intrc = ' '
      Then Nop
   When intrc = '00'
      Then Do
         Say 'MERVA API command MPFP ended with intrc 00.'
         Say 'MFS has detected checking errors.'
      End
   Otherwise
      Say 'MERVA API command MPFP failed with intrc' intrc'.'
End /* -Select */

/* .. PUT it back */
intqueue = 'L2DE0'                     /* target queue name       */
Address DSLAPI "PUT"
...
```

MSGG Get Message (MFS)

The MSGG function maps (transforms) the message currently in the API internal queue buffer to the buffer you supply using the specified message identifier and device format code.

This is like the GETM and GETS function but can be used for messages of any size. The MSGSWIFT prefix is not returned. To obtain the fields in the prefix use the MPFG service.

```
►►──MSGG──(──INTWSTOR──,──buffer──,──length──)─────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR must be set:

INTMSGID
The message identifier of the message to be formatted. A message identifier identifies a MERVA ESA MCB via the message type table. If INTMSGID is blank, the actual message identification of the message is used.

INTFRMID
The format code in the INTMSGID MCB that is to be used to format the message. If INTFRMID is blank, or the format code does not exist in the MCB, the first device format in the MCB is used. If INTFRMID is set to Q, the output buffer will contain a message in internal buffer format. In this case an MCB is not used.

buffer A storage area, without a buffer prefix, into which the message will be stored.

length A 31-bit binary value containing the length of the buffer. On return, this field contains the length of the returned message. If the buffer is too small to contain the message, its length is returned in *length* and *buffer* contains the truncated message.

Usage Notes

- You would normally leave INTMSGID blank to avoid the risk of specifying an incorrect message type and consequently getting an incompletely mapped message.
- You would normally specify the format code INTFRMID. The default, the first device format in the MCB, may not be what you want. To obtain a SWIFT message in SWIFT II format set INTFRMID to ‘W’.

Return Codes

INTRC = spaces
The call was successful.

INTRC = 00
The MFS has detected checking errors. The *buffer* contains the message. Additional information is contained in fields INTERMSG, INTERMF1, INTERMF2, and INTERMF3.

INTRC = 01
The call has failed. Additional information is contained in fields INTERMSG, INTERMF1, INTERMF2, and INTERMF3.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG, INTERMF1, INTERMF2, and INTERMF3.

INTRC = 03
The specified buffer length *length* is too small. The length needed is returned in *length*. The *buffer* contains the truncated message.

Examples

Here is an example in COBOL:

```
working storage section.
copy dslapiws.
01  msg-buffer              pic x(5000).
77  buffer-size             pic s9(8) binary.
...
    move 'L3do0' to intqueue
    move 'getn' to intfunc
    call dslapi using intwstor
...
    move length of msg-buffer to buffer-size
    move 'W' to intfrmid
    move 'msgg' to intfunc
    call dslapi using intwstor msg-buffer buffer-size
    if intrc = spaces then
      display msg-buffer(1:buffer-size)
```

Here is an example in REXX:

```
/* GETN and FREE a queue element */
intqueue = 'L1DE0'                    /* queue name              */
intqsn   = 0                          /* queue sequence number   */
Address DSLAPI "GETN"                 /* .. sets actual QSN      */
If intrc ¬= ' ' Then ...
Address DSLAPI "FREE"
If intrc ¬= ' ' Then ...

/* Map the message from internal buffer to external format */
intmsgid = ' '                        /* default message type    */
intfrmid = 'W'                        /* SWIFT II format         */
Address DSLAPI "MSGG"
If intrc = ' '
Then Do
   Say ' '
   Say 'MERVA API command MSGG was successful.'
   parse_me = msgsmsg
   Say 'Message in external format:'
   Say Copies('-',70)
   Do While parse_me ¬== ''           /* print in chunks of 70   */
      Parse Var parse_me /* With */ 1 chunk 71 parse_me
      Say chunk
   End
   Say Copies('-',70)
End
Else
   Say 'MERVA API command MSGG failed with intrc' intrc'.'
```

MSGP Put Message (MFS)

The MSGP function maps (transforms) the message in your buffer to the API internal queue buffer using the specified message identifier and device format code.

This is like the PUTM and PUTS function but can handle messages of any size. The MSGSWIFT prefix is not part of the buffer. To add MSGSWIFT prefix fields to the API internal buffer use the MPFP service.

```
►►──MSGP──(──INTWSTOR──,──buffer──,──length──)──────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR must be set:

INTMSGID
The message identifier of the message to be formatted. A message identifier identifies a MERVA ESA MCB via the message type table.

If INTMSGID is blank, the message type determination exit is used to determine the message type. If the message type cannot be determined, the default message type 0DSL is used.

INTFRMID
The format code in the INTMSGID MCB that is to be used to format the message.

If INTFRMID is blank, the message type determination exit might set the format code. If it does not, or the specified format code does not exist in the MCB, the first device format in the MCB is used. The MERVA ESA message type determination exit can recognize SWIFT I and SWIFT II messages, Telex messages, and supported financial EDIFACT message types.

If INTFRMID is set to Q, the input buffer must contain a message in internal buffer format. In this case an MCB is not used.

buffer A storage area, without a buffer prefix, containing the message to be moved into the MERVA system.

length A 31-bit binary value containing the length of the message.

Usage Notes

- If you are sure of the type of message, you should specify INTMSGID. If you are not sure, then it is safer to set INTMSGID to blanks and let the message type determination exit determine the message.
- You would normally specify the format code INTFRMID. The default, the first device format in the MCB, may not be what you want.
- To map a SWIFT II message set INTFRMID to 'W', to map a SWIFT I message set it to 'S'.

Return Codes

INTRC = spaces
The call was successful.

INTRC = 00
The MFS has detected checking errors. Additional information is contained in fields INTERMSG, INTERMF1, INTERMF2, and INTERMF3.

INTRC = 01
The call has failed. Additional information is contained in fields INTERMSG, INTERMF1, INTERMF2, and INTERMF3.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG, INTERMF1, INTERMF2, and INTERMF3.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
77  msg-data                pic x(5000).
77  msg-size                pic s9(8) binary.
...
    move 'MSGP' to intfunc
    move spaces to intfrmid
    move spaces to intmsgid
    call dslapi using intwstor msg-data msg-size
    if intrc not = spaces then
...
```

Here is an example in REXX:

```
...
crlf = '0D25'x                     /* carriage return / line feed */
swbh = '{1:F01BANACCLLXBRA0000000000}'  /* S.W.I.F.T basic header*/
swah = '{2:I100BANBCCLLXBRAN}'          /* S.W.I.F.T applic hdr  */
msgh = '{4:'crlf                        /* bank message header   */
bmsg = ':20:TRN1234'crlf || ,           /* bank message          */
       ':32A:970606USD12,34'crlf || ,
       ':50:Anna Ameise'crlf'Aachen'crlf || ,
       ':59:/12345678'crlf'Berta Baer'crlf'Berlin'crlf
msgt = '-}'                             /* bank message trailer  */
msgx = swbh || swah || msgh || bmsg || msgt

intfrmid = ' '                          /* let MERVA determine   */
intmsgid = ' '                          /*    the message type   */
msgsmsg  = msgx                         /* the MERVA message     */
Address DSLAPI "MSGP"

Select
   When intrc = ' '
      Then Do                           /* PUT it to MERVA       */
         intqueue = 'L1DE0'             /* target queue name     */
         intqsn   = ' '                 /* clear output variable */
         Address DSLAPI "PUT"
         If intrc ¬= ' ' Then Say ...
      End
   When intrc = '00'
      Then Do
         Say 'MERVA API command MSGP ended with intrc 00.'
         Say 'MFS has detected checking errors.'
      End
   Otherwise
      Say 'MERVA API command MSGP failed with intrc' intrc'.'
End /* -Select */
...
```

Note: The same result could have been obtained explicitly giving the message type and format instead of using Message Type Determination:

```
...
    move 'W'    to intfrmid
    move 'S100' to intmsgid
...
```

or

```
...
    intfrmid = 'W'
    intmsgid = 'S100'
...
```

PRTI Initialize Printing Environment

The PRTI function initializes the print environment. This function is designed to be used together with PRTL and PRTT to format the message in the internal buffer, line by line, for the system printer device.

Syntax

```
►►──PRTI──(INTWSTOR)─────────────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameter in INTWSTOR is used:

INTQUEUE
The field TUCNAME is initially filled with this value.

INTFRMID
The format identifier to be used to format the message. Each device description in an MCB has a format identifier. When printing in prompt mode this identifier represents a language; when printing in noprompt mode this identifier represents a line format. If INTFRMID is blank, the default values are used. For printing in prompt mode, the default is E; for printing in noprompt mode, the default line format is Y.

Usage Notes

The formatting for the system printer uses environment information stored in a TUCB. This information can be inspected and modified using the FLDG and FLDP functions.

The following fields are especially important for formatting for printer devices:

TUCBCOMP
The compression format used for printing:

0	No compression. Empty fields and blank lines are printed.
1	Field compression. Only fields with data are printed (PROMPT UNIT mode).
2	Line compression. Empty data areas are not printed (PROMPT LINE mode).
3	Field and line compression. Only fields with data are printed (PROMPT UNIT LINE mode).
4	The message is printed in NOPROMPT format.

The default is 0. For more details see the *MERVA for ESA Macro Reference*, DSLFNT macro, parameter PRFORM.

TUCBLID
The language identification used for the selection of the system printer device description in the MCB. This identification is used when printing in prompt format (the field TUCBCOMP contains a value between 0 and 3). The default is E.

TUCBNID
: The line format identification used for the selection of the network device description in the MCB. This identification is used when printing in noprompt format (the field TUCBCOMP contains the value 4). The default is Y.

TUCBROWN
: The number of lines for each page. The default is 55.

TUCFRAMB
: Identifies the MCB for the bottom frame on each printed page. A blank indicates that no bottom frame is to be used, 0BOT indicates that the standard MERVA ESA bottom frame (2 blank lines) is to be used.

TUCFRAMT
: Identifies the MCB for the top frame on each printed page. A blank indicates that no top frame is to be used, 0TOP indicates that the standard MERVA ESA top frame is to be used.

TUCMSGID
: Identifies the MCB used for printing the message. Usually it contains the message identification or the name of the cover MCB. The default value is '0COV '.

TUCNAME
: The function name printed on the top of each page when the DSL0TOP top frame MCB is used.

Return Codes

INTRC = spaces
: DSLAPI has initialized a message for printing successfully.

INTRC = 01
: DSLAPI could not initialize the message for printing. Additional information is contained in fields INTERMSG and INTERMF1.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
77  dslapi                 pic x(8) value 'DSLAPI'.
...
    move 'PRTI' to intfunc
    move 'E' to intfrmid
    call dslapi using intwstor
...
```

PRTL Create a Print Line of a Message

The PRTL function formats the next line to be printed. The calling application may choose to actually print the result line returned in the buffer.

A PRTI call should be issued before the first PRTL call.

Syntax

```
►►──PRTL──(INTWSTOR,prtline)──────────────────────────────────────────►◄
```

INTWSTOR	API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.
prtline	A 133-byte fixed-length buffer. DSLAPI places the print line into this buffer. The first character is an ASA control character: The character ‘1’ in the first column indicates a request for a page eject; otherwise a blank indicating normal single line spacing. Empty (blank) lines are not suppressed but are returned as output by the PRTL function.

Usage Notes

The processing of a message consists of a PRTI call, followed by a sequence of PRTL calls until INTRC=09 is received. This indicates that all lines of the message have been returned. The print lines returned by each PRTL call can be printed by the application or stored in a file.

After one message has been completely formatted, the next message may be retrieved using the API queue management get functions. An optional PRTI call can be used to initialize the printing environment for the next message. Then a sequence of PRTL calls will format the message from the internal TOF buffer.

A PRTT call should follow as the last call after a print sequence.

The API INIT, REEN, or TERM calls are not allowed within a sequence of print calls.

No other API queue management calls should be used within a sequence of PRTL statements for a single message.

Return Codes

INTRC = spaces
The call was successful.

INTRC = 01
The call has failed. Additional information is contained in fields INTERMSG and INTERMF1.

INTRC = 09
End of the printout reached, no more print lines for the message are available.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
01  prtline.
    02 ctrl-char            pic x(1).
    02 prt-data             pic x(132).
...
    move 'PRTL' to intfunc
    call dslapi using intwstor, prtline
    if intrc = spaces then
      perform
        display '  ' prt-data
      end-perform
...
```

Here is an example in REXX:

```
...
/* GET a message */
intqueue = 'L1DE0'                     /* queue name             */
intqsn   = 123                         /* queue sequence number   */
Address DSLAPI "GET"
If intrc ¬= ' ' Then ...

/* PRTL the message line by line */
intrc = ' '                            /* init PRTL rc            */
Do While intrc = ' '                   /* loop while PRTL rc = ' '*/
   Address DSLAPI "PRTL"
   If intrc = ' ' Then Say prtline
End
...
```

PRTT Terminate Printing Environment

The PRTT function terminates the formatting of messages for the printer. The resources used for creating the print lines are released. If this call is not used after a PRTI call has been used, the resources are automatically released with the API TERM call.

Syntax

```
►►──PRTT──(INTWSTOR)──────────────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

No parameters in INTWSTOR are used.

Return Codes

INTRC = spaces
DSLAPI has terminated the printing of a message.

INTRC = 01
DSLAPI could not terminate the printing of a message. Additional information is contained in field INTERMSG.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
77  dslapi                pic x(8) value 'DSLAPI'.
...
    move 'PRTT' to intfunc
    call dslapi using intwstor
...
```

PUT Put a Queue Element

The PUT function takes the queue element from the internal queue buffer and puts it in the specified MERVA ESA queue.

```
►►──PUT──(──INTWSTOR──)──────────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameter in INTWSTOR is used:

INTQUEUE
The name of the MERVA ESA queue to which the queue element is to be written.

Return Codes

INTRC = spaces
The call was successful. Additional information is contained in fields INTQSN, INTKEY1, and INTKEY2.

INTRC = 01
INTQUEUE is not defined. Additional information is contained in fields INTERMSG and INTSHUTD.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
...
77  queue-name            pic x(8).
...
    move 'MSGP' to intfunc
    move 'W' to intfrmid
    call dslapi using intwstor msg-data msg-size
...
    move 'PUT ' to intfunc
    move queue-name to intqueue
    call dslapi using intwstor
    if intrc = spaces then
      display 'The message has been placed in ' intqueue
              ' with QSN ' intqsn
...
```

Here is an example in C/370:

```
...
 #include "dslapc.h"
 struct INTWSTOR ws;
 char queue[8];
...
 memcpy(ws.INTFUNC,"PUT ",4);
 memcpy(ws.INTQUEUE,queue,8);
 DSLAPI(&ws);
 if (memcmp(ws.INTRC,"  ",2) != 0) {
...
```

Here is an example in REXX:

```
...
/* MSGP a message */
intfrmid = 'W'                        /* SWIFT II line format    */
intmsgid = 'S100'                     /* message identifier      */
msgsmsg  = '{1:F01BANACCLLXBRA00...'   /* the MERVA message      */
Address DSLAPI "MSGP"
If intrc ¬= ' ' Then ...

intqueue = queue_name                 /* target queue name       */
intqsn   = ' '                        /* clear output variable   */
Address DSLAPI "PUT"
If intrc = ' '
Then Do
   Say 'MERVA API command PUT was successful.'
   Say 'Queue name :' intqueue
   Say 'QSN ...... :' intqsn
End
Else
   Say 'MERVA API command PUT failed with intrc' intrc'.'
...
```

PUTB Put a Queue Element with Automatic Delete

The PUTB function takes the queue element from the internal queue buffer and puts it in the specified MERVA ESA queue. The function automatically deletes the queue element in the queue INTBQUE with QSN INTBQSN.

Use this function when moving a queue element from one queue to another. MERVA ESA ensures that the element will not be lost or duplicated in the event of a system restart.

```
►►──PUTB──(──INTWSTOR──)───────────────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR must be set:

INTQUEUE
The name of the MERVA ESA queue to which the queue element is to be written.

INTBQUE
The name of the MERVA ESA queue containing the queue element to be automatically deleted.

INTBQSN
The queue sequence number of the queue element in queue INTBQUE to be automatically deleted.

Return Codes

INTRC = spaces
The call was successful. Additional information is contained in fields INTQSN, INTKEY1, and INTKEY2.

INTRC = 01
INTQUEUE is not defined. Additional information is contained in fields INTERMSG and INTSHUTD.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
...
    move 'getn' to intfunc
    call dslapi using intwstor
...
    move intqueue to intbque
    move intqsn to intbqsn
    move 'L2de0' to intqueue
    move 'putb' to intfunc
    call dslapi using intwstor
    if intrc = spaces then
      display ' moved msg from ' intbque intbqsn
              ' to ' intqueue intqsn
...
```

Here is an example in C/370:

```
...
 #include "dslapc.h"
 struct INTWSTOR ws;
 char queue[2][8];
 int saveqsn;
...
 memcpy(ws.INTFUNC,"PUTB",4);
 memcpy(ws.INTQUEUE,queue[1],8);     /* target queue */
 memcpy(ws.INTBQUE,queue[0],8);      /* source queue */
 ws.INTBQSN = saveqsn;               /* qsn of source msg */
 DSLAPI(&ws);
 if (memcmp(ws.INTRC,"  ",2) != 0) {
...
```

Here is an example in REXX:

```
...
/* GETN a queue element */
intqueue = 'L1DE0'                      /* queue name              */
intqsn   = 0                            /* queue sequence number   */
Address DSLAPI "GETN"
If intrc = ' '
Then Do
   savequeue = intqueue                 /* save queue name         */
   saveqsn   = intqsn                   /* save QSN                */
End
Else Do
   Say 'MERVA API command GETN failed with intrc' intrc'.'
   ...
End

/* and PUTB it */
intqueue = 'L2DE0'                      /* target queue name       */
intbque  = savequeue
intbqsn  = saveqsn
Address DSLAPI "PUTB"
If intrc = ' '
Then Do
   Say 'MERVA API command PUTB was successful.'
   Say 'From queue name :' savequeue
   Say 'From QSN        :' saveqsn
   Say 'To queue name   :' intqueue
   Say 'To QSN          :' intqsn
End
Else Do
   Say ' '
   Say 'MERVA API command PUTB failed with intrc' intrc'.'
   Say 'INTERMSG:' Strip(intermsg)
   Say 'INTSHUTD:' Strip(intshutd)
End
...
```

PUTM Put Message (MFS)

The PUTM function maps a message from external format to the MERVA internal format. The message in the buffer MSGSWIFT is mapped through the internal TOF to the internal queue buffer using the message type identifier specified in the INTWSTOR field INTMSGID, and the format identifier specified in INTFRMID. The message in the message buffer follows the rules of the specified message type. You must set the buffer size and data size values in the MSGSMSG structure.

The function writes the additional user header fields from the MSGSWIFT prefix to the TOF on nesting identifier 0. The MSGSWIFT prefix is described in Table 10 on page 89.

Note: You should prefer the MSGP function instead of the PUTM function.

```
►►──PUTM──(──INTWSTOR──,──MSGSWIFT──)──────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR are used:

INTMSGID
The message identifier of the message to be formatted. A message identifier identifies a MERVA ESA MCB via the message type table.

If INTMSGID is blank, the message type determination exit is used to determine the message type. If the message type cannot be determined the default message type 0DSL is used.

INTFRMID
INTFRMID identifies the line format. If INTFRMID is blank, the first line format in the MCB is used unless the message type determination exit can determine the format. The MERVA ESA message type determination exit can recognize SWIFT I and SWIFT II messages, Telex messages, and supported financial EDIFACT message types.

MSGSWIFT
A variable-length buffer defined by the MSGSWIFT structure, copybook DSLAPIMS. You must set the buffer size and data size values in the MSGSMSG structure.

Return Codes

INTRC = spaces
The call was successful.

INTRC = 00
MFS has detected checking errors. Additional information is contained in fields INTERMSG, INTERMF1, INTERMF2, and INTERMF3.

The call was successful.

INTRC = 01
The call has failed. Additional information is contained in fields INTERMSG, INTERMF1, INTERMF2, and INTERMF3.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
copy dslapims.
       03 msgdata           pic x(12280).
...
    add 4 to i giving datasize in msgsmsg
    move 'PUTM' to intfunc
    move 's999' to intmsgid
    move 'W' to intfrmid
    call dslapi using intwstor msgswift
    if intrc not = spaces then
...
```

Note: Because a SWIFT message is being imported the same result could have been obtained using Message Type Determination:

```
...
    move spaces to intmsgid
    move spaces to intfrmid
...
```

Here is an example in C/370:

```
...
  #include "dslapc.h"
  struct INTWSTOR ws;
  struct {
     struct MSGSWIFT hdr;
     char buffer[12280];
     } ms;
  int read_input_message(char *);
...
  if ((ms.hdr.prefix.datasize =
      read_input_message(ms.buffer)) == 0) {
...
  }
...
  ms.hdr.prefix.datasize = ms.hdr.prefix.datasize +4;
  memcpy(ws.INTFRMID,"W",1);          /* SWIFT II format id */
  memcpy(ws.INTMSGID,"        ",8);   /* MERVA determines msg.type */
  memcpy(ws.INTFUNC,"PUTM",4);
  DSLAPI(&ws,&ms);
  if (memcmp(ws.INTRC,"  ",2) != 0) {
...
```

PUTR Restore a Queue Element

The PUTR function takes the queue element from the internal queue buffer and puts it in the specified MERVA ESA queue with the specified QSN. Both key fields and the DOUBLE indicator can be set. This function allows a queue element to be restored to the same state that it had in an earlier get operation.

Syntax

```
►►──PUTR──(INTWSTOR)──────────────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR must be set:

INTQUEUE
The name of the MERVA ESA queue to which the queue element is to be written.

INTQSN
The queue sequence number of the queue element in queue INTQUEUE.

INTKEY1
The first key value of the queue element in queue INTQUEUE. If the queue is not defined with KEY1, the parameter is ignored.

INTKEY2
The second key value of the queue element in queue INTQUEUE. If the queue is not defined with KEY2, the parameter is ignored.

INTDOUBL
Either blanks or the indicator DOUBLE to request that the double indicator is to be written.

Usage Notes

No queue element with the same or a higher QSN may exist in the specified queue.

Return Codes

INTRC = spaces
The call was successful.

INTRC = 01
INTQUEUE is not defined. Additional information is contained in field INTERMSG.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
...
    move queue_name to intqueue
    move sequence_number to intqsn
    move 'DOUBLE' to intdoubl
    move 'putr' to intfunc
    call dslapi using intwstor
    if intrc = spaces then
      display ' msg restored in ' intqueue
...
```

Here is an example in C/370:

```
...
 #include "dslapc.h"
 struct INTWSTOR ws;
 char queue[2][8];
 int saveqsn;
...
 memcpy(ws.INTFUNC,"PUTR",4);
 memcpy(ws.INTQUEUE,queue[1],8);     /* target queue */
 ws.INTQSN = saveqsn;                /* qsn of source msg */
 memcpy(ws.INTDOUBL,"DOUBLE",6);
 DSLAPI(&ws);
 if (memcmp(ws.INTRC,"  ",2) != 0) {
...
```

Here is an example in REXX:

```
...
/* GETN a queue element */
intqueue = 'L1DE0'                     /* queue name              */
intqsn   = 0                           /* queue sequence number   */
Address DSLAPI "GETN"
If intrc = ' '
Then Do
   savequeue = intqueue                /* save queue name         */
   saveqsn   = intqsn                  /* save QSN                */
   savekey1  = intkey1                 /* save key 1              */
   savekey2  = intkey2                 /* save key 2              */
   savedoubl = intdoubl                /* save double indicator   */
   Address DSLAPI "DELE"               /* delete the message      */
End
Else Do
   Say 'MERVA API command GETN failed with intrc' intrc'.'
   ...
End

/* and PUTR it */
intqueue = savequeue                   /* queue name              */
intqsn   = saveqsn
intkey1  = savekey1
intkey2  = savekey2
intdoubl = savedoubl
Address DSLAPI "PUTR"
If intrc = ' '
Then Do
   Say 'MERVA API command PUTR successful, message restored.'
   Say 'To queue name   :' intqueue
   Say 'To QSN          :' intqsn
End
Else Do
   Say 'MERVA API command PUTR failed with intrc' intrc'.'
   Say 'INTERMSG:' Strip(intermsg)
   Say 'INTSHUTD:' Strip(intshutd)
End
...
```

PUTS Put SWIFT Message (MFS)

The PUTS function maps a S.W.I.F.T message from external to MERVA internal format. The S.W.I.F.T message in the buffer MSGSWIFT is mapped through the internal TOF to the internal queue buffer. You must set the buffer size and data size values in the MSGSMSG structure.

The type of the S.W.I.F.T message and its format, SWIFT I or SWIFT II, is determined by inspecting the message.

The function writes the additional user header fields from the MSGSWIFT prefix to the TOF on nesting identifier 0. The MSGSWIFT prefix is described in Table 10 on page 89.

Note: You should prefer the MSGP function instead of the PUTS function.

```
►►──PUTS──(──INTWSTOR──,──MSGSWIFT──)────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

MSGSWIFT
A variable-length buffer defined by the MSGSWIFT structure, copybook DSLAPIMS. You must set the buffer size and data size values in the MSGSMSG structure.

Return Codes

INTRC = spaces
The call is successful.

INTRC = 00
The MFS has detected checking errors. Additional information is contained in fields INTERMSG, INTERMF1, INTERMF2, and INTERMF3.

INTRC = 01
The call has failed. Additional information is contained in fields INTERMSG, INTERMF1, INTERMF2, and INTERMF3.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
copy dslapims.
       03 msgdata           pic x(12280).
...
    move 'PUTS' to intfunc
    call dslapi using intwstor msgswift
    if intrc not = spaces then
...
```

Here is an example in C/370:

```
...
#include "dslapc.h"
...
  struct INTWSTOR ws;
  struct {
     struct MSGSWIFT hdr;
     char buffer[12280];
     } ms;
  int read_input_message(char *);
...
  if ((ms.hdr.prefix.datasize =
      read_input_message(ms.buffer)) == 0) {
...
  }
  ms.hdr.prefix.bufsize = sizeof ms;
  ms.hdr.prefix.datasize = ms.hdr.prefix.datasize +4;
  memcpy(ws.INTFUNC,"PUTS",4);
  DSLAPI(&ws,&ms);
  if (memcmp(ws.INTRC,"  ",2) != 0) {
...
```

QLF Queue List First

The QLF function sets up an internal list of references to queue elements selected by QSN, generic key, or both from a specified queue. The list is in ascending QSN sequence.

DSLAPI manages internally a cursor into the list. The cursor is set to the first list entry, and the QSN and keys of the referenced queue element are returned in INTWSTOR.

The QLF function is designed to be used together with the QLN (Queue List Next) function to process selected queue elements in ascending QSN sequence.

```
►►──QLF──(──INTWSTOR──)──────────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR must be set:

INTQUEUE
The name of the MERVA ESA queue from which elements are to be selected.

INTQSN
The queue sequence number (QSN) where the list is to start.

A QSN of zero indicates that all elements in the queue can be included in the list.

INTKEY1
If INTKEY1 is not blank, only queue elements with a first key matching this key are included in the list.

INTKEY2
If INTKEY2 is not blank, only queue elements with a second key matching this key are included in the list.

Both INTKEY1 and INTKEY2 can contain non-blank values in which case selected elements must satisfy both key specifications.

The keys can be generic, that is, they can contain wildcards:

'%' matches any single character

'*' matches any number of characters, including no characters.

Note: The list is not preserved by SAVE and SAVL. Following a REEN you must reissue the QLF call.

Return Codes

INTRC = spaces
: The call was successful. An internal list has been established. INTQSN contains the QSN of the queue element referenced by the first entry in the list. INTKEY1 contains the element's first key, or spaces if no first key is defined for the queue. INTKEY2 contains the element's second key, or spaces if no second key is defined for the queue. INTBUSY contains 'BUSY' if the element is in service.

INTRC = 01
: INTQUEUE is not defined. Additional information is contained in field INTERMSG.

INTRC = 02
: The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

INTRC = 09
: Either the queue is empty or no queue elements with a QSN equal to or higher than INTQSN and matching the specified keys INTKEY1 and INTKEY2 exist.

Examples

Here is an example in COBOL:

```
...
data division.
working-storage section.
copy dslapiws.
...
procedure division.
...
    move spaces to intrc
    move 0 to intqsn
    move 'l2de0' to intqueue
    move '%ABC*1' to intkey1
    move 'qlf' to intfunc
    perform until intrc not = spaces
      call 'DSLAPI' using intwstor
      if intrc = spaces then
        move 'getc' to intfunc
        call 'DSLAPI' using intwstor
        if intrc = spaces or intrc = '08' then
          if intrc = '08' then
            display intqueue ' ' intqsn ' is busy'
            move spaces to intrc
          else
            perform process-element
          end-if
          move 'qln' to intfunc
        end-if
      end-if
    end-perform
...
```

Here is an example in REXX:

The EXEC lists, in ascending QSN sequence, all queue elements of a MERVA ESA queue with their key values and the BUSY indicator:

```
 ...
 intqueue = 'L1DE0'                   /* queue name              */
 intqsn   = 0                         /* QSN - start at top      */
 intkey1  = ' '                       /* key 1                   */
 intkey2  = ' '                       /* key 2                   */
 Say 'Queue name :' intqueue
 Say 'Start QSN  :' intqsn

 Call Run_QLF
 If intrc = ' ' Then Call Run_QLN
 ...

 /* -- QLF -- */
Run_QLF:

 Address DSLAPI "QLF"
 If intrc = ' '
 Then Do
    Say 'QSN       '                ,
        'Key 1                   ' ,
        'Key 2                   ' ,
        'Busy'
    Say '----------'                ,
        '------------------------' ,
        '------------------------' ,
        '----'
    Say Right(intqsn,10) ,            /* first list item         */
        Left(intkey1,24) ,
        Left(intkey2,24) ,
        intbusy
 End

 Return

 /* -- QLN -- */
Run_QLN:

 intrc = ' '                          /* init QLN rc             */
 Do While intrc = ' '                 /* loop while QLN rc = ' '  */
    Address DSLAPI "QLN"
    If intrc = ' '
    Then Do
       Say Right(intqsn,10) ,         /* next list item          */
           Left(intkey1,24) ,
           Left(intkey2,24) ,
           intbusy
    End
 End

 Return
```

QLL Queue List Last

The QLL function sets up an internal list of references to queue elements selected by QSN, generic key, or both from a specified queue. The list is in ascending QSN sequence.

DSLAPI manages internally a cursor into the list. The cursor is set to the last list entry, and the QSN and keys of the referenced queue element are returned in INTWSTOR.

The QLL function is designed to be used together with the QLP (Queue List Previous) function to process selected queue elements in descending QSN sequence.

```
►►—QLL—(—INTWSTOR—)——————————————————————————————————————————►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR must be set:

INTQUEUE
The name of the MERVA ESA queue from which elements are to be selected.

INTQSN
The highest queue sequence number (QSN) to be included in the list. Only elements with a QSN less than or equal to this number will be selected.

A QSN of zero indicates that all elements in the queue can be included in the list.

INTKEY1
If INTKEY1 is not blank, only queue elements with a first key matching this key are included in the list.

INTKEY2
If INTKEY2 is not blank, only queue elements with a second key matching this key are included in the list.

Both INTKEY1 and INTKEY2 can contain non-blank values in which case selected elements must satisfy both key specifications.

The keys can be generic, that is, they can contain wildcards:

'%'	matches any single character
'*'	matches any number of characters, including no characters.

Note: The list is not preserved by SAVE and SAVL. Following a REEN you must reissue the QLL call.

Return Codes

INTRC = spaces
The call was successful. An internal list has been established. INTQSN contains the QSN of the queue element referenced by the last entry in the list. INTKEY1 contains the element's first key, or spaces if no first key is defined for the queue. INTKEY2 contains the element's second key, or spaces if no second key is defined for the queue. INTBUSY contains 'BUSY' if the element is in service.

INTRC = 01
INTQUEUE is not defined. Additional information is contained in field INTERMSG.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

INTRC = 09
Either the queue is empty or no queue elements with a QSN equal to or lower than INTQSN and matching the specified keys INTKEY1 and INTKEY2 exist.

QLN Queue List Next

The QLN function updates the cursor into the internal queue list to point to the next, succeeding, entry. The QSN and keys of the queue element referenced by this entry are returned in INTWSTOR.

It is an error to invoke QLN if an internal list has not previously been established by a QLF (or QLL) call.

The QLN function is designed to be used together with the QLF (Queue List First) function to process selected queue elements in ascending QSN sequence.

Note: You can cause an unintended program loop if, when processing elements using a list, you write them back to the same queue (using ROUB or PUT, for example). This effectively extends repeatedly the list which you are processing.

```
►►──QLN──(──INTWSTOR──)──────────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

No variables in INTWSTOR need to be set before invoking QLN.

Return Codes

INTRC = spaces
The call was successful. INTQSN contains the QSN of the queue element referenced by the next entry in the list. INTKEY1 contains the element's first key, or spaces if no first key is defined for the queue. INTKEY2 contains the element's second key, or spaces if no second key is defined for the queue. INTBUSY contains 'BUSY' if the element is in service.

INTRC = 01
No internal list has been established by a previous QLF or QLL call.

INTRC = 09
There are no more elements in the queue matching the specification used to establish the internal list.

QLP Queue List Previous

The QLP function updates the cursor into the internal queue list to point to the preceding entry. The QSN and keys of the queue element referenced by this entry are returned in INTWSTOR.

It is an error to invoke QLP if an internal list has not previously been established by a QLL (or QLF) call.

The QLP function is designed to be used together with the QLL (Queue List Last) function to process selected queue elements in descending QSN sequence.

```
►►──QLP──(──INTWSTOR──)────────────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

No variables in INTWSTOR need to be set before invoking QLP.

Return Codes

INTRC = spaces
The call was successful. INTQSN contains the QSN of the queue element referenced by the previous entry in the list. INTKEY1 contains the element's first key, or spaces if no first key is defined for the queue. INTKEY2 contains the element's second key, or spaces if no second key is defined for the queue. INTBUSY contains 'BUSY' if the element is in service.

INTRC = 01
No internal list has been established by a previous QLF or QLL call.

INTRC = 09
There are no more elements in the queue matching the specification used to establish the internal list.

READ Read a Field (TOF)

The READ function reads the field identified by the field reference from the internal queue buffer through the internal TOF into the specified buffer.

```
►►──READ──(──INTWSTOR──,──TOFPARM──,──buffer──)──────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

TOFPARM
The field reference of the field to be read.

buffer A buffer of up to 32KB containing a MERVA buffer prefix. You must set the buffer-size field in the buffer prefix.

Usage Notes

- After TOF access calls (like READ) the input field-reference data might have been changed by the TOF supervisor to its output parameters. Note especially that after an unsuccessful READ the returned TOFFDNAM value is unpredictable, that is, does not contain the failing TOFFDNAM parameter.
- See "TOF Access Parameters TOFPARM" on page 85 and the description of macro DSLTSV in the *MERVA for ESA Macro Reference* for more details about field references and request modifiers.

Return Codes

INTRC = spaces
The call was successful. The TOFPARM structure contains the current field reference and the MERVA ESA TOF supervisor return and reason codes, TOFTSVRC and TOFTSVRS.

Note: This return code only suggests that control was successfully passed to the TOF supervisor. You must also check TOFTSVRC and TOFTSVRS.

INTRC = 01
Mapping from the queue buffer to the TOF has failed. Additional information is contained in field INTERMSG.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
copy dslapitp.
...
77  todays-date            pic x(6).
01  tofbuf.
    copy dslapibp.
    03 tofdata pic x(100).
...
    move length of tofbuf to bufsize of tofbuf
    move 'READ' to intfunc
    move 'DSLDATE1' to toffdnam
    move spaces to tofmodif
    move 0 to toffdnl
    move 1 to toffdfg, toffdoc, toffdda
    call 'dslapi' using intwstor, tofparm, tofbuf
    if intrc = spaces then
       move tofdata(1:6) to todays-date
...
```

Here is an example in C/370:

```
...
 #include "dslapc.h"
 struct INTWSTOR ws;
 struct TOFPARM tofpl;
 struct { char userid[8];        /* MSGTRACE field structure */
          char function[8];
          char error_code[4];
          char date[6];
          char time[6];
 } msgtrace;
 struct {
   struct BufferPrefix pfx;      /* standard MERVA buffer prefix */
   char tofdata[80];
 } tofbuffer;
...
 memcpy(ws.INTFUNC,"READ",4);        /* read..                */
 memcpy(tofpl.TOFFDNAM,"DSLDATE1",8); /* ..date in YYMMDD form */
 tofpl.TOFFDNL = 0;
 tofpl.TOFFDFG = 1;
 tofpl.TOFFDOC = 1;
 tofpl.TOFFDDA = 1;
 memset(tofpl.TOFMODIF,' ',sizeof tofpl.TOFMODIF);
 DSLAPI(&ws,&tofpl,&tofbuffer);
 if (memcmp(ws.INTRC,"  ",2) == 0) {
   memcpy(msgtrace.date,&tofbuffer;tofdata,6);
...
```

Here is an example in REXX:

```
...
toffdnam = 'DSLDATE1'                /* date in YYMMDD format    */
toffdnl  = 0                         /* nesting level identifier */
toffdfg  = 1                         /* field group index        */
toffdoc  = 1                         /* rep. sequence index      */
toffdda  = 1                         /* data area index          */
tofmodif = ' '                       /* request modifier         */
tofdata  = ''                        /* clear output variable    */
Address DSLAPI "READ"

If intrc = ' ' & toftsvrc = 0 & toftsvrs = 0
Then Do
   Say 'MERVA API command READ was successful.'
   Say 'TOF field :' toffdnam
   Say 'TOF data  :' tofdata
End
Else
   Say 'MERVA API command READ failed with intrc' intrc',' ,
       'toftsvrc' toftsvrc', toftsvrs' toftsvrs'.'
...
```

REEN Reenter API Environment

The REEN function initializes the API interface using the API environment saved by a previous SAVE or SAVL call.

```
►►──REEN──(──INTWSTOR──,──buffer──)─────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameter in INTWSTOR must be set:

INTCWA
Set this address field to 0.

buffer A buffer without a MERVA buffer prefix containing the API environment created by a SAVE or SAVL call.

Return Codes

INTRC = spaces
DSLAPI has initialized successfully. Additional information is contained in field INTSHUTD (INACTIVE or SHUTDOWN).

Note: If INTSHUTD contains the value INACTIVE, then all queue management, journal, user file, and command execution functions will be rejected with return code 02.

INTRC = 02
DSLAPI is not reentered. Additional information is contained in field INTERMSG.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
...
    move low-values to intwstor
    move 'REEN' to intfunc
    call 'dslapi' using intwstor save-buffer
    if intrc not = spaces then
...
```

Here is an example in C/370:

```
...
 #include "dslapc.h"
 struct INTWSTOR ws;
 char *p;
...
 /* restart MERVA API with the saved API buffers */
 memcpy(ws.INTFUNC,"REEN",4);    /* re-initialize DSLAPI */
 DSLAPI(&ws,p);                  /* ..using saved buffers */
 if (memcmp(ws.INTRC,"  ",2) != 0) {
   print_error_codes(&ws);
   terminate_API(&ws);
   return(4);
 }
 free(p);                        /* release the buffer storage */
...
```

REPL Replace a Queue Element

The REPL function replaces the specified element in the MERVA ESA queue data set with the queue element in the internal queue buffer.

```
►►—REPL—(—INTWSTOR—)——————————————————————————————————————————————►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR must be set:

INTQUEUE
The name of the MERVA ESA queue or function containing the queue element to be replaced.

INTQSN
The queue sequence number (QSN) of the queue element to be replaced.

Usage Notes

- The QSN of the replaced element does not change.
- The *in-service* indicator is not reset.

Return Codes

INTRC = spaces
The call was successful. Additional information is contained in fields INTKEY1 and INTKEY2.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
copy dslapitp.
77  xyz-qsn               pic s9(8) binary.
77  xyz-queue             pic x(8).
...
    move xyz-qsn to intqsn
    move xyz-queue to intqueue
    move 'getn' to intfunc
    call dslapi using intwstor
...
    move xyz-qsn to intqsn
    move xyz-queue to intqueue
    move 'repl' to intfunc
    call dslapi using intwstor
    if intrc not = spaces then
...
```

Here is an example in REXX:

```
...
/* GETN a queue element */
intqueue = xyz_queue                   /* queue name              */
intqsn   = xyz_qsn                     /* queue sequence number   */
Address DSLAPI "GETN"
If intrc = ' '
Then
   saveqsn   = intqsn                  /* save actual QSN         */
Else Do
   Say 'MERVA API command GETN failed with intrc' intrc'.'
   ...
End

...

/* REPLace the queue element */
intqueue = xyz_queue                   /* queue name              */
intqsn   = saveqsn                     /* queue sequence number   */
Address DSLAPI "REPL"
If intrc ¬= ' ' Then ...

/* and FREE it */
intqueue = xyz_queue                   /* queue name              */
intqsn   = saveqsn                     /* queue sequence number   */
Address DSLAPI "FREE"
If intrc ¬= ' ' Then
...
```

ROU Route a Queue Element

The ROU function routes the queue element from the internal queue buffer to the queues selected by the routing table of the specified MERVA ESA function.

```
►►──ROU──(──INTWSTOR──)────────────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameter in INTWSTOR must be set:

INTQUEUE
The name of the MERVA ESA function whose routing table is to be used.

Return Codes

INTRC = spaces
The call was successful. Additional information is contained in fields INTQSN, INTKEY1, and INTKEY2.

INTRC = 01
Either the routing of INTQUEUE has failed or the selected queues have not been defined. Additional information is contained in fields INTERMSG and INTSHUTD.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
77  msg-data                pic x(5000).
77  msg-size                pic s9(8) binary.
...
    move 'MSGP' to intfunc
    move 'W' to intfrmid
    call dslapi using msg-data msg-size
...
    move 'ROU ' to intfunc
    move queue-name to intqueue
    call dslapi using intwstor
    if intrc not = spaces then
...
```

Here is an example in REXX:

```
...
/* GETN and FREE a queue element */
intqueue = 'L1DE0'                     /* queue name              */
intqsn   = 0                           /* queue sequence number   */
Address DSLAPI "GETN"                  /* .. sets actual QSN      */
If intrc ¬= ' ' Then ...
Address DSLAPI "FREE"
If intrc ¬= ' ' Then ...
...

/* set MSGOK field in TOF to 'YES' */
toffdnam = 'MSGOK'                     /* name of the field       */
toffdnl  = 0                           /* nesting level index     */
toffdfg  = 1                           /* field group index       */
toffdoc  = 1                           /* rep. sequence index     */
toffdda  = 1                           /* data area index         */
tofmodif = ' '                         /* request modifier        */
tofdata  = 'YES'
Address DSLAPI "WRIT"
If intrc = ' ' & toftsvrc = 0 & toftsvrs = 0
Then
   Nop                                 /* ok, continue            */
Else
   ...                                 /* WRIT failed             */

/* L1AI0 routes queue elements                     */
/* - with MSGOK field = 'YES' to queue L1RFINN      */
/* - with MSGOK field = 'NO'  to queue L1VE0.       */
/* (see function table and routing module DWSL1AI0) */
intqueue = 'L1AI0'
Address DSLAPI "ROU"
If intrc ¬= ' ' Then
...
```

ROUB Route a Queue Element with Automatic Delete

The ROUB function routes the queue element in the internal queue buffer to the queues selected by the routing table of the specified MERVA ESA function.

The queue element specified in the back reference INTBQSN and INTBQUE is simultaneously deleted.

Use this function when routing a queue element from one queue to another queue or queues. MERVA ESA ensures that the element will not be lost or duplicated in the event of a system restart.

```
►►──ROUB──(──INTWSTOR──)──────────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR must be set:

INTQUEUE
The name of the MERVA ESA function whose routing table is to be used.

INTBQUE
The name of the MERVA ESA queue containing the queue element to be automatically deleted.

INTBQSN
The queue sequence number of the queue element in queue INTBQUE to be automatically deleted.

Return Codes

INTRC = spaces
The call was successful. Additional information is contained in fields INTQSN, INTKEY1, and INTKEY2.

INTRC = 01
Either the routing of INTQUEUE has failed or the selected queues have not been defined. Additional information is contained in fields INTERMSG and INTSHUTD.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
...
    move 'L3do0' to intqueue
    move 'getn' to intfunc
    call dslapi using intwstor
...
    move intqueue to intbque
    move intqsn to intbqsn
    move 'L3do0' to intqueue
    move 'roub' to intfunc
    call dslapi using intwstor
    if intrc not = spaces then
...
```

Here is an example in REXX:

```
...
/* GETN a queue element */
intqueue = 'L1DE0'                    /* queue name              */
intqsn   = 0                          /* queue sequence number   */
Address DSLAPI "GETN"
If intrc = ' '
Then Do
   savequeue = intqueue               /* save queue name         */
   saveqsn   = intqsn                 /* save QSN                */
End
Else Do
   Say 'MERVA API command GETN failed with intrc' intrc'.'
   ...
End

/* ROUB to queue L1AI0, which uses routing module DWSL1AI0 */
intqueue = 'L1AI0'
intbque  = savequeue                  /* .. will be deleted      */
intbqsn  = saveqsn
Address DSLAPI "ROUB"
If intrc = ' '
Then Do
   Say ' '
   Say 'MERVA API command ROUB was successful.'
   Say 'From queue name :' savequeue
   Say 'From QSN        :' saveqsn
   Say 'To queue name   :' intqueue
   Say 'To QSN          :' intqsn
End
Else Do
   Say ' '
   Say 'MERVA API command ROUB failed with intrc' intrc'.'
End
...
```

ROUD Route Queue Element Directly

The ROUD function takes the queue element with the specified QSN from the MERVA ESA queue and routes it. The queue element is also put into the internal queue buffer.

The function is unconditional because it retrieves a queue element regardless of its *in-service* status.

The ROUD function also ignores the hold status of a queue. This means the function retrieves queue elements, even if the queue is in hold status.

Use this function when routing a queue element unchanged from one queue to another. MERVA ESA ensures that the element will not be lost or duplicated in the event of a system restart.

If the queue element is to be modified in the process, the GET and ROUB API calls should be used instead.

Syntax

```
►►──ROUD──(INTWSTOR)──────────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

The following parameters in INTWSTOR must be set:

INTQUEUE
The name of the MERVA ESA queue containing the queue element.

INTQSN
The queue sequence number (QSN) of the queue element to be routed.

Return Codes

INTRC = spaces
The call was successful. Additional information is contained in fields INTQSN, INTKEY1, and INTKEY2. The routed queue element is also in the internal queue buffer.

INTRC = 01
INTQUEUE is not defined. Additional information is contained in field INTERMSG.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
...
    move current-qsn to intqsn
    move 'L2AI0' to intqueue
    move 'roud' to intfunc
    call dslapi using intwstor
    if intrc = spaces then
      move intqsn to current-qsn
...
```

Here is an example in REXX:

```
...
intqueue = 'L2AI0'                     /* queue name              */
intqsn   = current_qsn                 /* queue sequence number   */
Address DSLAPI "ROUD"
If intrc = ' '
Then
   /* routing performed, queue element data can be read .. */
Else
   /* access of queue element failed, or routing failed    */
...
```

ROUN Route Next Queue Element Directly

The ROUN function takes the next queue element with a QSN higher than the specified QSN from the MERVA ESA queue and routes it. The queue element is also put into the internal queue buffer.

The function is unconditional because it retrieves a queue element regardless of its *in-service* status.

The ROUN function also ignores the hold status of a queue. This means the function retrieves queue elements, even if the queue is in hold status.

Use this function when routing a queue element unchanged from one queue to another. MERVA ESA ensures that the element will not be lost or duplicated in the event of a system restart.

When the queue element is to be modified in the process, the GETU and ROUB API calls should be used instead.

Syntax

```
►►──ROUN──(INTWSTOR)──────────────────────────────────────────────►◄
```

INTWSTOR
: API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

  The following parameters in INTWSTOR must be set:

  INTQUEUE
  : The name of the MERVA ESA queue containing the queue element.

  INTQSN
  : The queue sequence number (QSN) of the queue element prior to the element to be routed. Specify a QSN of zero to indicate the first element in the queue.

Return Codes

INTRC = spaces
: The call was successful. Additional information is contained in fields INTQSN, INTKEY1, and INTKEY2. The routed queue element is also in the internal queue buffer.

INTRC = 01
: INTQUEUE is not defined. Additional information is contained in field INTERMSG.

INTRC = 02
: The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
...
    move current-qsn to intqsn
    move 'L2AI0' to intqueue
    move 'roun' to intfunc
    call dslapi using intwstor
    if intrc = spaces then
      move intqsn to current-qsn
...
```

Here is an example in REXX:

```
...
intqueue = 'L2AI0'                      /* queue name             */
intqsn   = current_qsn                  /* queue sequence number  */
Address DSLAPI "ROUN"
If intrc = ' '
Then
   current_qsn = intqsn                 /* save QSN               */
Else
   ...
```

SAVE Save API Environment

The SAVE function copies the API internal buffers to a storage area you provide so that you can save the current API environment from one step of an online transaction dialog to the next.

When the next dialog step is started API should be initialized with this area using the REEN function instead of the INIT function.

```
►►──SAVE──(──INTWSTOR──,──buffer──)────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

buffer A buffer without a MERVA buffer prefix. The buffer must be large enough to contain the interface working storage and the internal buffers (TOF and queue).

The initial size of the internal buffers is determined by the specifications in the customization module DSLPRM. The maximum size that can occur is 4KB + 32KB + 32KB = 68KB.

Usage Notes

You should use the SAVL function instead of this function to be able to support dynamic internal TOF and queue buffer sizes. For an example refer to sample program DSLBA04 in “DSLBA04x” on page 208.

Return Codes

INTRC = spaces
DSLAPI has saved the working storage and the internal buffers successfully.

Examples

Here is an example in C/370:

```
...
 #include "dslapc.h"
 struct INTWSTOR ws;
 char *p;
...
 if (( p = (char *)malloc(1024*(4+32+32)) ) == NULL) {
   printf("...not enough storage\n\n");
   terminate_API(&ws);
   return(4);
 }
 memcpy(ws.INTFUNC,"SAVE",4);    /* copy internal buffers */
 DSLAPI(&ws,p);
 if (memcmp(ws.INTRC,"  ",2) != 0) {
...
```

SAVL Save API Environment

The SAVL function copies the API internal buffers to a storage area you provide so that you can save the current API environment from one step of an online transaction dialog to the next.

The size of the buffer you need is in the INTWSTOR structure, field INTSIZE. Note that this value can change after each API call.

When the next dialog step is started API should be initialized with this area using the REEN function instead of the INIT function.

This is the alternative for the large message environment to the SAVE function.

```
►►──SAVL──(──INTWSTOR──,──buffer──)──────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

buffer A storage area, without a buffer prefix, into which the API internal work areas are stored. The size of the buffer must be at least as large as the value in the INTWSTOR INTSIZE field.

Return Codes

INTRC = spaces
The call was successful.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
77  storage-id             pic s9(8) binary.
77  storage-address        pointer.
...
linkage section.
01  save-buffer            pic x.
...
    move 0 to storage-id
    call 'CEEGTST' using storage-id, intsize,
                         storage-address, fc
...
    set address of save-buffer to storage-address
    move 'SAVE' to intfunc
    call 'dslapi' using intwstor save-buffer
    if intrc not = spaces then
...
```

TERM Terminate API

The TERM function allows DSLAPI to terminate the API environment.

```
►►──TERM──(──INTWSTOR──)──────────────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

Return Codes

The TERM function does not issue any return codes.

Examples

Here is an example in COBOL:

```
       ...
       working storage section.
       copy dslapiws.
       ...
           move 'TERM' to intfunc
           call dslapi using intwstor
       ...
```

Here is a PL/I example:

```
   ...
   dcl dslapi entry options(assembler,inter);
   %include dslapiws;
   dcl ws like intwstor automatic;
   ...
   ws.intfunc='TERM';
   call dslapi (ws);
   ...
 END  TESTREAD;
```

USRG User File Record Get

The USRG function reads the user file record with the specified key from the MERVA ESA user file and puts it into the record buffer. The DSLAPI functions USRG and USRN can only be used if EXDSP=YES is specified in your DSLPRM parameter module.

```
►►──USRG──(──INTWSTOR──,──key──,──DSLUSRS──)──────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

key
An 8-byte field containing the user identification. DSLAPI folds the value to uppercase.

DSLUSRS
A buffer defined by the user file record structure, DSLUSRS. DSLUSRS is defined in the copybook DSLAPIUS.

Return Codes

INTRC = spaces
The call was successful. The user file record has been placed into the record buffer.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

INTRC = 09
There is no user file record with the requested key.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
copy dslapius.
77  user-id                pic x(8).
...
    move user-name to user-id
    move 'USRG' to intfunc
    call 'dslapi' using intwstor, user-id, dslusrs
    if intrc not = spaces then
...
```

Here is an example in C/370:

```
...
 #include "dslapc.h"
 struct INTWSTOR ws;
 struct DSLUSRS  ur;
 char userid[8];
...
     /* get the user record for the specified userid */
     memcpy(ws.INTFUNC,"USRG",4);
     DSLAPI(&ws,&userid,&ur);
     if (memcmp(ws.INTRC,"  ",2) == 0) {
       print_user_record(&ur);
     }
     else {
...
```

Here is an example in REXX:

```
...
USRKEY = 'MAS1'
Address DSLAPI "USRG"

If intrc = ' '
Then Do
   Say 'MERVA API command USRG was successful.'
   Say 'User Id   :' usrukey
   Say 'Name .... :' usruname
   Say 'Origin Id :' usruorid
End
Else
   Say 'MERVA API command USRG failed with intrc' intrc'.'
...
```

USRN User File Get Next

The USRN function reads the next user file record with a key greater than the specified key from the MERVA ESA user file and puts it into the record buffer. The key of the record read is returned in the key parameter. To get the first record of the user file specify a blank key. The DSLAPI functions USRG and USRN can only be used if EXDSP=YES is specified in your DSLPRM parameter module.

This function can be used to read the MERVA ESA user file sequentially.

```
►►──USRN──(──INTWSTOR──,──key──,──DSLUSRS──)──────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

key An 8-byte field containing the user identification. DSLAPI folds the value to uppercase. After each retrieval it contains the key of the record retrieved.

DSLUSRS
A buffer defined by the user file record structure, DSLUSRS. DSLUSRS is defined in the copybook DSLAPIUS.

Return Codes

INTRC = spaces
The call was successful. The user file record has been placed into the record buffer. Its key has been written into USRKEY.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

INTRC = 09
Either the user file is empty or no user file record with a key greater than USRKEY exists.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
copy dslapius.
77  user-id                pic x(8).
...
    move spaces to user-id
    move 'USRN' to intfunc
    call 'dslapi' using intwstor, user-id, dslusrs
    perform until intrc not = spaces
      perform print-user-record
      call 'dslapi' using intwstor, user-id, dslusrs
    end-perform
...
```

Here is an example in C/370:

```
...
 #include "dslapc.h"
 struct INTWSTOR ws;
 struct DSLUSRS  ur;
 char userid[8];
...
     /* get sequentially all user records */
     memcpy(ws.INTFUNC,"USRN",4);
     while (memcmp(ws.INTRC,"  ",2) == 0) {
       DSLAPI(&ws,userid,&ur);
       if (memcmp(ws.INTRC,"  ",2) == 0) {
         print_user_record(&ur);
       }
       else {
...
```

Here is an example in REXX:

```
...
usrkey  = ' '                         /* USRKEY - start at top    */
usrn_rc = ' '                         /* init USRN rc             */

Do While usrn_rc = ' '                /* loop while USRN rc = ' ' */
   Address DSLAPI "USRN"
   usrn_rc = intrc                    /* save USRN rc             */
   If usrn_rc = ' '
   Then Do

      Say ' '
      Say 'MERVA API command USRN was successful.'
      Say 'User Id ...... :' usrukey
      Say 'Name ......... :' usruname
      Say 'Origin Id .... :' usruorid

      ufd = 'User Functions :'
      If usruftab.0 = 0               /* no 'allowed functions'   */
      Then                            /* print at least head line */
         Say ufd
      Else Do
         Do i = 1 To usruftab.0 By 6
            funcline = '
            Do j = i To i + 5 While j <= usruftab.0
               funcline = funcline || Left(usruftab.j,8) || ' '
            End
            Say ufd funcline
            ufd = Copies(' ',Length(ufd))   /* clear description  */
         End
      End
      Drop usruftab.                  /* .. be tidy               */

   End

   Else Do
      Say ' '
      Say 'MERVA API command USRN failed with intrc' intrc'.'
   End
End
...
```

WRIT Write a Field (TOF)

The WRIT function moves data from the field buffer through the internal TOF to the specified field in the internal queue buffer.

```
►►──WRIT──(──INTWSTOR──,──TOFPARM──,──buffer──)──────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

TOFPARM
The field reference of the field to be written.

buffer A buffer of up to 32KB containing a MERVA buffer prefix. You must set the buffer-size field and the data length field in the buffer prefix.

Usage Notes

- If the data length field in the buffer prefix is 0, DSLAPI determines the data size by finding the last nonblank character.
- After TOF access calls (like WRIT) the input field-reference data might have been changed by the TOF supervisor to its output parameters. Note especially that after an unsuccessful WRIT the returned TOFFDNAM value is unpredictable, that is, does not contain the failing TOFFDNAM parameter.
- See “TOF Access Parameters TOFPARM” on page 85 and the description of macro DSLTSV in the *MERVA for ESA Macro Reference* for more details about field references and request modifiers.

Return Codes

INTRC = spaces
The call was successful. The TOFPARM structure contains the current field reference and the MERVA ESA TOF supervisor return and reason codes, TOFTSVRC and TOFTSVRS.

Note: This return code only suggests that control was successfully passed to the TOF supervisor. You must also check TOFTSVRC and TOFTSVRS.

INTRC = 01
The call has failed for one of the following reasons:
- An empty field buffer
- Failure to map from the internal queue buffer to the internal TOF, or from the internal TOF to the internal queue buffer.

Additional information is contained in field INTERMSG.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
copy dslapitp.
01  tofbuf.
    copy dslapibp.
    03 tofdata pic x(100).
...
    move 'WRIT' to intfunc
    move 'MSGTRACE' to toffdnam
    move 0 to toffdnl
    move 1 to toffdfg, toffdoc
    move 32767 to toffdda
    move spaces to tofmodif
    string 'dslba04b', 'myuserid', '0000', today, tod,
        delimited by size into tofdata
    end-string
    move 36 to datasize of tofbuf
    call 'dslapi' using intwstor, tofparm, tofbuf
...
```

Here is an example in C/370:

```
...
 #include "dslapc.h"
 struct TOFPARM tofpl;
 struct {
   struct BufferPrefix pfx;      /* standard MERVA buffer prefix */
   char tofdata[80];
 } tofbuffer;
 struct { char userid[8];        /* MSGTRACE field structure */
          char function[8];
          char error_code[4];
          char date[6];
          char time[6];
 } msgtrace;
...
 memcpy(tofbuffer.tofdata,&msgtrace,sizeof msgtrace);
 tofbuffer.pfx.datasize = sizeof msgtrace +4;
 memcpy(ws.INTFUNC,"WRIT",4);         /* TOF write..          */
 memcpy(tofpl.TOFFDNAM,"MSGTRACE",8); /* ..MSGTRACE field     */
 tofpl.TOFFDDA = 32767;               /* ..data area          */
 tofpl.TOFFDNL = 0;
 tofpl.TOFFDFG = 1;
 tofpl.TOFFDOC = 1;
 memset(tofpl.TOFMODIF,' ',sizeof tofpl.TOFMODIF);
 DSLAPI(&ws,&tofpl,&tofbuffer);
...
```

Here is an example in REXX:

```
...
pgm      = 'MYAPIPGM'                 /* 8 char. program name    */
queue    = Left(myqueue,8)            /* 8 char. queue name      */
mfsrc    = '0000'                     /* 4 char. MFS rc          */
date     = Substr(Date('s'),3)        /* 6 char. date YYMMDD     */
Parse Value Time('n') With hh ':' mm ':' ss
time     = hh || mm || ss             /* 6 char. time HHMMSS     */
lterm    = '        '                 /* 8 char. terminal name   */
tofdata  = pgm || queue || mfsrc || , /* data to be written      */
           date || time || lterm

toffdnam = 'MSGTRACE'                 /* name of the field       */
toffdnl  = 0                          /* nesting level index     */
toffdfg  = 1                          /* field group index       */
toffdoc  = 1                          /* rep. sequence index     */
toffdda  = 32767                      /* data area index         */
tofmodif = ' '                        /* request modifier        */
Address DSLAPI "WRIT"
If intrc = ' ' & toftsvrc = 0 & toftsvrs = 0
Then
   Nop                                /* ok                      */
Else
   ...                                /* WRIT failed             */
```

WTO Write to Operator

The WTO function adds an operator message to the system console and to the MERVA ESA display message table. The message is also written to the MERVA journal.

```
►►──WTO──(──INTWSTOR──,──buffer──)────────────────────────────────────►◄
```

INTWSTOR
API interface working storage. The INTWSTOR structure is defined by the copybook DSLAPIWS.

buffer A fixed-length buffer of 120 bytes, without a buffer prefix, containing the operator message.

Return Codes

INTRC = spaces
The call was successful.

INTRC = 01
The specified operator message is blank.

INTRC = 02
The call has failed. Additional information is contained in fields INTERMSG and INTSHUTD.

Examples

Here is an example in COBOL:

```
...
working storage section.
copy dslapiws.
77  opdata                 pic x(120).
...
    call dslapi using intwstor
    if intrc not = spaces then
      move spaces to opdata
      string 'PGM123: function ', intfunc, ', RC=', intrc,
          ' ' intbusy ' ' intshutd ' ' intermsg
          delimited by size into opdata
      move 'wto' to intfunc
      call dslapi using intwstor opdata
...
```

Here is a PL/I example:

```
...
dcl dslapi entry options(assembler);
%include dslapiws;
  ...
  dcl 1 opmsg,
        2 data       char(160);  /* only 1st 120 bytes are output */
  ...
  if intrc ¬= '  ' then do;
    put string (opmsg.data)  edit
      ('Prgname:', intfunc, 'rc =', intrc, intermsg) (5(a,x(1)));
    intfunc = 'wto ';
    call dslapi (intwstor, opmsg);
    return(8);
  end;
  ...
```

Here is an example in REXX:

```
...
wtomsg = 'MERVA WTO test message. Please ignore.'
Address DSLAPI "WTO"
If intrc = ' '
Then
   Say 'MERVA API command WTO was successful.'
Else Do
   Say 'MERVA API command WTO failed with intrc' intrc'.'
   Say 'INTERMSG:' Strip(intermsg)
   Say 'INTSHUTD:' Strip(intshutd)
End
...
```

Part 3. Appendixes

Appendix A. Migration and Compatibility

This appendix describes migration and compatibility aspects of the DSLAPI interface.

Differences between MERVA ESA Version 3 Release 3 and Version 4 Release 1

All applications using DSLAPI for MERVA ESA V3.3 should run unchanged with MERVA ESA V4.1.

To be consistent with all other API functions the API functions GET and GETC now return INTRC=01 instead of INTRC=02 when the specified INTQUEUE is not defined.

Under MVS the library with the MERVA ESA programs has a new name. Under CICS specify SDSLLODB and SDSLLODC, under IMS specify SDSLLODB and SDSLLODI, instead of SDSLLOD0.

For programs running in IMS environment, the IMS PCB list address must be written to field COMPCBLA using the API FLDP function. If this field is not set, IMS databases cannot be accessed from within MERVA ESA API functions.

Differences between MERVA ESA Version 3 Release 2 and Release 3

All applications using DSLAPI for MERVA ESA V3.2 should run unchanged with MERVA ESA V3.3.

The new journal key structure, JRN2KEY, in copybook DSLAPIJK, defines the journal key for a four-digit year format. The total length of the journal key is unchanged. If you are using a four-digit year in the journal, the REXX API functions JRLN and JRNNreturn an eight-digit date in the variable jrnkdate 'journal date'.

Differences between MERVA/ESA V3.1 and MERVA ESA Version 3 Release 2

For MERVA ESA running under VSE/ESA™ 1.3 or higher, DSLAPI can run in AMODE 31 and exploit storage above the 16MB line. Large message processing is supported. All applications using DSLAPI for MERVA/ESA V3.1 should run unchanged with MERVA ESA V3.2.

Differences between MERVA/370 Version 2 and MERVA ESA Version 3

MERVA ESA Version 3 includes the following changes:

- To exploit the MERVA ESA support for messages larger than 32KB new API functions must be used:

MERVA ESA V3 Function	Purpose	MERVA/370 V2 Function
MSGG	MFS get message	GETM, GETS
MSGP	MFS put message	PUTM, PUTS
MPFG	MFS get MSGSWIFT prefix	GETM, GETS
MPFP	MFS put MSGSWIFT prefix	PUTM, PUTS
JRLG	Get journal record	JRNG
JRLN	Get next journal record	JRNN
JRLP	Put journal record	JRNP
SAVL	Save DSLAPI environment	SAVE

- The High-level language copybooks have been revised:
  - Each structure is now in a separate copybook. This allows users to include only the structures they need.
  - Copybook names are the same in each language.
  - C/370 versions of the API structures have been defined in a C header-file.

  The MERVA/370 V2 copybooks DSLAPCBL and DSLAPPLI are unchanged.
- Segmented journal records. If JRNBUF=SEG has been specified in DSLPRM, and the MAXBUF value is greater than 32KB, a journal retrieval might return a segment of a journal record; the application must explicitly retrieve the separate segments.

  The new journal key structure, JRNKEY, in copybook DSLAPIJK, defines the segmented journal key. The nonsegmented form of the key is only defined in the MERVA/370 V2 copybooks, except for the Assembler copybook, which defines both forms.
- The MSGTRACE field is now variable length and a subfield for the terminal ID has been added.
- A Write-to-Operator function, WTO, has been added.

Appendix B. Sample Programs

A number of sample API programs are distributed with MERVA ESA. This appendix describes these programs.

All sample programs are named in a consistent fashion: *DSLBAnnx*, where 'nn' is the sample program number, and 'x' indicates the programming language:

A	Assembler
B	COBOL
C	C/370
P	PL/I
R	REXX.

Note: These programs are merely designed to demonstrate the use of MERVA ESA API services. They are not otherwise intended as realistic examples of MERVA applications.

For example, in "Chapter 1. Introduction and Concepts" on page 3 you are advised to use dynamic linkage to invoke DSLAPI. Because it is a little simpler, however, these programs use static linkage.

The sample programs are:

- Batch API programs. Each program exists in a COBOL, PL/I, C/370, and an Assembler version. DSLBA01x, DSLBA02x, and DSLBA03x also exist in a REXX version. DSLBA10R exists in REXX only.

DSLBA01x	MERVA command service
DSLBA02x	Export a SWIFT message from MERVA ESA
DSLBA03x	Import a SWIFT message into MERVA ESA
DSLBA04x	SAVE and REEN services
DSLBA05x	User file services
DSLBA10R	Update a queue element.

- A transaction for automatic start. This program, too, exists in each of the supported languages, except Assembler and REXX.

DSLBA06x	Queue Management services.

- A conversational transaction. Programs that together implement a CICS dialog. Each program is in COBOL and PL/I.

DSLBA20x	Dialog control (does not use the API)
DSLBA21x	Data entry
DSLBA22x	Data verification
DSLBA23x	Queue inspection
DSLBA24x	S.W.I.F.T message map
DSLBA25x	Function selection map
DSLBA26x	Data entry map

DSLBA27x	Data verification map.

- An MFS HLL exit for the CICS and batch environment. There is a COBOL, PL/I, and C version.

DSLBA30x	HLL version of the DSLMS911 sample field separation exit.

The following sample programs are reproduced in this Appendix:

DSLBA03B	COBOL batch program
DSLBA06P	PL/I transaction for automatic start
DSLBA30C	C/370 MFS exit.

Refer to the distribution libraries for the other programs. Sample programs are distributed in the MERVA ESA samples library, MERVA.SDSLSAM0. In VSE they are distributed in the source sublibrary.

Batch Programs

The MERVA ESA sample library contains six sample batch programs that illustrate most API services.

The samples run in MVS or VSE batch. The API functions are independent of the operating system and can be used in both environments in the same way.

For Assembler samples running under VSE, however, the MVS LOAD macro must be replaced by a VSE CDLOAD macro or by loading a V-address constant.

DSLBA01x

The sample program DSLBA01x executes a MERVA ESA operator command and prints the command response to the standard output file. The command to be executed can be input via the job-step parameter. Alternatively, a default command is executed.

The command response buffer contains 10 lines of output. If a commands response is longer than 10 lines, the command must be submitted again to obtain the next group of 10 lines. This must be repeated until all lines of the response have been returned.

Submitting the command again after the last lines of the response have been returned causes the first lines of a new response to be returned. The program saves the first response buffer and then compares it with subsequent response buffers to determine when the response is complete. But note that this is not a completely reliable way of recognizing the end of a response; the status of the system may have changed so that the first lines are different.

DSLBA02x

The sample program DSLBA02x moves a message from one queue to another using the automatic delete facility (PUTB). The queue names are given in the PARM parameter on the EXEC statement of the JCL.

The message is also mapped (MSGG) into both SWIFT I and SWIFT II net format. An application program, after mapping a message into an external format like this, would normally write the record to a file or data base. This is not done by the sample program, it merely demonstrates how to get a message into a buffer in an external format.

DSLBA03x

The sample program DSLBA03x is a simple example of importing a message into MERVA ESA.

A message in SWIFT II format is read from the standard input, the CRLF (carriage-return, line-feed) code is appended as necessary, and then the message string is mapped into MERVA ESA internal format using the MFS service MSGP.

Then the message is written to a queue using the queue management PUT service. This appends a message to a queue. The QSN assigned to the message is returned by the PUT and displayed in a message to the standard output before the program terminates.

The name of the queue to which the SWIFT message is to be written is input via the PARM parameter on the EXEC statement.

The COBOL version of this sample is shown in Figure 8.

```
 identification division.
 program-id.    dslba03b.
*******************************************************************
*
* Function    : MERVA ESA sample API program
*         This program shows how to use the DSLAPI
*            MFS   - Message Format service, and
*            QMG   - Queue Management service
*         to create a S.W.I.F.T message and put it into a queue.
*         The program is executed in MVS batch and requires a
*         queue name to be specified in the EXEC PARM.
*
*         A SWIFT II message is read from SYSIN (trailing ';'
*         are replaced by CRLF (X'0D25')), input to MERVA, and
*         written to the queue.
*         Example:
*         //API EXEC PGM=DSLBA03B,PARM='L2DE0'
*         //SYSIN DD       *
*         {1:F01TIBMDEPPAXXX0000000000}
*         {2:I999TIBMDEPPAXXXN}
*         {3:{108:DSLBA03B S1}}
*         {4:;
*         :20:DSLBA03B S1;
*         :21:MT S999;
*         :79:EXAMPLE OF DSLAPI MSGP FUNCTION;
*             THIS IS SWIFT II FORMAT;
*         -}
*         /*
*
* Dependencies: MERVA ESA must be active
*
* Environment : MVS batch, VSE batch
*
******************************************************************/

 date-compiled.
 data division.
 working storage section.
 copy dslapiws.
 77  msgdata                 pic x(10000).
 77  msgdata-size            pic s9(8) binary.
 77  queue-name              pic x(8).
 77  dslapi                  pic x(8) value 'DSLAPI'.
 77  i                       pic s9(4) binary.
 77  j                       pic s9(4) binary.
 77  input-line              pic x(80).
 77  opdata                  pic x(100).

 linkage section.
 01  parmdata.
     02  parm-size           pic s9(4) binary.
     02  parm-string         pic x(8).
  procedure division using parmdata.
     move 0 to return-code
```

Figure 8. DSLBA03B COBOL Sample Batch API Program (Part 1 of 3)

```
  *    read the queue name from the EXEC PARM
      if parm-size > 0 then
        move parm-string(1:parm-size) to queue-name
      else
        display 'Supply queue name in EXEC PARM'
        move 8 to return-code
        goback
      end-if
 *    read in the S.W.I.F.T message
 *    (i is an index into the message buffer)
      move 1 to i
      perform read-input-message
      if i = 1 then
        display 'Supply input message in SYSIN'
        move 8 to return-code
        goback
      end-if
      move i to msgdata-size

  *   initialize the MERVA API
      move 'INIT' to intfunc
      set intcwa to null
      call dslapi using intwstor
      if intrc not = spaces then
        perform write-error-message
        move 8 to return-code
        goback
      end-if

  *   move the message to the API internal queue buffer
      move 'MSGP' to intfunc
      move 'W' to intfrmid
      call dslapi using intwstor msgdata msgdata-size
      if intrc not = spaces then
        perform write-error-message
        perform terminate-api
        move 8 to return-code
        goback
      end-if

  *   move the message to the target queue
      move 'PUT ' to intfunc
      move queue-name to intqueue
      call dslapi using intwstor
      if intrc not = spaces then
        perform write-error-message
        perform terminate-api
        move 8 to return-code
        goback
      end-if

      display 'The message has been placed in ' intqueue
          ' with QSN ' intqsn
      display 'finished'
      perform terminate-api
      goback
      .
```

Figure 8. DSLBA03B COBOL Sample Batch API Program (Part 2 of 3)

```
 terminate-api.
    move 'TERM' to intfunc
    call dslapi using intwstor
    .

write-error-message.
    move spaces to opdata
    string 'dslba03b: function ', intfunc, ', RC=', intrc,
        ' ' intbusy ' ' intshutd ' ' intermsg
        delimited by size into opdata
    display opdata
    .
read-input-message.
    move spaces to input-line
    accept input-line
    perform until input-line = spaces
      perform varying j from length of input-line
                        by -1
                        until input-line (j:1) not = space
      end-perform
      if input-line (j:1) = ';' then
         move X'0D25' to input-line (j:2)
         add 1 to j
      end-if
      move input-line (1:j) to msgdata (i:j)
      add j to i
      move spaces to input-line
      accept input-line
    end-perform
    subtract 1 from i
    .

end program dslba03b.
```

Figure 8. DSLBA03B COBOL Sample Batch API Program (Part 3 of 3)

DSLBA04x

The sample program DSLBA04x shows the use of the SAVE and REEN functions. These functions can be used in conversational applications, where processing is suspended for indefinite time intervals, to save the API environment. This is discussed in “Saving the Tokenized Form of a Message in Your Own Database (IMS) or in Temporary Storage or Transient Data (CICS)” on page 28.

The program reads a message from a MERVA ESA queue using GETN. This flags the queue element *in-service* to indicate that it should not be updated by another user. Then the API environment, including the internal queue and TOF buffers, is saved with the SAVE function and DSLAPI terminated (TERM). (The COBOL program uses a *Language Environment/370* service to obtain dynamic storage for the SAVE buffer.)

After DSLAPI is reinitialized with the reenter function (REEN), restoring the saved environment and the internal buffers, the messages MSGTRACE field is updated with a new entry (data area). Note that the message is not reread from the queue; following the SAVE and REEN the message is still in the internal queue buffer.

The date and time for the MSGTRACE entry are read from MERVA, demonstrating the reading of MERVA ESA system fields. Normally, you would use a high-level language function to get the date or time.

The MSGTRACE data area is written using the TOF WRIT function. Setting the data area index to a high value ensures that the data area is appended as the last data area in the field. Since MERVA does not allow gaps in the data area index sequence, the data area is given the next highest unused index, not the index you specify.

Then the message is moved to a second MERVA ESA queue data set, using the PUTB function to simultaneously delete the message from the first queue.

A confirmation message is written to the standard output before the API is normally terminated with the TERM function.

Note that, if an error occurs after the GETN, the queue FREE service is used to clear the *in-service* indicator. If this were not done, the queue element would remain flagged until MERVA ESA termination.

DSLBA05x

The sample program DSLBA05x shows the use of the USRG and USRN functions. The program prints the contents of the user file record, the key of which is given in the PARM parameter on the EXEC statement. The user identification, the user name, the origin identification, and the allowed functions for this user are printed to the standard output. If the parameter is omitted, all user file records are printed.

See also batch utility DSLBA15R in “Appendix C. Batch Utilities in REXX” on page 227.

DSLBA10R - Update a Queue Element

DSLBA10R reads a queue element, changes the value of the SW108 field in the TOF, adds a new MSGTRACE, and replaces the message in the queue. It then reads the message again and prints the new values to show that the fields were changed. Dependencies: MERVA ESA must be active.

Job Control Statements

The following figure shows the MVS JCL to run DSLBA10R.

```
//.......  JOB  ...........
//REXXB    EXEC PGM=DSLAREXX,REGION=8M,
//              PARM='DSLBA10R,parm1 parm2 parm3'
//*
//*          .. MERVA ESA LOAD LIBRARY
//STEPLIB  DD DSN=loadlib,DISP=SHR
//*
//*          .. ON THIS PDS: DSLBA10R
//SYSEXEC  DD DSN=samplib,DISP=SHR
//*
//*          .. LISTING DATASET (VB136)
//SYSTSPRT DD DSN=listds,DISP=OLD
//
```

Figure 9. MVS JCL to Run Sample Program DSLBA10R - Update a Queue Element

In the JCL, the lowercase parameters have the following meanings:

loadlib	The name of the load library containing the MERVA ESA programs.
samplib	The name of the library containing the program DSLBA10R.
listds	The name of the listing data set. Must be preallocated, record format VB, logical record length 136 recommended.

Runtime Parameters

The following parameters can be specified in the PARM field of the EXEC statement:

parm1	Queue name.
parm2	Queue sequence number (QSN). If not specified, the first queue element found is updated.
parm3	Log level. From **1** (basic) to **4** (all). The default is 2.

Sample printout

The following figure shows the information printed after the execution of the DSLBA10R program.

```
MERVA ESA V4.1 DSLBA10R                      11. Oct. 1999  13:26:35


DDDDD    SSSSSS   LLL      BBBBB     AAAA      111    0000    RRRRR
DDDDDD   SSSSSS   LLL      BBBBBB   AAAAAA    1111   000000   RRRRRR
DD  DD   SS       LLL      BB  BB   AA  AA   11111   00  00   RR  RR
DD  DD   SS       LLL      BB  BB   AA  AA     111   00  00   RR  RR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA     111   00  00   RRRRRR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA     111   00  00   RRRRRR
DD  DD       SS   LLL      BB  BB   AA  AA     111   00  00   RR  RR
DD  DD       SS   LLL      BB  BB   AA  AA     111   00  00   RR  RR
DDDDDD   SSSSSS   LLLLLL   BBBBBB   AA  AA     111   000000   RR  RR
DDDDD    SSSSSS   LLLLLL   BBBBB    AA  AA     111    0000    RR  RR


Read a queue element, add a new MSGTRACE, change field SW108 (User Ref.),
and write it back to its queue.


DSLBA10R_001I : Runtime parameter 'Queue name' .......... : L1DE0

DSLBA10R_002I : Runtime parameter 'Queue sequence number' : 0

DSLBA10R_003I : Runtime parameter 'Log level' ........... : 1
                Allowed log levels are: 1 .. 4, and '*'.

DSLBA10R_005I : GETN of the following queue element was successful:
                - INTQUEUE : L1DE0
                - INTQSN   : 40
                - INTKEY1  : T5
                - INTKEY2  :

DSLBA10R_007I : MSGTRACE data areas 'BEFORE'
                1: MAS1    L1DE0   0000960507172408A105
                2: MASDBCS L1DE0   0000960509143922D804

DSLBA10R_008I : SW108 'BEFORE' : My old reference

DSLBA10R_013I : MERVA API command REPL was successful.
                Queue name : L1DE0
                QSN ...... : 40

DSLBA10R_014I : MERVA API command GET was successful.
                Queue name : L1DE0
                QSN ...... : 40
                Key 1 .... : T5
                Key 2 .... :

DSLBA10R_007I : MSGTRACE data areas 'AFTER'
                1: MAS1    L1DE0   0000960507172408A105
                2: MASDBCS L1DE0   0000960509143922D804
                3: DSLBA10RL1DE0   0000971011132635

DSLBA10R_008I : SW108 'AFTER' : A new reference

DSLBA10R_015I : DSLBA10R ended with return code 0 - successful.
                Total processing time was 0.42 seconds.
```

Figure 10. Printout of the DSLBA10R Sample Program

Sample Transaction for Automatic Start

Sample program DSLBA06x is an example CICS background transaction that is automatically started by MERVA ESA when a message is moved to the queue with which the transaction is associated. A transaction is associated with a MERVA queue, or function, by the TRAN= parameter of the function definition macro DSLFNT. “Writing a Nonconversational Transaction” on page 25 describes how you can code your application program for automatic starting.

Transactions for automatic start are typically used to automatically route messages depending on the content of various fields. Processing involves inspecting the message and then moving the message to the appropriate target queue or queues. Since this logic can be completely defined in a MERVA Routing table, this program merely reads messages sequentially from the queue (GETN), appends a data area to the MSGTRACE field (TOF WRIT), and then invokes the routing logic (ROUB) to move the message with automatic delete to the target queues. (An example Routing table is not supplied, refer to any of the tables supplied with MERVA ESA for an example.)

Figure 11 is an example of a transaction for automatic start written in PL/I.

```
DSLBA06:   PROCEDURE OPTIONS(MAIN);
                                  /* ..must be uppercase for cics   */
                                  /* eib addressing set by cics */
/*******************************************************************
*                                                                  *
* Function   : MERVA ESA sample message-processing transaction     *
*                                                                  *
*              This program is an example CICS transaction that    *
*              could be associated with a MERVA function to process *
*              automatically any messages routed to the function.  *
*                                                                  *
*              A transaction is associated with a MERVA function by *
*              the TRAN= parameter of the function definition macro *
*              DSLFNT.                                             *
*                                                                  *
*              In this example message-processing is limited to    *
*              inspecting the message and routing it further       *
*              depending on the contents of various fields.  Since  *
*              this routing logic can be completely defined in a    *
*              Routing table, this program merely reads messages    *
*              sequentially from the queue, appends a data area to  *
*              the MSGTRACE field, and then invokes the routing     *
*              logic.  (The Routing table is not supplied, refer    *
*              to any of the tables supplied with MERVA ESA for     *
*              an example).                                        *
*                                                                  *
*                                                                  *
* Dependencies:                                                    *
*                                                                  *
* Environment : MVS CICS, VSE CICS                                 *
*                                                                  *
*******************************************************************/
```

Figure 11. DSLBA06P PL/I Sample CICS API Transaction for Automatic Start (Part 1 of 4)

```
%include dslapiws;
dcl ws like intwstor automatic;
%include dslapitp;
dcl tp like tofparm automatic;
%include dslapitu;
dcl tucb like inttucb automatic;
dcl 1 api_parm_list,
      2 parm(3)              ptr;
dcl saveqsn bin fixed(31);
/* get the MERVA TUCB                                          */
EXEC CICS RETRIEVE INTO(TUCB) LENGTH(STORAGE(TUCB));

/* initialize the MERVA API                                    */
ws.INTFUNC = 'INIT';
unspec(ws.INTCWA) = 0;
api_parm_list.parm(1) = addr(ws);
EXEC CICS LINK PROGRAM('DSLAPCIC')
  COMMAREA(API_PARM_LIST) LENGTH(4);
if ws.INTRC ¬= '  ' then do;
  call write_to_operator;
  EXEC CICS RETURN;
end;

/* process sequentially all messages in the queue              */
ws.INTQSN = 0;                          /* start at the beginning */
do while (ws.INTRC = '  ');
  ws.INTFUNC = 'GETN';                      /* exclusive get next */
  ws.INTQUEUE = tucb.tucname;                /* --from this queue */
  api_parm_list.parm(1) = addr(ws);
  EXEC CICS LINK PROGRAM('DSLAPCIC')
    COMMAREA(API_PARM_LIST) LENGTH(4);
  if ws.INTRC ¬= '  ' then
    leave;
  saveqsn = ws.INTQSN;                       /* save QSN for ROUB */

  /* add a msgtrace data area to the msg                       */
  call write_msgtrace;
  if ws.INTRC ¬= '  ' then
    call write_to_operator;

  /* move the message to the next queue using a routing table  */
  ws.INTFUNC = 'ROUB';
  ws.INTQUEUE = tucb.tucname;   /* use this functions rtng table  */
  ws.INTBQUE = tucb.tucname;                      /* source queue */
  ws.INTBQSN = saveqsn;                      /* qsn of source msg */
  EXEC CICS LINK PROGRAM('DSLAPCIC')
    COMMAREA(API_PARM_LIST) LENGTH(4);
end; /* end while */
if ws.INTRC ¬= '09' then do;                /* no more messages ? */
  call write_to_operator;
end;

/* terminate the MERVA API                                     */
call terminate;
EXEC CICS RETURN;
```

Figure 11. DSLBA06P PL/I Sample CICS API Transaction for Automatic Start (Part 2 of 4)

```
  /*******************************************************************/
  /*                                                                 */
  /*  Write an error-message                                         */
  /*                                                                 */
  /*******************************************************************/
  write_to_operator: procedure;
    dcl 1 opmsg,
          2 data       char(120);

    put string (opmsg.data)  edit
      ('dslba06, function ', ws.INTFUNC, ', RC=', ws.INTRC) (4(a));
    ws.INTFUNC = 'WTO ';
    api_parm_list.parm(1) = addr(ws);
    api_parm_list.parm(2) = addr(opmsg);
    EXEC CICS LINK PROGRAM('DSLAPCIC')
      COMMAREA(API_PARM_LIST) LENGTH(8);
  end write_to_operator;

   /*******************************************************************/
  /*                                                                 */
  /*  Terminate the API interface                                    */
  /*                                                                 */
  /*******************************************************************/
  terminate: procedure;
    ws.INTFUNC = 'TERM';
    api_parm_list.parm(1) = addr(ws);
    EXEC CICS LINK PROGRAM('DSLAPCIC')
      COMMAREA(API_PARM_LIST) LENGTH(4);
  end terminate;
/*******************************************************************/
  /*                                                                 */
  /*  Append a data area to the MSGTRACE field of the message in the */
  /*  API internal queue-buffer                                      */
  /*                                                                 */
  /*******************************************************************/
  write_msgtrace: procedure;
    dcl 1 tofbuffer,
          %include dslapibp;
          2 data       char(100);
    dcl 1 api_parm_list,
          2 parm(3)              ptr;
    dcl today                    char(6);
    dcl tod                      char(6);

    tofbuffer.buffer_prefix.bufsize = length(tofbuffer.data) + 8;
    today, tod = ' ';
    /*  get date and time from MERVA  */
    ws.INTFUNC = 'READ';
    tp.toffdnam = 'DSLDATE1';
    tp.tofmodif = ' ';
    tp.toffdnl = 0;
    tp.toffdfg, tp.toffdoc, tp.toffdda = 1;
    api_parm_list.parm(1) = addr(ws);
    api_parm_list.parm(2) = addr(tp);
    api_parm_list.parm(3) = addr(tofbuffer);
    EXEC CICS LINK PROGRAM('DSLAPCIC')
         COMMAREA(API_PARM_LIST) LENGTH(12);
    if ws.INTRC = '  ' then do;
      today = substr(tofbuffer.data,1,6);
    end;
```

Figure 11. DSLBA06P PL/I Sample CICS API Transaction for Automatic Start (Part 3 of 4)

```
      ws.INTFUNC = 'READ';
   tp.toffdnam = 'DSLTIME1';
   EXEC CICS LINK PROGRAM('DSLAPCIC')
        COMMAREA(API_PARM_LIST) LENGTH(12);
   if ws.INTRC = '  ' then do;
     tod = substr(tofbuffer.data,1,6);
   end;
   /*  now add a new MSGTRACE data area.. */
   ws.INTFUNC = 'WRIT';
   tp.toffdnam = 'MSGTRACE';
   tp.toffdnl = 0;
   tp.toffdfg, tp.toffdoc = 1;
   tp.toffdda = 32767;
   put string (tofbuffer.data) edit
         ('dslba06', tucb.tucname, '0000', today, tod)
         ( a(8),     a(8),          a(4),  a(6),  a(6));
   tofbuffer.buffer_prefix.datasize = 36;
   EXEC CICS LINK PROGRAM('DSLAPCIC')
        COMMAREA(API_PARM_LIST) LENGTH(12);
 end write_msgtrace;

end dslba06;
```

Figure 11. DSLBA06P PL/I Sample CICS API Transaction for Automatic Start (Part 4 of 4)

Conversational Transaction

Figure 12 shows the general structure of the user sample dialog system. There are four programs, each in COBOL and PL/I, which together make up the dialog application. They use API functions for data entry and routing (DSLBA21x), data verification and routing (DSLBA22x), and to get information from the MERVA ESA message queues (DSLBA23x).

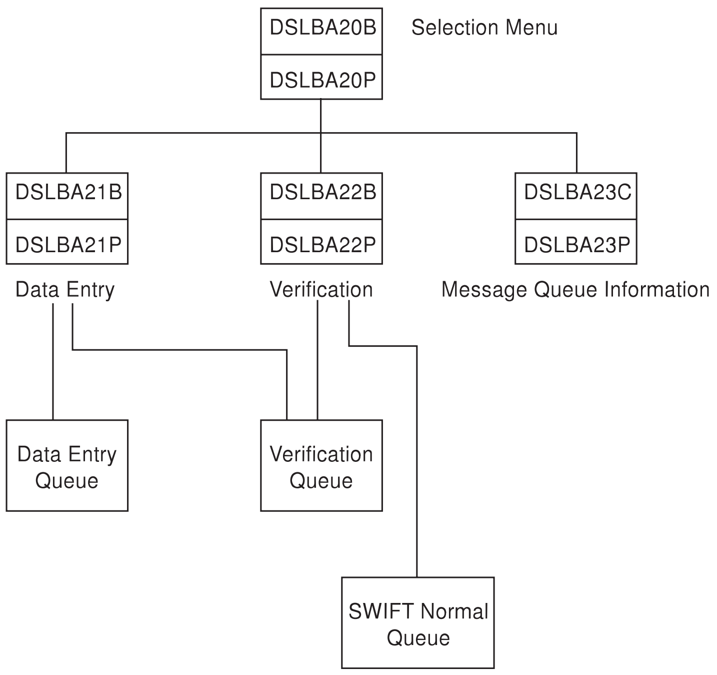


Figure 12. General Structure of the User Sample Dialog System

Working Storage Structure of the Dialog Programs

Figure 13 shows the working storage structure that is used in all sample dialog programs.

```
Common area (more details in source text).


Working areas for DSLAPI:

     PARWSTOR    (working area with 72 characters).

     PARMLIST    (address list for DSLAPI).
         ADDR1
         ADDR2
         ADDR3

     LLPARM      (length-field of parmlist).

     TOFFIELD    (data buffer for using TOF-services in sample
                  programs DSLBA21x and DSLBA22x).

     W-INPUT-HEADER
                 (data buffer to prepare the MSGSWIFT buffer
                  in sample program DSLBA21x).


Copy Book for BMS Map.


Standard working area for all maps in this sample dialog-system.


Copy Book for DSLAPI:

     DSLAPCBL for COBOL programs.
     DSLAPPLI for PL/I  programs.


Copy Book DFHAID for CICS.


Working area for sample programs (more details in source text)
```

Figure 13. Working Storage Structure Used in All Sample Programs with DSLAPI Functions

Note: All source code, BMS map sources, and the map copybooks for these samples are provided with the machine-readable material.

Sample Application Program DSLBA20x

The application program DSLBA20x displays the selection panel from which the user chooses one of the possible transactions in this sample dialog system. It is a CICS program and does not use API services. If you need more information about this program, refer to the source code provided with MERVA ESA.

Sample Application Program DSLBA21x

With the application program DSLBA21x you can enter a SWIFT message type S100 and route it to the Data Entry queue or the Verification queue. The following example uses the TOF service functions of DSLAPI. Figure 14 shows the processing structure

```
Preparing address list for DSLAPI with external program DSLAPIPL
for the working storage area INTWSTOR:

       Call DSLAPIPL with PARWSTOR PARMLIST INTWSTOR MSGSWIFT.


Initializing DSLAPI:

       INTFUNC = 'INIT'
       Call DSLAPI via EXEC CICS link program ('DSLAPCIC')
       with address list (PARMLIST)


Before you put a new entered message in a queue you have to
prepare the MSGSWIFT buffer and put a dummy element in the internal
Queue Buffer with the 'Put SWIFT Message' function to initialize
the TOF:

       INTFUNC = 'PUTS'
       INTQUEUE = 'D3DE0' (for example)
       INTQSN = Zero
       INIZIALIZE MSGSWIFT FIELDS
       INPUT—HEADER OF MSGSWIFT = W—INPUT—HEADER
       Call DSLAPI via EXEC CICS link program ('DSLAPCIC')
       with address list (PARMLIST)


Preparing address list for DSLAPI with external program DSLAPIPL
for the working storage areas INTWSTOR, TOFPARM, and TOFFIELD:

       Call DSLAPIPL with PARWSTOR PARMLIST INTWSTOR TOFPARM
                          TOFFIELD

Write the fields SW20, SW32, SW50, and SW59 from the field buffer
(TOFFIELD) through the Internal TOF to the Internal Queue Buffer
(INTWSTOR):

   ┌─►  INTFUNC = 'WRIT'
   3    TOFMODIF = 'VFIRST'
   3    TOFFDNAM = SWIFT field name
   3    LENGTH2 = Zero (DSLAPI calculates the actual data length
   3                    from the data buffer in TOFFIELD)
   3    Put data you entered in the map to the data buffer in
   3    TOFFIELD (T—DATA)
   3    Call DSLAPI via EXEC CICS link program ('DSLAPCIC')
   └──  with address list (PARMLIST)


Preparing address list for DSLAPI with external program DSLAPIPL
for the working storage area INTWSTOR:

       Call DSLAPIPL with PARWSTOR PARMLIST INTWSTOR.
```

Figure 14. Processing Structure of Program DSLBA21x (Part 1 of 2)

```
Take the queue element from the Internal Queue Buffer (INTWSTOR)
and put it in the MERVA queue with the 'PUT' Function:

       INTFUNC = 'PUT '
       INTQUEUE = 'D3VE0' (for example)
       Call DSLAPI via EXEC CICS link program ('DSLAPCIC')
       with address list (PARMLIST)


Termination DSLAPI:

       INTFUNC = 'TERM'
       Call DSLAPI via EXEC CICS link program ('DSLAPCIC')
       with address list (PARMLIST)
```

Figure 14. Processing Structure of Program DSLBA21x (Part 2 of 2)

Sample Application Program DSLBA22x

With the application program DSLBA22x you can verify a SWIFT message type S100 and route it to the Verification queue or to the SWIFT Normal queue. The following example also uses the TOF service functions of DSLAPI. Figure 15 shows the processing structure.

```
Preparing address list for DSLAPI with external program DSLAPIPL
for the working storage area INTWSTOR:

       Call DSLAPIPL with PARWSTOR PARMLIST INTWSTOR.


Initializing DSLAPI:

       INTFUNC = 'INIT'
       Call DSLAPI via EXEC CICS link program ('DSLAPCIC')
       with address list (PARMLIST)


Browse a queue element and put it in the Internal Queue Buffer
with the 'Get Next Unconditionally' Function:

       INTFUNC = 'GETU'
       INTQUEUE = 'D3VE0' (for example)
       INTQSN = Zero (to get the first queue element) or
       INTQSN = QSN of the last call you have saved in the Common
                Area (to get the next Queue Element)
       Call DSLAPI via EXEC CICS link program ('DSLAPCIC')
       with address list (PARMLIST)


Preparing address list for DSLAPI with external program DSLAPIPL
for the working storage areas INTWSTOR, TOFPARM, and TOFFIELD:

       Call DSLAPIPL with PARWSTOR PARMLIST INTWSTOR TOFPARM
                          TOFFIELD
```

Figure 15. Processing Structure of Program DSLBA22x (Part 1 of 4)

```
Read the fields SW20, SW32, SW50, and SW59 from the Internal Queue
Buffer through the Internal TOF and put it in the field buffer:

  ┌─►  INTFUNC = 'READ'
  3    TOFMODIF = 'VFIRST'
  3    TOFFDNAM = SWIFT field name
  3    LENGTH2 = Zero (DSLAPI calculates the actual data length
  3                    from the data buffer in TOFFIELD)
  3    Call DSLAPI via EXEC CICS link program ('DSLAPCIC')
  3    with address list (PARMLIST)
  3    Put data you have in the data buffer to the appropriate
  └──  fields in the map.
```

Figure 15. Processing Structure of Program DSLBA22x (Part 2 of 4)

```
Preparing address list for DSLAPI with external program DSLAPIPL
for the working storage area INTWSTOR:

       Call DSLAPIPL with PARWSTOR PARMLIST INTWSTOR.


Retrieve the same queue element and put it in the Internal Queue
Buffer with the 'Get Next' Function:

       INTFUNC = 'GETN'
       INTQUEUE = 'D3VE0' (for example)
       INTQSN = QSN you have saved in the Common Area (to get the
                same Queue Element)
       Call DSLAPI via EXEC CICS link program ('DSLAPCIC')
       with address list (PARMLIST)


Write the fields SW20, SW32, SW50, and SW59 from the field buffer
(TOFFIELD) through the Internal TOF to the Internal Queue Buffer
(INTWSTOR):

  ┌─►  INTFUNC = 'WRIT'
  3    TOFMODIF = 'VFIRST'
  3    TOFFDNAM = SWIFT field name
  3    LENGTH2 = Zero (DSLAPI calculates the actual data length
  3                    from the data buffer in TOFFIELD)
  3    Put data you verified in the map to the data buffer in
  3    TOFFIELD (T—DATA)
  3    Call DSLAPI via EXEC CICS link program ('DSLAPCIC')
  └──  with address list (PARMLIST)
```

Figure 15. Processing Structure of Program DSLBA22x (Part 3 of 4)

```
Preparing address list for DSLAPI with external program DSLAPIPL
for the working storage area INTWSTOR:

       Call DSLAPIPL with PARWSTOR PARMLIST INTWSTOR.


Take the queue element from the Internal Queue Buffer (INTWSTOR)
and put it in the MERVA queue with the 'PUT with Back—Reference'
Function:

       INTFUNC = 'PUTB'
       INTBQUEUE = 'D3VE0' (for example)
       INTBQSN = INTQSN (QSN from the element you want to route)
       INTQUEUE = 'D3SWN' (for example)
       Call DSLAPI via EXEC CICS link program ('DSLAPCIC')
       with address list (PARMLIST)


Termination DSLAPI:

       INTFUNC = 'TERM'
       Call DSLAPI via EXEC CICS link program ('DSLAPCIC')
       with address list (PARMLIST)
```

Figure 15. Processing Structure of Program DSLBA22x (Part 4 of 4)

Sample Application Program DSLBA23x

With the application program DSLBA23x you can display messages of all MERVA ESA queues in S.W.I.F.T format. The following example uses the field-service program DSLAPFFS. Figure 16 shows the processing structure of the program DSLBA23x.

```
Preparing address list for DSLAPI with external program DSLAPIPL
for the working storage areas INTWSTOR and MSGSWIFT:

       Call DSLAPIPL with PARWSTOR PARMLIST INTWSTOR MSGSWIFT.


Initializing DSLAPI:
       INTFUNC = 'INIT'
       Call DSLAPI via EXEC CICS link program ('DSLAPCIC')
       with address list (PARMLIST)


Browse a queue element and put it in the Internal Queue Buffer
with the 'Get Next Unconditionally' Function:

       INTFUNC = 'GETU'
       INTQUEUE = Queue name you entered in Panel
       INTQSN = Zero (to get the first queue element) or
       INTQSN = QSN of the last call you have saved in the Common
                Area (to get the next Queue Element)
       Call DSLAPI via EXEC CICS link program ('DSLAPCIC')
       with address list (PARMLIST)


Map the message in SWIFT format from the Internal Queue Buffer
(INTWSTOR) through the Internal TOF to the Message Buffer
(MSGSWIFT) with the 'Get SWIFT Message' Function:

       INTFUNC = 'GETS'
       Call DSLAPI via EXEC CICS link program ('DSLAPCIC')
       with address list (PARMLIST)


Initializing field service program DSLAPFFS:

       FLDFUNC = 'INIT'
       Call DSLAPFFS with FLDWSTOR MSGSWIFT.


Get the message line by line from MSGSWIFT until end of message:

    ┌─► FLDFUNC = 'DATA'
    3
    └── Call DSLAPFFS with FLDWSTOR MSGSWIFT.


Termination DSLAPI:

       INTFUNC = 'TERM'
       Call DSLAPI via EXEC CICS link program ('DSLAPCIC')
       with address list (PARMLIST)
```

Figure 16. Processing Structure of Program DSLBA23x

High-Level Language MFS Exit

The sample high-level language (HLL) MFS exit, DSLBA30x, is an HLL version of the Assembler sample separation exit DSLMS911. It returns the last-but-one data area of the field named in the MFS field reference.

A separation exit routine is usually used to isolate part of a field. It can be invoked by defining a subfield of the field and specifying a separation exit routine (SEPR=) for the subfield. A subfield of the MSGTRACE field has been defined in this way in the MERVA field definition table, DSLFDTT, to invoke separation exit DSLMS911:

```
MSGTRLB1 DSLLSUBF LENGTH=(0,80,V),SEPR=911
```

The exit routine first checks that it has been invoked properly: there must be an output buffer for the subfield to be returned, and the exit routine must have been invoked to read the subfield.

Then field-level access services (FLDG) are used to obtain the name of the field and its position from the MFS field reference. The exit routine determines the last-but-one data area number by using the TOF LASTDA modifier to obtain the index of the last data area.

If the field has more than one data area, the last-but-one data area is then read into the output buffer, otherwise the data-size field in the output buffer prefix is set to zero to indicate there is no subfield. Then control is returned to MFS.

Note that the sample does not support subfields in nested repeatable sequences.

```
 #pragma runopts(EXECOPS)
/*********************************************************************
*                                                                    *
* Function    : MERVA ESA sample MFS separation exit DSLMS911         *
*     Returns the last-but-one data area of the field named in       *
*     the MFS field-reference.                                       *
*     It is invoked by defining a subfield of the field and          *
*     specifying a separation exit (SEPR=911) for the subfield.      *
*     A subfield of the MERVA MSGTRACE field is so defined:          *
*        MSGTRLB1 DSLLSUBF LENGTH=(0,80,V),SEPR=911                  *
*                                                                    *
*********************************************************************/

 #include <cics.h>
 #include <string.h>
 #include <stddef.h>
 #include "dslapc.h"

   struct obuf {                    /* mfslobuf */
     struct BufferPrefix pfx;
     char odata[1];                 /* we dont know the length */
   };
   union {
     char fld_string[100];
     int fld_value;
   } fld_buffer;
```

Figure 17. DSLBA30C C/370 Sample HLL MFS Exit (Part 1 of 4)

```
int main(int argc, char *argv[]) {
  struct {char *parm1; char *parm2; char *parm3;} *ca_ptr;
  struct INTWSTOR *ws_ptr;
  struct MFSL *mfs_ptr;
  struct TOFPARM tofpl;
  struct obuf *optr;
  char cics;
  void call_fld_service
      (char,struct INTWSTOR *, void *, void *);
  void call_tof_service
      (char,struct INTWSTOR *,struct TOFPARM *, void *);

  /* access the parameters */
  if (argc > 1) {                 /* not CICS */
    cics = 0;
    ws_ptr = (struct INTWSTOR *) argv[0];
    mfs_ptr = (struct MFSL *) argv[1];
  } else {
    cics = 1;
    EXEC CICS ADDRESS COMMAREA(ca_ptr);
    EXEC CICS ADDRESS EIB(dfheiptr);
    ws_ptr = (struct INTWSTOR *) ca_ptr->parm1;
    mfs_ptr = (struct MFSL *) ca_ptr->parm2;
  }

  if (mfs_ptr->MFSLOBUF == NULL) {  /* an output buffer is needed */
    mfs_ptr->MFSLRET = MFSROK;
    return;
  } else {
    optr = (struct obuf *) mfs_ptr->MFSLOBUF;
  }

  if (!mfs_ptr->MFSLOPT2.MFSLO2RD) {
    mfs_ptr->MFSLREAS = MFSREMC7;
    mfs_ptr->MFSLRET = MFSRWNG;
    return;
  }

  /* FLDSTEX must be on if we were invoked for a subfield */
  memcpy(ws_ptr->INTFUNC,"FLDG",4);
  call_fld_service(cics,ws_ptr,"FLDSTEX",&fld_buffer);
  if ( memcmp(ws_ptr->INTRC,"  ",2) != 0 ||  /* no MFS fld.ref. */
       memcmp(fld_buffer.fld_string,"0",1) == 0 ) {
    optr->pfx.datasize = 0;       /* indicate field is empty */
    mfs_ptr->MFSLRET = MFSROK;
    return;
  }
  /* get the subfields main field-name from the MFS fld.ref. */
  call_fld_service(cics,ws_ptr,"FLDNAME0",&fld_buffer);
  if ( memcmp(fld_buffer.fld_string," ",1) == 0 ) {
    optr->pfx.datasize = 0;       /* indicate field is empty */
    mfs_ptr->MFSLRET = MFSROK;
    return;
  } else {
    memcpy(tofpl.TOFFDNAM,fld_buffer.fld_string,8);
  }
```

Figure 17. DSLBA30C C/370 Sample HLL MFS Exit (Part 2 of 4)

```
  /* move the MFS fld-reference to the API TOFPARM.. */
  call_fld_service(cics,ws_ptr,"FLDNI",&fld_buffer);
  tofpl.TOFFDNL = fld_buffer.fld_value;
  call_fld_service(cics,ws_ptr,"FLDFG",&fld_buffer);
  tofpl.TOFFDFG = fld_buffer.fld_value;
  call_fld_service(cics,ws_ptr,"FLDRS",&fld_buffer);
  tofpl.TOFFDOC = fld_buffer.fld_value;
  call_fld_service(cics,ws_ptr,"FLDDA",&fld_buffer);
  tofpl.TOFFDDA = fld_buffer.fld_value;

  /* find the last data area index of the field */
  memset(tofpl.TOFMODIF,' ',sizeof tofpl.TOFMODIF);
  memcpy(tofpl.TOFMODIF,"LASTDA",6);
  memcpy(ws_ptr->INTFUNC,"READ",4);
  call_tof_service(cics,ws_ptr,&tofpl,optr);
  if ( memcmp(ws_ptr->INTRC,"  ",2) != 0 ||  /* something wrong */
       tofpl.TOFTSVRC != 0 ) {
    optr->pfx.datasize = 0;       /* indicate field is empty */
    mfs_ptr->MFSLRET = MFSROK;
    return;
  }

  if (tofpl.TOFFDDA > 1) {
    /* now read last data area but one */
    tofpl.TOFFDDA --;
    memset(tofpl.TOFMODIF,' ',sizeof tofpl.TOFMODIF);
    memcpy(ws_ptr->INTFUNC,"READ",4);
    call_tof_service(cics,ws_ptr,&tofpl,optr);
    if ( memcmp(ws_ptr->INTRC,"  ",2) != 0 || /* something wrong */
         tofpl.TOFTSVRC != 0 ) {
      optr->pfx.datasize = 0;       /* indicate field is empty */
    }
  } else {
    optr->pfx.datasize = 0;       /* indicate field is empty */
  }
  mfs_ptr->MFSLRET = MFSROK;
  return;
}

void call_tof_service(char cics, struct INTWSTOR *ws_ptr,
              struct TOFPARM *tofpl, void *buf_ptr) {
  struct { char *parm1;
           char *parm2;
           char *parm3;
  } apipl;

  if (cics) {
    apipl.parm1 = (char *) ws_ptr;
    apipl.parm2 = (char *) tofpl;
    apipl.parm3 = (char *) buf_ptr;
    EXEC CICS LINK PROGRAM("DSLAPCIC") COMMAREA(&apipl) LENGTH(12);
  } else {
    DSLAPI(ws_ptr,tofpl,buf_ptr);
  }
}
```

Figure 17. DSLBA30C C/370 Sample HLL MFS Exit (Part 3 of 4)

```
void call_fld_service(char cics, struct INTWSTOR *ws_ptr,
              void * fldname, void * flddata) {
  struct { char *parm1;
           char *parm2;
           char *parm3;
  } apipl;

  if (cics) {
    apipl.parm1 = (char *) ws_ptr;
    apipl.parm2 = fldname;
    apipl.parm3 = flddata;
    EXEC CICS LINK PROGRAM("DSLAPCIC") COMMAREA(&apipl) LENGTH(12);
  } else {
    DSLAPI(ws_ptr,fldname,flddata);
  }
}
```

Figure 17. DSLBA30C C/370 Sample HLL MFS Exit (Part 4 of 4)

Appendix C. Batch Utilities in REXX

A number of batch utilities written in REXX are distributed with MERVA ESA. This appendix describes these programs.

All batch utilities are named in a consistent fashion: *DSLBAnnR*, where ‘nn’ is the utility number, and ‘R’ indicates the programming language REXX:

DSLBA12R	Print a specified or all queue elements of a queue
DSLBA13R	Print the MERVA ESA journal
DSLBA14R	Scanning a message TOF to display the TOF structure
DSLBA15R	Print the MERVA ESA User file
DSLBA16R	Print a cross-reference of function names and allowed user IDs from the User file
DSLBA17R	Check date fields in the user file
DSLBA50R	Print queue status list
DSLBA51R	Print queue key list
DSLBA52R	Copy or move messages from one queue to another. Optionally sort them by key value
DSLBA53R	Scan a queue for ‘old’ messages.

Batch utilities are distributed in the MERVA ESA samples library, MERVA.SDSLSAM0. In VSE they are distributed in the source sublibrary.

DSLBA12R - Print Queue Element(s)

DSLBA12R reads a specified or all queue elements of a queue, maps them to SWIFT II format, and then prints them. Dependencies: MERVA ESA must be active.

Job Control Statements

The following figure shows the MVS JCL to run DSLBA12R.

```
//.......  JOB  ...........
//REXXB    EXEC PGM=DSLAREXX,REGION=8M,
//             PARM='DSLBA12R,parm1 parm2 parm3'
//*
//*           .. MERVA ESA LOAD LIBRARY
//STEPLIB  DD DSN=loadlib,DISP=SHR
//*
//*           .. IF CURCODE=FILE SPECIFIED IN DSLPRM
//DWSCUR   DD DSN=curfile,DISP=SHR
//*
//*           .. ON THIS PDS: DSLBA12R
//SYSEXEC  DD DSN=samplib,DISP=SHR
//*
//*           .. LISTING DATASET (VB136)
//SYSTSPRT DD DSN=listds,DISP=OLD
//
```

Figure 18. MVS JCL to Run Batch Utility DSLBA12R - Print Queue Elements

In the JCL, the lowercase parameters have the following meanings:

loadlib	The name of the load library containing the MERVA ESA programs.
curfile	The name of the currency code file. Needed only when CURCODE=FILE is specified in your DSLPRM.
samplib	The name of the library containing the program DSLBA12R.
listds	The name of the listing data set. Must be preallocated, record format VB, logical record length 136 recommended.

Runtime Parameters

The following parameters can be specified in the PARM field of the EXEC statement:

parm1	Queue name.
parm2	Queue sequence number (QSN). If not specified, or specified as '*', all queue elements not flagged 'in-service' are printed.
parm3	Log level. From **1** (basic) to **4** (all). The default is 2.

Customization

In the MERVA ESA customization module DSLPRM you can set the following parameter:

prtname	Your institution name as it should appear in the printout of (most) REXX batch utilities.

Sample Printout

The following figure shows the information printed after the execution of the DSLBA12R utility.

```
MERVA ESA V4.1 DSLBA12R                       11. Oct. 1999  15:32:11
(C) Copyright IBM Corp. 1997, 1999

+ ----------------------------------------------------------- +
|         S A M P L E   B A N K   B o e b l i n g e n         |
+ ----------------------------------------------------------- +


DDDDD    SSSSSS   LLL      BBBBB     AAAA      111    2222    RRRRR
DDDDDD   SSSSSS   LLL      BBBBBB   AAAAAA    1111   222222   RRRRRR
DD  DD   SS       LLL      BB  BB   AA  AA   11111   22  22   RR  RR
DD  DD   SS       LLL      BB  BB   AA  AA     111       22   RR  RR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA     111      22    RRRRRR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA     111     22     RRRRRR
DD  DD       SS   LLL      BB  BB   AA  AA     111    22      RR  RR
DD  DD       SS   LLL      BB  BB   AA  AA     111   22  22   RR  RR
DDDDDD   SSSSSS   LLLLLL   BBBBBB   AA  AA     111   222222   RR  RR
DDDDD    SSSSSS   LLLLLL   BBBBB    AA  AA     111   222222   RR  RR


Print unconditionally specified queue element, or if no QSN specified,
or specified as '*', all queue elements not flagged 'in-service'.

DSLBA12R_001I : Runtime parameter 'Queue name' .......... : L1DE0

DSLBA12R_002I : Runtime parameter 'Queue sequence number' : *

DSLBA12R_003I : Runtime parameter 'Log level' ........... : 2
                Allowed log levels are: 1 .. 4, and '*'.


================================================================================

DSLBA12R_011I : GETN of the following element in queue L1DE0 was successful:
                - INTQSN  : 40
                - INTKEY1 : T5
                - INTKEY2 :

Message (183 characters):
----------------------------------------------------------------------
{1:F01VNDEBET2AXXX0000000000}{2:I100VNDOBET2XVIBN}{3:{108:A new refere
nce}}{4:  :20:T5  :32A:960505DEM55,55  :50:Emil Eisbaer  :52A:SAMPBANK
  :59:/5678  Eusebia Eldorado  :71A:BEN  -}
----------------------------------------------------------------------
```

Figure 19. Printout of the DSLBA12R Utility (Part 1 of 2)

```
================================================================================

DSLBA12R_011I : GETN of the following element in queue L1DE0 was successful:
                - INTQSN  : 43
                - INTKEY1 : T7
                - INTKEY2 :
                - INTDOUBL: DOUBLE

Message (175 characters):
----------------------------------------------------------------------
{1:F01VNDEBET2AXXX0000000000}{2:I100VNDOBET2ABICN}{4:  :20:T7  :32A:96
0303DEM77,00  :50:Gustav Gans  :52D:SAMPBANKXXX  :53A:SAMPBANK001  :59
:/7890  Gerhard Geier  :71A:OUR  -}
----------------------------------------------------------------------
================================================================================

DSLBA12R_011I : GETN of the following element in queue L1DE0 was successful:
                - INTQSN  : 44
                - INTKEY1 : T8
                - INTKEY2 :
                - INTDOUBL: DOUBLE

DSLBA12R_103W : MSGG for queue element 44 in queue L1DE0
                failed with intrc 00.
                MFS has detected checking errors.
                DSL883I MFS04141 MFS=PUT/NET ID=         RC=04 RS=141
                DWS3516 Field SW79 contains a non-SWIFT character

Message (122 characters):
----------------------------------------------------------------------
{1:F01VNDEBET2AXXX0000000000}{2:I199VNDOBET2AXXXN}{3:{108:A new refere
nce}}{4:  :20:T8  :79:Hi. This MT199 has errors!  -}
----------------------------------------------------------------------

.
.
.

================================================================================

DSLBA12R_005I : QSN          MSGG return code
                ----------   ------------------------------
                0000000040   (blank) = ok
                0000000043   (blank) = ok
                0000000044   ***  00 = checking error  ***
                ...

DSLBA12R_006I : Number of successfully GETNed queue elements: 35
                Number of successfully MSGGed queue elements: 24

DSLBA12R_007I : DSLBA12R ended with return code 4 - Warning.
                Total processing time was 4.37 seconds.
```

Figure 19. Printout of the DSLBA12R Utility (Part 2 of 2)

DSLBA13R - Print the MERVA Journal

DSLBA13R counts occurrences of journal IDs and optionally prints specified or all journal records to a sequential data set. Dependencies: MERVA ESA must be active.

Job Control Statements

The following figure shows the MVS JCL to run DSLBA13R.

```
//.......  JOB  ...........
//REXXB    EXEC PGM=DSLAREXX,REGION=8M,
//             PARM='DSLBA13R,parm1 parm2 parm3 parm4'
//*
//*          .. MERVA ESA LOAD LIBRARY
//STEPLIB  DD DSN=loadlib,DISP=SHR
//*
//*          .. ON THIS PDS: DSLBA13R
//SYSEXEC  DD DSN=samplib,DISP=SHR
//*
//*          .. LISTING DATASET (VB136)
//SYSTSPRT DD DSN=listds,DISP=OLD
//
```

Figure 20. MVS JCL to Run Batch Utility DSLBA13R - Print MERVA Journal

In the JCL, the lowercase parameters have the following meanings:

loadlib	The name of the load library containing the MERVA ESA programs.
samplib	The name of the library containing the program DSLBA13R.
listds	The name of the listing data set. Must be preallocated, record format VB, logical record length 136 recommended.

Runtime Parameters

The following parameters can be specified in the PARM field of the EXEC statement:

parm1	Start date. If specified, must be a 6-digit number in the format YYMMDD, an 8-digit number in the format YYYYMMDD, or '*'. It is not checked that it is a valid date.
parm2	Start time. If specified, must be a 4-digit number in the format HHMM, a 6-digit number in the format HHMMSS, a 8-digit number in the format HHMMSSPP, or '*'. It is not checked that it is a valid time.
parm3	Journal ID from 00 to FF. If no journal ID is specified or specified as '*', then all journal records are printed. If specified as '-' (dash), then the journal IDs are only counted and no journal records are printed.
parm4	Log level. From **1** (basic) to **4** (all). The default is 2.

Customization

In the MERVA ESA customization module DSLPRM you can set the following parameter:

prtname	Your institution name as it should appear in the printout of (most) REXX batch utilities.

Sample Printout

The following figure shows the information printed after the execution of the DSLBA13R utility.

```
MERVA ESA V4.1 DSLBA13R                      11. Oct. 1999  17:25:56
(C) Copyright IBM Corp. 1997, 1999

+ ----------------------------------------------------------- +
|         S A M P L E   B A N K   B o e b l i n g e n         |
+ ----------------------------------------------------------- +


DDDDD    SSSSSS   LLL      BBBBB     AAAA      111    3333    RRRRR
DDDDDD   SSSSSS   LLL      BBBBBB   AAAAAA    1111   333333   RRRRRR
DD  DD   SS       LLL      BB  BB   AA  AA   11111   33  33   RR  RR
DD  DD   SS       LLL      BB  BB   AA  AA     111       33   RR  RR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA     111    3333    RRRRRR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA     111    3333    RRRRRR
DD  DD       SS   LLL      BB  BB   AA  AA     111       33   RR  RR
DD  DD       SS   LLL      BB  BB   AA  AA     111   33  33   RR  RR
DDDDDD   SSSSSS   LLLLLL   BBBBBB   AA  AA     111   333333   RR  RR
DDDDD    SSSSSS   LLLLLL   BBBBB    AA  AA     111    3333    RR  RR


Count occurrences of journal IDs and optionally print specified or
all journal records.

DSLBA13R_001I : Runtime parameter 'From date'  : 19990501

DSLBA13R_002I : Runtime parameter 'From time'  : *

DSLBA13R_003I : Runtime parameter 'Journal ID' : 04
                '-' = count only, '*' = all

DSLBA13R_004I : Runtime parameter 'Log level'  : 2
                Allowed log levels are:
                1 = basic .. 4 = all

DSLBA13R_006I : Current DSLPRM settings:

                MERVA name ............ (NAME)    : MERVAESA
                MERVA identifier ...... (DSLID)   : MHEG
                CVT extension ......... (CVTEXTO) : 124
                Journal size .......... (JRNBUF)  : 15950
                        segmentation              : SEG
                        2- or 4-digit year        : YYYY

                Note: These are the values of the first DSLPRM module
                      found in the STEPLIB concatenation.
```

Figure 21. Printout of the DSLBA13R Utility (Part 1 of 2)

```
================================================================================
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
JOURNAL ID and KEY OF RECORD : X'04' - 19990506160931000   SON       DSLNUSR
        MAS1            A105
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
JOURNAL ID and KEY OF RECORD : X'04' - 19990509143245000   SON       DSLNUSR
        MASDBCS         D804
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
JOURNAL ID and KEY OF RECORD : X'04' - 19990514110937000   SON       DSLNUSR
        MAS2            A105
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
JOURNAL ID and KEY OF RECORD : X'04' - 19990514125640000   SON       DSLNUSR
        MASHEG          A105
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
JOURNAL ID and KEY OF RECORD : X'04' - 19990514130336000   SON       DSLNUSR
        MASHEG4         A105
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
JOURNAL ID and KEY OF RECORD : X'04' - 19990514135208000   SON       DSLNUSR
        MAS4            A105
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
JOURNAL ID and KEY OF RECORD : X'04' - 19990515143805000   SON       DSLNUSR
        DFHAC200        A105
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
JOURNAL ID and KEY OF RECORD : X'04' - 19990614155827000   SON       DSLNUSR
        SL000A          A105
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
JOURNAL ID and KEY OF RECORD : X'04' - 19990716145108000   SON       DSLNUSR
        CHINA1          A105
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
JOURNAL ID and KEY OF RECORD : X'04' - 19990716145202000   SON       DSLNUSR
        HUGO            A107

⋮

================================================================================

DSLBA13R_009I : Total number of journal records processed: 18669
                - record type X'00': 136
                - record type X'02': 136
                - record type X'03': 122
                - record type X'04': 460   ** printed **
                - record type X'05': 172
                - record type X'06': 936
                - record type X'07': 9190
                - record type X'08': 134
                - record type X'09': 122
                - record type X'10': 204
                - record type X'13': 209
                - record type X'14': 688
                - record type X'16': 5822
                - record type X'17': 132
                - record type X'18': 7
                - record type X'50': 55
                - record type X'51': 52
                - record type X'5F': 3
                - record type X'70': 38
                - record type X'72': 12
                - record type X'77': 2
                - record type X'7F': 37

DSLBA13R_010I : DSLBA13R ended with return code 0 - successful.
                Total processing time was 187.73 seconds.
```

Figure 21. Printout of the DSLBA13R Utility (Part 2 of 2)

DSLBA14R - Scanning a TOF

If you are uncertain about TOFs, and are not quite sure what nesting identifiers, field groups, data areas, and so on, are, an EXEC like the following can be a help.

It simply scans a TOF, listing out each data area of each field together with the field's field reference, that is, its nesting identifier, field group index, data area index, and occurrence number. Three levels of nested repeatable sequences are allowed for. Dependencies: MERVA ESA must be active.

Job Control Statements

The following figure shows the MVS JCL to run DSLBA14R.

```
//.......  JOB  ...........
//REXXB    EXEC PGM=DSLAREXX,REGION=8M,
//              PARM='DSLBA14R,parm1 parm2 parm3'
//*
//*            .. MERVA ESA LOAD LIBRARY
//STEPLIB  DD DSN=loadlib,DISP=SHR
//*
//*            .. ON THIS PDS: DSLBA14R
//SYSEXEC  DD DSN=samplib,DISP=SHR
//*
//*            .. LISTING DATASET (VB136)
//SYSTSPRT DD DSN=listds,DISP=OLD
//
```

Figure 22. MVS JCL to Run Batch Utility DSLBA14R - Scan a TOF

In the JCL, the lowercase parameters have the following meanings:

loadlib	The name of the load library containing the MERVA ESA programs.
samplib	The name of the library containing the program DSLBA14R.
listds	The name of the listing data set. Must be preallocated, record format VB, logical record length 136 recommended.

Runtime Parameters

The following parameters can be specified in the PARM field of the EXEC statement:

parm1	Queue name.
parm2	Queue sequence number (QSN). If not specified, the first queue element found is scanned.
parm3	Log level. From **1** (basic) to **4** (all). The default is 2.

Customization

In the MERVA ESA customization module DSLPRM you can set the following parameter:

prtname	Your institution name as it should appear in the printout of (most) REXX batch utilities.

Sample printout

The following figure shows the information printed after the execution of the DSLBA14R utility.

```
MERVA ESA V4.1 DSLBA14R                      26. Feb. 1999  10:13:22
(C) Copyright IBM Corp. 1997, 1999

+ ----------------------------------------------------------- +
|         S A M P L E   B A N K   B o e b l i n g e n         |
+ ----------------------------------------------------------- +


DDDDD    SSSSSS   LLL      BBBBB     AAAA      111       44   RRRRR
DDDDDD   SSSSSS   LLL      BBBBBB   AAAAAA    1111      444   RRRRRR
DD  DD   SS       LLL      BB  BB   AA  AA   11111     44     RR  RR
DD  DD   SS       LLL      BB  BB   AA  AA     111    44      RR  RR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA     111   444444   RRRRR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA     111   444444   RRRRRR
DD  DD       SS   LLL      BB  BB   AA  AA     111     44     RR  RR
DD  DD       SS   LLL      BB  BB   AA  AA     111     44     RR  RR
DDDDDD   SSSSSS   LLLLLL   BBBBBB   AA  AA     111     44     RR  RR
DDDDD    SSSSSS   LLLLLL   BBBBB    AA  AA     111     44     RR  RR


P r i n t   a   T O F S C A N .


DSLBA14R_001I : DSLBA14R started by user HEG at 26. Feb. 1999 10:13:22

------------------------------------------------------------------------------

INTQUEUE : L1DE0
INTQSN   : 3
INTKEY1  : T-990630-M100-17
INTKEY2  :


         Nesting Level
         | Field Group Index
         | | Repeatable Sequence Index
         | | | Data Area
         | | | |
         | | | |     No. of Nested Repeatable Sequences
         | | | |     | Nested Repeatable Sequence no. 1 .. 3 (3 blank if zero)
         V V V V     V V V V

DSLERRM  0 1 1 1     1 1 0       79
DSLEXIT  0 1 1 1     1 1 0        8 S100
NLEXIT   0 1 1 1     1 1 0        8 DSLEXIT
MSGTRACE 0 1 1 1     1 1 0       40 MAS1    L1DE0   00009902260953259605
MSGTRACE 0 1 1 2     1 1 0       40 DSLECQUUL1DE0   00009902260953379605
DSLUMR   0 1 1 1     1 1 0       28 MERVAESA0000003990226095337
SWBH     1 1 1 1     1 1 0       25 F01VNDEBET2AXXX0000000000
SWAH     1 2 1 1     1 1 0       17 I100VNDOBET2AXXXN
SAAH     1 2 1 1     1 1 0       12 VNDOBET2XXX
SAAH     1 2 1 2     1 1 0       12 VNDOBET2AXXX
SAAH     1 2 1 3     1 1 0       36 *VENDOR O
SAAH     1 2 1 4     1 1 0       20 *IBM GSDL BOEBLINGEN
SW103    1 3 1 1     1 1 0          ·· has no data areas
SW113    1 3 1 1     1 1 0          ·· has no data areas
```

Figure 23. Printout of the DSLBA14R Utility (TOFSCAN) (Part 1 of 2)

```
SW108    1 3 1 1     1 1 0       13 My SW108 ref.
SW115    1 3 1 1     1 1 0          ·· has no data areas
SW119    1 3 1 1     1 1 0          ·· has no data areas
SW20     1 5 1 1     1 1 0       16 T-990630-M100-17
SW32     1 5 1 1     1 1 0       15 990630USD123,45
SW50     1 5 1 1     1 1 0       12 Anton Ameise
SW50     1 5 1 2     1 1 0       14 Am Weberhof 7a
SW50     1 5 1 3     1 1 0       12 52070 Aachen
SW50     1 5 1 4     1 1 0        7 Germany
SW52     1 5 1 1     1 1 0          ·· has no data areas
SA52     1 5 1 1     1 1 0          ·· has no data areas
SW53     1 5 1 1     1 1 0          ·· has no data areas
SA53     1 5 1 1     1 1 0          ·· has no data areas
SW54     1 5 1 1     1 1 0          ·· has no data areas
SA54     1 5 1 1     1 1 0          ·· has no data areas
SW56     1 5 1 1     1 1 0          ·· has no data areas
SA56     1 5 1 1     1 1 0          ·· has no data areas
SW57     1 5 1 1     1 1 0       10 1234567890
SA57     1 5 1 1     1 1 0          ·· has no data areas
SW59     1 5 1 1     1 1 0       10 Berta Baer
SW59     1 5 1 2     1 1 0       14 Bundesallee 2b
SW59     1 5 1 3     1 1 0       12 10719 Berlin
SW59     1 5 1 4     1 1 0        7 Germany
SW70     1 5 1 1     1 1 0       21 No details of Payment
SW70     1 5 1 2     1 1 0       26 Just to have a second line
SW71     1 5 1 1     1 1 0        3 OUR
SW72     1 5 1 1     1 1 0       29 /REC/ This is SWIFT II format
SWTRAIL  1 255 1 1   1 1 0          ·· has no data areas

·· TOFSCAN finished

------------------------------------------------------------------------------

DSLBA14R_004I : DSLBA14R ended with return code 0 - successful.
                Total processing time was 0.85 seconds.
```

Figure 23. Printout of the DSLBA14R Utility (TOFSCAN) (Part 2 of 2)

DSLBA15R - Print the User File

You can use the batch utility DSLBA15R to list all or selected user file records. Dependencies:

- MERVA ESA must be active.
- EXDSP=YES must have been specified in the DSLPRM parameter module.

Job Control Statements

The following figure shows the MVS JCL to list the MERVA ESA user file.

```
//.......  JOB  ...........
//REXXB    EXEC PGM=DSLAREXX,REGION=8M,
//             PARM='DSLBA15R,parm1 parm2 parm3'
//*
//*           .. MERVA ESA LOAD LIBRARY
//STEPLIB  DD DSN=loadlib,DISP=SHR
//*
//*           .. ON THIS PDS: DSLBA15R
//SYSEXEC  DD DSN=samplib,DISP=SHR
//*
//*           .. LISTING DATASET (VB136)
//SYSTSPRT DD DSN=listds,DISP=OLD
//
```

Figure 24. MVS JCL to Run Batch Utility DSLBA15R - List User File

In the JCL, the lowercase parameters have the following meanings:

loadlib	The name of the load library containing the MERVA ESA programs.
samplib	The name of the library containing the program DSLBA15R.
listds	The name of the listing data set. Must be preallocated, record format VB, logical record length 136 recommended.

Runtime Parameters

The following parameters can be specified in the PARM field of the EXEC statement:

parm1
User ID or user ID pattern. parm1 may be one of:

- A user ID, for example MYUSER. The user file record of this user ID will be listed.
- A string with a trailing ‘*’, for example MYUS*. The user file records of all user IDs starting with the specified pattern will be listed.
- Not specified or ‘*’. All user file records will be listed.

parm2
Output format

F1 List all values of the user file. This is the default.

F2 The following values are listed in line format:

1. User ID
2. User name
3. Origin ID
4. User type (B, K, L, ..)
5. FLM administrator

6. Group ID (if DSLPRM USGRP=YES)

parm3 Log level. From **1** (basic) to **4** (all). The default is 2.

Customization

In the MERVA ESA customization module DSLPRM you can set the following parameter:

prtname Your institution name as it should appear in the printout of (most) REXX batch utilities.

Sample Printout of User File

The following figure shows the information printed after the execution of the DSLBA15R utility.f

```
MERVA ESA V4.1 DSLBA15R                      11. Oct. 1999  13:41:54
(C) Copyright IBM Corp. 1997, 1999

+ ----------------------------------------------------------- +
|         S A M P L E   B A N K   B o e b l i n g e n         |
+ ----------------------------------------------------------- +


DDDDD    SSSSSS   LLL      BBBBB     AAAA      111   555555   RRRRR
DDDDDD   SSSSSS   LLL      BBBBBB   AAAAAA    1111   555555   RRRRRR
DD  DD   SS       LLL      BB  BB   AA  AA   11111   55       RR  RR
DD  DD   SS       LLL      BB  BB   AA  AA     111   55       RR  RR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA     111   55555    RRRRRR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA     111   555555   RRRRRR
DD  DD       SS   LLL      BB  BB   AA  AA     111       55   RR  RR
DD  DD       SS   LLL      BB  BB   AA  AA     111       55   RR  RR
DDDDDD   SSSSSS   LLLLLL   BBBBBB   AA  AA     111   555555   RR  RR
DDDDD    SSSSSS   LLLLLL   BBBBB    AA  AA     111   55555    RR  RR


Print   M E R V A   E S A   U s e r   F i l e .


DSLBA15R_001I : Runtime parameter 'User ID' ..... : MAS1

DSLBA15R_002I : Runtime parameter 'Output format' : F1
                Allowed formats are: F1, F2, and '*'.

DSLBA15R_003I : Runtime parameter 'Log level' ... : 1
                Allowed log levels are: 1 .. 4, and '*'.

DSLBA15R_011I : User file record of User Id MAS1.

------------------------------------------------------------------------------
```

Figure 25. Printout of the DSLBA15R Utility (Part 1 of 2)

```
User ID ............ : MAS1
Name ............... : MASTER USER 1
Origin ID .......... : VNDEBET2AXXX
Group ID ........... :
User type .......... : M
FLM administrator ... : YES
Language ID ........ : E
Noprompt line format  : W
Default network ..... : S
User functions ...... : CMD      MSC      USR      FLM      L1RFINN  L1DE0
                        L1AI0    L1VE0    L1ACK    L1PR0    L2DE0    L3*
                        TX2USESQ USEERROR
Allowed message types : ********
Unauthorized commands : QSWITCH QW
User data 1 ......... : USER DATA LINE 1
User data 2 ......... : USER DATA LINE 2

PF-key-set name ..... :
Rejected Sign-Ons ... : 0
Password ............ : --------
Date of last Pw chg   : 1999/08/09
Time of last Pw chg   : 12:44:07
Date of last update   : 1999/10/04
Time of last update   : 12:32:55
Update user ID ...... : MAS1
Date of last sign-on  : 1999/10/11

------------------------------------------------------------------------------

DSLBA15R_018I : DSLBA15R ended with return code 0 - successful.
                Total processing time was 0.34 seconds.
```

Figure 25. Printout of the DSLBA15R Utility (Part 2 of 2)

DSLBA16R - Print Cross Reference Function Names - Allowed User IDs

You can use the batch utility DSLBA16R to print a cross reference of function names and allowed user IDs from the user file. Dependencies:

- MERVA ESA must be active.
- EXDSP=YES must have been specified in the DSLPRM parameter module.

Job Control Statements

The following figure shows the MVS JCL to print the cross-reference.

```
//.......  JOB  ...........
//REXXB    EXEC PGM=DSLAREXX,REGION=8M,
//              PARM='DSLBA16R,parm1 parm2 parm3'
//*
//*           .. MERVA ESA LOAD LIBRARY
//STEPLIB  DD DSN=loadlib,DISP=SHR
//*
//*           .. ON THIS PDS: DSLBA16R
//SYSEXEC  DD DSN=samplib,DISP=SHR
//*
//*           .. LISTING DATASET (VB136)
//SYSTSPRT DD DSN=listds,DISP=OLD
//
```

Figure 26. MVS JCL to Run Batch Utility DSLBA16R

In the JCL, the lowercase parameters have the following meanings:

loadlib	The name of the load library containing the MERVA ESA programs.
samplib	The name of the library containing the program DSLBA16R.
listds	The name of the listing data set. Must be preallocated, record format VB, logical record length 136 recommended.

Runtime Parameters

The following parameters can be specified in the PARM field of the EXEC statement:

parm1
: Function name. If no function name is specified, or specified with an ‘*’ (asterisk), then all function names are listed. Any other wildcard is used literally. For example if you specify ‘L1*’, a cross-reference ‘L1*’ and its allowed user IDs is printed, not a a cross-reference for L1AI0, L1DE0, L1VE0, ··.

parm2
: Output format

F1	Print each user ID on a separate line
F2	Like F1, plus a list with all user IDs
F3	Print six user IDs on each line
F4	Like F3, plus a list with all user IDs.

parm3
: Log level. From **1** (basic) to **4** (all). The default is 2.

Customization

In the MERVA ESA customization module DSLPRM you can set the following parameter:

prtname	Your institution name as it should appear in the printout of (most) REXX batch utilities.

Sample Printout

The following figure shows the information printed after the execution of the DSLBA16R utility.

```
MERVA ESA V4.1 DSLBA16R                      23. Oct. 1999  12:43:44
(C) Copyright IBM Corp. 1997, 1999

+ ---------------------------------------------------------- +
|         S A M P L E   B A N K   B o e b l i n g e n        |
+ ---------------------------------------------------------- +


DDDDD    SSSSSS   LLL      BBBBB     AAAA      111    6666    RRRRR
DDDDDD   SSSSSS   LLL      BBBBBB   AAAAAA    1111   666666   RRRRRR
DD  DD   SS       LLL      BB  BB   AA  AA   11111   66       RR  RR
DD  DD   SS       LLL      BB  BB   AA  AA     111   66       RR  RR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA     111   66666    RRRRRR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA     111   666666   RRRRRR
DD  DD       SS   LLL      BB  BB   AA  AA     111   66  66   RR  RR
DD  DD       SS   LLL      BB  BB   AA  AA     111   66  66   RR  RR
DDDDDD   SSSSSS   LLLLLL   BBBBBB   AA  AA     111   666666   RR  RR
DDDDD    SSSSSS   LLLLLL   BBBBB    AA  AA     111    6666    RR  RR


Print   M E R V A   E S A   F u n c t i o n   N a m e s   and  which
User IDs are allowed to each of them.
The list is sorted by Function name, within the Function by User ID.
User IDs in pending state and their assigned functions are ignored.


DSLBA16R_001I : Runtime parameter 'Function name' : *

DSLBA16R_002I : Runtime parameter 'Output format' : F1
                Allowed formats are: F1 .. F4, and '*'.

DSLBA16R_003I : Runtime parameter 'Log level' ... : 1
                Allowed log levels are: 1 .. 4, and '*'.
```

Figure 27. Printout of the DSLBA16R Utility (Part 1 of 2)

```
----------------------------------------------------------------------

DSLBA16R_010I : All assigned Function Names with their allowed User IDs.

  no.  Function     no.  User ID   Origin ID
-----  --------   -----  --------  ------------

    1  AUT            1  CBROWN    VNDEBET2AXXX
                      2  EJONES    VNDEBET2AXXX
                      3  GSMITH    VNDOBET2AXXX
                      4  MAS1      VNDEBET2AXXX

    2  CMD            1  CBROWN    VNDEBET2AXXX
                      2  EJONES    VNDEBET2AXXX
                      3  GSMITH    VNDOBET2AXXX
                      4  IFTADMIN
                      5  MASBOP    VNDEBET2AXXX
                      6  MASHEGO   VNDOBET2AXXX
                      7  MASMLINK  VNDEBET2AXXX
                      8  MASSWIFT  VNDEBET2AXXX
                      9  MAS1      VNDEBET2AXXX
                     10  MAS2      VNDEBET2AXXX

    3  EKAIBZCQ       1  IFTADMIN
                      2  MAS1      VNDEBET2AXXX

⋮



----------------------------------------------------------------------

DSLBA16R_013I : Total number of records in user file   : 23
                Total number of users in pending state : 2
                Total number of Function names used    : 73

DSLBA16R_014I : DSLBA16R ended with return code 0 - successful.
                Total processing time was 1.88 seconds.
```

Figure 27. Printout of the DSLBA16R Utility (Part 2 of 2)

DSLBA17R - Check Date Fields in the User File

You can use the batch utility DSLBA17R to print the
- last sign-on date
- password change date
- last User file update date

with elapsed days since today of all or specified User file records. You can flag those records where the number of elapsed days is greater than a specified value. This can be used, for example, to list user IDs that did not sign-on to MERVA since *nnn* days or did not change their password since *nnn* days.

DSLBA17R runs also against User file records processed with previous MERVA releases. If it finds 2-digit year dates from those releases, it will interpret the date according to the SWIFT rules.

Dependencies:
- MERVA ESA must be active.
- EXDSP=YES must have been specified in the DSLPRM parameter module.
- The last sign-on date in the User file is only maintained by MERVA when a SONNUM value greater than 0 (zero) has been specified in the DSLPRM parameter module.

Job Control Statements

The following figure shows the MVS JCL to list the dates of the MERVA ESA User file.

```
//.......  JOB  ...........
//REXXB    EXEC PGM=DSLAREXX,REGION=8M,
//              PARM='DSLBA17R,parm1 parm2 parm3 parm4 parm5 parm6
//              ... parm9'
//*
//*          .. MERVA ESA LOAD LIBRARY
//STEPLIB  DD DSN=loadlib,DISP=SHR
//*
//*          .. ON THIS PDS: DSLBA17R
//SYSEXEC  DD DSN=samplib,DISP=SHR
//*
//*          .. LISTING DATASET (VB136)
//SYSTSPRT DD DSN=listds,DISP=OLD
//
```

Figure 28. DSLBA17R (Check User File Dates) Sample JCL (MVS)

In the JCL, the lowercase parameters have the following meanings:

loadlib	The name of the load library containing the MERVA ESA programs.
samplib	The name of the library containing the program DSLBA17R.
listds	The name of the listing data set. Must be preallocated, record format VB, logical record length 136 recommended.

Runtime Parameters

The following parameters can be specified in the PARM field of the EXEC statement:

parm1	User ID or user ID pattern. parm1 may be one of: • A user ID, for example MYUSER. The dates of the User file record of this user ID will be listed. • A string with a trailing '*', for example MYUS*. The dates of the User file records of all user IDs starting with the specified pattern will be listed. • '*'. The dates of all User file records will be listed.
parm2	SO ('last sign-on date')
parm3	Number of days for SO ('last sign-on date'). The meaning of 'Number of days' is described in detail below.
parm4	PW ('last password change date')
parm5	Number of days for PW ('last password change date'). The meaning of 'Number of days' is described in detail below.
parm6	UF ('last User file change date')
parm7	Number of days for UF ('last User file change date'). The meaning of 'Number of days' is described in detail below.
parm8	List level **ALL** List all (matching) user IDs and flag those that are within one of the specified dates. ALL is forced when only one user ID is to be processed. **ONLY** List only those (matching) user IDs that are within one of the specified dates.
parm9	Log level. From **1** (basic) to **4** (all). The default is 2.

The meaning of the **Number of days** is as follows:

- The Number of days is a positive number

  Flag all users who signed on / changed their password / with changed User file within the last *nnn* days
- The Number of days is a negative number

  Flag all users who did not sign on / did not change their password / with unchanged User file within the last *nnn* days
- The Number of days is 0 (zero)

  The last sign-on date / password change date / User file change date is not used as criterion to flag a record.

To list, for example, all MAS user IDs that have not signed on for 60 days or have not changed their password for 150 days, you would specify:

```
PARM='DSLBA17R,MAS* SO -60 PW -150 UF 0 ALL 2'
```

Customization

In the MERVA ESA customization module DSLPRM you can set the following parameter:

prtname	Your institution name as it should appear in the printout of (most) REXX batch utilities.

Sample Printout of User File Dates Report

The following figure shows the information printed after the execution of the DSLBA17R utility.

```
MERVA ESA V4.1 DSLBA17R                      16. Jul. 1999  11:28:47
(C) Copyright IBM Corp. 1999

+ ---------------------------------------------------------- +
|         S A M P L E   B A N K   B o e b l i n g e n        |
+ ---------------------------------------------------------- +


DDDDD    SSSSSS   LLL      BBBBB     AAAA      111   777777   RRRRR
DDDDDD   SSSSSS   LLL      BBBBBB   AAAAAA    1111   777777   RRRRRR
DD  DD   SS       LLL      BB  BB   AA  AA   11111      77    RR  RR
DD  DD   SS       LLL      BB  BB   AA  AA     111      77    RR  RR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA     111     77     RRRRR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA     111     77     RRRRRR
DD  DD       SS   LLL      BB  BB   AA  AA     111    77      RR  RR
DD  DD       SS   LLL      BB  BB   AA  AA     111    77      RR  RR
DDDDDD   SSSSSS   LLLLLL   BBBBBB   AA  AA     111   77       RR  RR
DDDDD    SSSSSS   LLLLLL   BBBBB    AA  AA     111   77       RR  RR


Print   M E R V A   E S A   U s e r   F i l e   dates:
Last Sign-on date, Password change, User file change.

Note: The last sign-on date is stored only if your installation
      specified the DSLPRM parameter SONNUM greater than 0.


DSLBA17R_001I : Specified runtime parameter line:
                MAS*  SO -60  PW -150  UF 0  ALL 3

DSLBA17R_002I : Runtime parameters:

                1. User ID (pattern) ........ : MAS*
                2. No. of days SO - Sign-On   : -60
                3. No. of days PW - Password  : -150
                4. No. of days UF - user file : 0
                5. List level ............... : ALL
                6. Log level ................ : 2

DSLBA17R_003I : DSLBA17R will flag/list all user file records
                matching the specified user ID pattern MAS*
                - where the user has not signed-on for 60 days
                - where the user has not changed the password for 150 days

DSLBA17R_005I : Current DSLPRM settings:

                CVT extension, MVS only (CVTEXTO) : 124
                MERVA identifier ...... (DSLID)   : MHEG
                MERVA name ............ (NAME)    : MERVAESA

                Note: These are the values of the first DSLPRM module
                      found in the STEPLIB concatenation.

DSLBA17R_008I : User file record(s) and the last date
                - the user signed-on
                - the user changed the password
                - the user file record was changed.
```

Figure 29. DSLBA17R (Check User File Dates) Sample Printout (Part 1 of 2)

```
------------------------------------------------------------------------------··

                                     Last Sign-On      Password Change   User ··
F No.   User ID  User name           YYYY/MM/DD Days   YYYY/MM/DD Days   YYYY/
- ----- -------- ------------------  ---------- ----   ---------- ----   -----
>     1 MASGCH   MASTER USER GCH                   *  1997/09/15  304*  1997/
>     2 MASHUS   MASTER USER HUS     1998/04/14   93*  1998/04/14   93   1998/
      3 MAS0     MASTER USER 0       1998/07/07    9   1998/04/02  105   1998/
      4 MAS1     MASTER USER 1       1998/07/15    1   1998/02/27  139   1998/
      5 MAS2     MASTER USER 2       1998/07/07    9   1998/02/27  139   1998/
>     6 MAS3     MASTER USER 3       1998/04/15   92*  1998/02/27  139   1998/
>     7 MAS4     MASTER USER 4       1998/05/06   71*  1998/02/27  139   1998/
>     8 MAS5     MASTER USER 5       1998/07/16    0   1998/01/21  176*  1998/
>     9 MAS6     MASTER USER 6       1998/04/14   93*  1998/01/21  176*  1998/

------------------------------------------------------------------------------··

DSLBA17R_010I : Number of user file records processed : 9
                Number of user file records flagged   : 6
                Users not signed-on for 60 days   ... : 5
                No password change for 150 days   ... : 3

DSLBA17R_009I : DSLBA17R ended with return code 0 - successful.
                Total processing time was 1.18 seconds.
```

Figure 29. DSLBA17R (Check User File Dates) Sample Printout (Part 2 of 2)

All User file records where the number of days is greater than specified are flagged with '>' under heading F (flag) and an '*' (asterisk) is printed right next to the number of days.

DSLBA50R - Print Queue Status List

You can use the batch utility DSLBA50R to list the MERVA ESA queues and their status in batch. The queues can be listed in alphabetical order or in descending order of the number of messages in the queue. Dependencies: MERVA ESA must be active.

The program shows how an API program can use the DQ command and the DQSORTED command to retrieve the names of all or specified MERVA ESA queues together with their status.

Job Control Statements

The following figure shows the MVS JCL to print a queue status list.

```
//.......  JOB  ...........
//REXXB    EXEC PGM=DSLAREXX,REGION=8M,PARM=DSLBA50R
//*
//SYSTSIN  DD *
     * Comments start with '*' and ';'
     QUEUE    = cccccccc      ; Queue pattern
     DISPLAY  = ccccccc       ; Display mode
     LOGLEVEL = n             ; Log level
/*
//*
//*          .. MERVA ESA LOAD LIBRARY
//STEPLIB  DD DSN=loadlib,DISP=SHR
//*
//*          .. ON THIS PDS: DSLBA50R
//SYSEXEC  DD DSN=samplib,DISP=SHR
//*
//*          .. LISTING DATASET (VB84)
//SYSTSPRT DD DSN=listds,DISP=OLD
//
```

Figure 30. DSLBA50R (Print Queue Status) Sample JCL (MVS)

In the JCL, the lowercase parameters have the following meanings:

loadlib	The name of the load library containing the MERVA ESA programs.
samplib	The name of the library containing the program DSLBA50R.
listds	The name of the listing data set. Must be preallocated, record format VB, logical record length 84 recommended.

Runtime Parameters

The runtime parameters are passed to DSLBA50R via SYSTSIN under MVS and via SYSIPT under VSE. They have the form KEYWORD = VALUE. Each pair must be coded on a separate line. The input is folded to uppercase and leading and trailing blanks are stripped off from the specified keyword value. Lines starting with a '*' are treatened as comments, a ';' starts a line comment.

Keyword	Descr.	Possible values
QUEUE	Queue pattern	Queue pattern. If specified as '*', all queue names are listed. DSLBA50R accepts the same queue pattern as the DQ command (Display the Queue Status). This parameter is required.

Keyword	Descr.	Possible values	
DISPLAY	Display mode	**ALL**	All (matching) queues are listed
		FILled	Only (matching) queues that contain messages are listed.
		FILLED*	Only (matching) queues that contain messages are listed. The list includes also hidden queues.
		SORTed	Only (matching) queues that contain messages are listed. The queues are listed in descending order of the number of messages in the queue.
		This parameter is optional, the default value used is ALL.	
LOGLEVEL	Log level	Log level from 1 (basic) to 4 (all). This parameter is optional, the default is 2. You should use 4 in case of problems only.	

Customization

In the MERVA ESA customization module DSLPRM you can set the following parameter:

prtname Your institution name as it should appear in the printout of (most) REXX batch utilities.

Sample Printout of Print Queue Status List

The following figure shows the information printed after the execution of the DSLBA50R utility. The headings used are the same as with the DQ command, with the exception that leading zeros are suppressed and the keys defined are shown as 1, 2, or 1 and 2.

```
MERVA ESA V4.1 DSLBA50R                        13. Aug. 1999  13:32:10
(C) Copyright IBM Corp. 1999

+ ---------------------------------------------------------- +
|         S A M P L E   B A N K   B o e b l i n g e n        |
+ ---------------------------------------------------------- +


DDDDD    SSSSSS   LLL      BBBBB     AAAA    555555    0000    RRRRR
DDDDDD   SSSSSS   LLL      BBBBBB   AAAAAA   555555   000000   RRRRRR
DD  DD   SS       LLL      BB  BB   AA  AA   55       00  00   RR  RR
DD  DD   SS       LLL      BB  BB   AA  AA   55       00  00   RR  RR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA   55555    00  00   RRRRR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA   555555   00  00   RRRRRR
DD  DD       SS   LLL      BB  BB   AA  AA       55   00  00   RR  RR
DD  DD       SS   LLL      BB  BB   AA  AA       55   00  00   RR  RR
DDDDDD   SSSSSS   LLLLLL   BBBBBB   AA  AA   555555   000000   RR  RR
DDDDD    SSSSSS   LLLLLL   BBBBB    AA  AA   55555     0000    RR  RR


Print a   Q u e u e   S t a t u s   L i s t   of all or specified queues.


DSLBA50R_002I : Runtime parameters:
                1. QUEUE    - Queue pattern : L*
                2. DISPLAY  - Display mode  : ALL
                3. LOGLEVEL - Log level ... : 1

DSLBA50R_005I : Performed command is: DQ L*

DSLBA50R_007I : Total number of queue names processed: 79

DSLBA50R_008I : Number  Function  S  KEY  USR  WAIT    THRSH  T
                ------  --------  -  ---  ---  ------  -----  -
                     1  L1ACK        1,2    0       0    100
                     2  L1AI0        1      0       2     50
                     3  L1AO0        1,2    0       0    100
                     4  L1CES               0       0    100
                     5  L1CESI    N         0       0    100
                     6  L1CSE     N  1,2    0       0    100
                     7  L1CSEO              0       0    100
                     8  L1DE0        1      0      10     50
                     9  L1DO0        1      0     352    100  T
                    10  L1ERROR             0       0    100
                    11  L1FREE              0       0    100
                    12  L1PR0     N         0       0    100
                    13  L1PR1     N         0       0    100
                    14  L1RFINN             0       0     30
                    15  L1RFINU             0       0     30
⋮

                    79  L3VE0        1,2    0       0     20

DSLBA50R_011I : DSLBA50R ended with return code 0 - successful.
                Total processing time was 0.54 seconds.
```

Figure 31. DSLBA50R (Print Queue Status) Sample Printout

DSLBA51R - Print Queue Key List

You can use the batch utility DSLBA51R to print a queue key list of all or specified queues. Optionally only queue elements are listed matching a specified KEY1 and/or KEY2 value. This effectively searches for a key value over the queue data set. Dependencies: MERVA ESA must be active.

Note: DSLBA51R issues a DQ *queue_pattern* FILLED command and then loops through the returned queue names.

Job Control Statements

The following figure shows the MVS JCL to print a queue key list in batch.

```
//.......  JOB  ............
//REXXB    EXEC PGM=DSLAREXX,REGION=8M,PARM=DSLBA51R
//*
//SYSTSIN  DD *
     * Comments start with '*' and ';'
     QUEUE    = cccccccc      ; Queue name (pattern)
     KEY1     = ccccccccc..   ; Key 1 value
     KEY2     = ccccccccc..   ; Key 2 value
     LOGLEVEL = n             ; Log level
/*
//*
//*           .. MERVA ESA LOAD LIBRARY
//STEPLIB  DD DSN=loadlib,DISP=SHR
//*
//*           .. ON THIS PDS: DSLBA51R
//SYSEXEC  DD DSN=samplib,DISP=SHR
//*
//*           .. LISTING DATASET (VB136)
//SYSTSPRT DD DSN=listds,DISP=OLD
//
```

Figure 32. DSLBA51R (Print Queue Key List) Sample JCL (MVS)

Runtime Parameters

The runtime parameters are passed to DSLBA51R via SYSTSIN under MVS and via SYSIPT under VSE. They have the form KEYWORD = VALUE. Each pair must be coded on a separate line. The input is folded to uppercase with the exception of the entered KEY1 and KEY2 value and leading and trailing blanks are stripped off from the specified keyword value. Lines starting with a '*' are treatened as comments, a ';' starts a line comment.

Keyword	Descr.	Possible values
QUEUE	Queue name	Queue name or queue name pattern. If specified as '*', all queues are processed. DSLBA51R accepts the same queue pattern as the DQ command (Display the Queue Status). This parameter is required.
KEY1	Key 1 value	Key 1 value to be matched. This parameter is optional.
KEY2	Key 2 value	Key 2 value to be matched. This parameter is optional.
LOGLEVEL	Log level	Log level from 1 (basic) to 4 (all). This parameter is optional, the default is 2. You should use 4 in case of problems only.

Notes:

1. DSLBA51R does not translate the entered key values to uppercase, the values are taken exactly as entered.
2. The keys can be generic, that is, they can contain wildcards:

'%'	matches any single character
'*'	matches any number of characters, including no characters.

3. Specify a '_' (underscore) for any leading, trailing, or imbedded blank in a key value. To list, for example, all L1* queues that have a key 1 value of '1998 ABC 001', you would specify:

```
QUEUE = L1*
KEY1  = 1998_ABC_001
```

4. Special characters like ampersands can be entered as they are, that is, there is no need to enclose them in quotes or paranthesis or to double them. To list, for example, all L1* queues that have a key-1 value that contains 'O'Smith', you would specify:

```
QUEUE = L1*
KEY1  = *O'Smith*
```

Data Set Names

In the JCL, the lowercase data set names have the following meanings:

loadlib	The name of the load library containing the MERVA ESA programs.
samplib	The name of the library containing the program DSLBA51R.
listds	The name of the listing data set. Must be preallocated, record format VB, logical record length 136 recommended.

Customization

In the MERVA ESA customization module DSLPRM you can set the following parameter:

prtname	Your institution name as it should appear in the printout of (most) REXX batch utilities.

Sample Printout of Print Queue Key List

The following figure shows the information printed after the execution of the DSLBA51R utility.

```
MERVA ESA V4.1 DSLBA51R                    22. Jun. 1998  10:34:05
(C) Copyright IBM Corp. 1999

+ ---------------------------------------------------------- +
|         S A M P L E   B A N K   B o e b l i n g e n        |
+ ---------------------------------------------------------- +


DDDDD    SSSSSS   LLL      BBBBB     AAAA    555555     111   RRRRR
DDDDDD   SSSSSS   LLL      BBBBBB   AAAAAA   555555    1111   RRRRRR
DD  DD   SS       LLL      BB  BB   AA  AA   55       11111   RR  RR
DD  DD   SS       LLL      BB  BB   AA  AA   55         111   RR  RR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA   55555      111   RRRRR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA   555555     111   RRRRRR
DD  DD       SS   LLL      BB  BB   AA  AA       55     111   RR  RR
DD  DD       SS   LLL      BB  BB   AA  AA       55     111   RR  RR
DDDDDD   SSSSSS   LLLLLL   BBBBBB   AA  AA   555555     111   RR  RR
DDDDD    SSSSSS   LLLLLL   BBBBB    AA  AA   55555      111   RR  RR


Print a   Q u e u e   K e y   L i s t   of all or specified queues.
Optionally list only those queue elements matching a specified KEY1
and/or KEY2 value.


DSLBA51R_002I : Runtime parameters:
                1. QUEUE    - Queue name (pattern) : L1*
                2. KEY1     - Key 1 value ........ : 19980812*
                3. KEY2     - Key 2 value ........ :
                4. LOGLEVEL - Log level .......... : 2

DSLBA51R_004I : Current DSLPRM settings:

                MERVA identifier  (DSLID)   : MHEG
                MERVA name ...... (NAME)    : MERVAESA
                CVT extension ... (CVTEXTO) : 124

                Note: These are the values of the first DSLPRM module
                      found in the STEPLIB concatenation.

DSLBA51R_009I : Queue Key List for queue L1ACK

                Queue    QSN        B Key 1                    Key 2
                -------- ---------- - ------------------------ -----------...
                L1ACK    0000000144   19980812-0001            120036
                L1ACK    0000000145   19980812-0002            120037
                L1ACK    0000000146 X 19980812-0003            120038
                L1ACK    0000000147   19980812-0004            120039

DSLBA51R_009I : Queue Key List for queue L1DE0

                Queue    QSN        B Key 1                    Key 2
                -------- ---------- - ------------------------ -----------...
                L1DE0    0000000179   19980812-0001
                L1DE0    0000000180   19980812-0002
                L1DE0    0000000182   19980812-0003
                L1DE0    0000000183   19980812-0004
                L1DE0    0000000184   19980812-0005
```

Figure 33. DSLBA51R (Print Queue Key List) Sample Printout (Part 1 of 2)

```
                 L1DE0    0000000185   19980812-0006
                 L1DE0    0000000186   19980812-0007
                 L1DE0    0000000188   19980812-0008
⋮


DSLBA51R_011I : Number of queues processed : 6

                                (Matching)
                Number Queue    Elements
                ------ -------- --------
                     1 L1ACK           4
                     2 L1DE0           8
                     3 L1DO0           0
                     4 L1PR0           0
                     5 L1PR1           0
                     6 L1VE0          10

DSLBA51R_010I : DSLBA51R ended with return code 0 - successful.
                Total processing time was 0.33 seconds.
```

Figure 33. DSLBA51R (Print Queue Key List) Sample Printout (Part 2 of 2)

Listing Fields

The ‘Queue Key List’ of the listing contains the following information:

Queue	Queue name
QSN	Queue sequence number
B	Shows that the message is currently in use by another user (‘BUSY’)
Key 1	Key 1 of the message
Key 2	Key 2 of the message.

DSLBA52R - Copy or Move Messages (and Sort by Key)

You can use the batch utility DSLBA52R to copy or move messages from one queue to another. Optionally the messages can be written to the output queue in order of their key 1 and/or key 2 value of the *input* queue, that is, the messages can be sorted. As the program bypasses the normal message flow as defined by the function table and the routing tables, the influence on security considerations has to be evaluated carefully.

Dependencies: MERVA ESA must be active.

Job Control Statements

The following figure shows the MVS JCL to copy or move messages from one queue to another.

```
//.......  JOB  ...........
//REXXB    EXEC PGM=DSLAREXX,REGION=0K,PARM=DSLBA52R
//*
//*           .. RUNTIME PARAMETERS
//SYSTSIN  DD  *
  * Comments start with '*' and ';'
  * HELP
  * -- Required parameters --
  FUNCTION  = cccccc       ; Function (COPY, LIST, or MOVE)
  QUEUE1    = cccccccc     ; Input queue name
  QUEUE2    = cccccccc     ; Output queue name
  * -- Optional parameters --
  BUSY      = ccccccc      ; BUSY disposition (PROCESS or SKIP)
  FROMQSN   = nnnnnnnnnn   ; From QSN
  LOGLEVEL  = n            ; Log level 1 .. 4
  SORT      = cccc         ; Sort order (ASIS, K1, K12, K2, or K21)
  TOQSN     = nnnnnnnnnn   ; To QSN
/*
//*
//*           .. MERVA ESA LOAD LIBRARY
//STEPLIB  DD DSN=loadlib,DISP=SHR
//*
//*           .. ON THIS PDS: DSLBA52R
//SYSEXEC  DD DSN=samplib,DISP=SHR
//*
//*           .. LISTING DATASET (VB136)
//SYSTSPRT DD DSN=listds,DISP=OLD
//
```

Figure 34. DSLBA52R (Copy or Move Messages) Sample JCL (MVS)

Runtime Parameters

The runtime parameters are passed to DSLBA52R via SYSTSIN under MVS and via SYSIPT under VSE. They have the form KEYWORD = VALUE. Each pair must be coded on a separate line. The input is folded to uppercase and leading and trailing blanks are stripped off from the specified keyword value. Lines starting with a '*' are treatened as comments, a ';' starts a line comment.

Required parameters:

Keyword	Descr.	Possible values
FUNCTION	Function to be performed	**COPY** Copy the queue elements from the input queue QUEUE1 to the output queue QUEUE2 **LIST** List the queue elements of queue QUEUE1 **MOVE** Move the queue elements from the input queue QUEUE1 to the output queue QUEUE2 This parameter is required.
QUEUE1	Input queue name	The name of the input queue. The messages of this queue will be processed. The entered input queue name can be further checked in user exit USEREXIT_Q1 of DSLBA52R. This parameter is required.
QUEUE2	Output queue name	The name of the output queue. The entered output queue name can be further checked in user exit USEREXIT_Q2 of DSLBA52R. This parameter is required, unless function LIST is used.

Optional parameters:

Keyword	Descr.	Possible values
BUSY	BUSY dispo sition	Specifies what happens when a message is BUSY. If specified, the keyword value must be one of: **PROCess** The message is processed (copied or moved). **SKIP** The message is not processed. This parameter is optional. If omitted, the default value used is SKIP.
FROMQSN	From QSN	Only messages with a QSN greater than or equal to this QSN will be processed. This parameter is optional. If omitted, the default value used is 0. See also TOQSN.
LOGLEVEL	Log level	**1** Only overview data is shown in the listing. **2** Detailed data for each message is shown. When parameter SORT = ASIS is specified, (only) one line is printed per message. **3** Detailed data for each message is shown. **4** Should be used in case of problems only. This parameter is optional. If omitted, the default value used is 2.

Keyword	Descr.	Possible values
SORT	SORT order	Specifies the order in which the messages are written to the target queue. If specified, the keyword value must be one of: **ASIS** Write the messages in their existing (QSN) sequence **K1** Write the messages sorted by their (input queue) key–1 value **K12** Write the messages sorted by their (input queue) key–1 and key–2 value **K2** Write the messages sorted by their (input queue) key-2 value **K21** Write the messages sorted by their (input queue) key–2 and key–1 value This parameter is optional. If omitted, the default value used is ASIS.
TOQSN	To QSN	Only messages with a QSN less than or equal to this QSN will be processed. This parameter is optional. If omitted, all messages are processed. See also FROMQSN.

HELP as the only parameter prints a description of the runtime parameters.

Data Set Names

In the JCL, the lowercase data set names have the following meanings:

loadlib	The name of the load library containing the MERVA ESA programs.
samplib	The name of the library containing the program DSLBA52R.
listds	The name of the listing data set. Must be preallocated, record format VB, logical record length 136 recommended.

Customization

In the MERVA ESA customization module DSLPRM you can set the following parameter:

prtname	Your institution name as it should appear in the printout of (most) REXX batch utilities.

You can use the following routines in DSLBA52R to reject entered runtime parameters:

1. USEREXIT_Q1 can be used to reject the entered value for runtime parameter QUEUE1, input queue.
2. USEREXIT_Q2 can be used to reject the entered value for runtime parameter QUEUE2, output queue.

Sample Queue Key List after Run of DSLBA52R

The following figure shows the queue key list of input queue L1DE0 and output queue L1SRT after DSLBA52R copied the messages in key–2 / key–1 order.

Note: The blanks in key 2 after the date and the currency have been inserted for easier reading. They are not part of the data.

```
Queue L1DE0 (Input)                  Queue L1SRT (Output, sorted by K21)

No.  Key 1     Key 2                 No.  Key 1     Key 2
---  --------  ------------------    ---  --------  ------------------

  1  TRN-0001  991010 AUD 1000,00      1  TRN-0001  991010 AUD 1000,00
  2  TRN-0002  991010 CHF 1000,00      2  TRN-0006  991010 AUD 2000,00
  3  TRN-0003  991010 DEM 1000,00      3  TRN-0011  991010 AUD 3000,00
  4  TRN-0004  991010 GBP 1000,00      4  TRN-0016  991010 AUD 4000,00
  5  TRN-0005  991010 USD 1000,00      5  TRN-0021  991010 AUD 5000,00
  6  TRN-0006  991010 AUD 2000,00      6  TRN-0026  991010 AUD 6000,00
  7  TRN-0007  991010 CHF 2000,00      7  TRN-0031  991010 AUD 7000,00
  8  TRN-0008  991010 DEM 2000,00      8  TRN-0036  991010 AUD 8000,00
  9  TRN-0009  991010 GBP 2000,00      9  TRN-0002  991010 CHF 1000,00
 10  TRN-0010  991010 USD 2000,00     10  TRN-0007  991010 CHF 2000,00

 11  TRN-0011  991010 AUD 3000,00     11  TRN-0012  991010 CHF 3000,00
 12  TRN-0012  991010 CHF 3000,00     12  TRN-0017  991010 CHF 4000,00
 13  TRN-0013  991010 DEM 3000,00     13  TRN-0022  991010 CHF 5000,00
 14  TRN-0014  991010 GBP 3000,00     14  TRN-0027  991010 CHF 6000,00
 15  TRN-0015  991010 USD 3000,00     15  TRN-0032  991010 CHF 7000,00
 16  TRN-0016  991010 AUD 4000,00     16  TRN-0037  991010 CHF 8000,00
 17  TRN-0017  991010 CHF 4000,00     17  TRN-0003  991010 DEM 1000,00
 18  TRN-0018  991010 DEM 4000,00     18  TRN-0008  991010 DEM 2000,00
 19  TRN-0019  991010 GBP 4000,00     19  TRN-0013  991010 DEM 3000,00
 20  TRN-0020  991010 USD 4000,00     20  TRN-0018  991010 DEM 4000,00

 21  TRN-0021  991010 AUD 5000,00     21  TRN-0023  991010 DEM 5000,00
 22  TRN-0022  991010 CHF 5000,00     22  TRN-0028  991010 DEM 6000,00
 23  TRN-0023  991010 DEM 5000,00     23  TRN-0033  991010 DEM 7000,00
 24  TRN-0024  991010 GBP 5000,00     24  TRN-0038  991010 DEM 8000,00
 25  TRN-0025  991010 USD 5000,00     25  TRN-0004  991010 GBP 1000,00
 26  TRN-0026  991010 AUD 6000,00     26  TRN-0009  991010 GBP 2000,00
 27  TRN-0027  991010 CHF 6000,00     27  TRN-0014  991010 GBP 3000,00
 28  TRN-0028  991010 DEM 6000,00     28  TRN-0019  991010 GBP 4000,00
 29  TRN-0029  991010 GBP 6000,00     29  TRN-0024  991010 GBP 5000,00
 30  TRN-0030  991010 USD 6000,00     30  TRN-0029  991010 GBP 6000,00

 31  TRN-0031  991010 AUD 7000,00     31  TRN-0034  991010 GBP 7000,00
 32  TRN-0032  991010 CHF 7000,00     32  TRN-0039  991010 GBP 8000,00
 33  TRN-0033  991010 DEM 7000,00     33  TRN-0005  991010 USD 1000,00
 34  TRN-0034  991010 GBP 7000,00     34  TRN-0010  991010 USD 2000,00
 35  TRN-0035  991010 USD 7000,00     35  TRN-0015  991010 USD 3000,00
 36  TRN-0036  991010 AUD 8000,00     36  TRN-0020  991010 USD 4000,00
 37  TRN-0037  991010 CHF 8000,00     37  TRN-0025  991010 USD 5000,00
 38  TRN-0038  991010 DEM 8000,00     38  TRN-0030  991010 USD 6000,00
 39  TRN-0039  991010 GBP 8000,00     39  TRN-0035  991010 USD 7000,00
 40  TRN-0040  991010 USD 8000,00     40  TRN-0040  991010 USD 8000,00
```

Figure 35. DSLBA52R (Copy or Move Messages) Queue Key List Before and After (Part 1 of 2)

```
 41  TRN-0041  991012 AUD 1000,00      41  TRN-0041  991012 AUD 1000,00
 42  TRN-0042  991012 CHF 1000,00      42  TRN-0046  991012 AUD 2000,00
 43  TRN-0043  991012 DEM 1000,00      43  TRN-0042  991012 CHF 1000,00
 44  TRN-0044  991012 GBP 1000,00      44  TRN-0047  991012 CHF 2000,00
 45  TRN-0045  991012 USD 1000,00      45  TRN-0043  991012 DEM 1000,00
 46  TRN-0046  991012 AUD 2000,00      46  TRN-0048  991012 DEM 2000,00
 47  TRN-0047  991012 CHF 2000,00      47  TRN-0044  991012 GBP 1000,00
 48  TRN-0048  991012 DEM 2000,00      48  TRN-0049  991012 GBP 2000,00
 49  TRN-0049  991012 GBP 2000,00      49  TRN-0045  991012 USD 1000,00
 50  TRN-0050  991012 USD 2000,00      50  TRN-0050  991012 USD 2000,00
```

Figure 35. DSLBA52R (Copy or Move Messages) Queue Key List Before and After (Part 2 of 2)

Sample Printout of DSLBA52R

The following figure shows the information printed after the execution of the DSLBA52R utility.

```
MERVA ESA V4.1 DSLBA52R                        26. Nov. 1999  17:23:43
(C) Copyright IBM Corp. 1999

+ ----------------------------------------------------------- +
|         S A M P L E   B A N K   B o e b l i n g e n         |
+ ----------------------------------------------------------- +


DDDDD    SSSSSS   LLL      BBBBB     AAAA    555555    2222    RRRRR
DDDDDD   SSSSSS   LLL      BBBBBB   AAAAAA   555555   222222   RRRRRR
DD  DD   SS       LLL      BB  BB   AA  AA   55       22  22   RR  RR
DD  DD   SS       LLL      BB  BB   AA  AA   55           22   RR  RR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA   55555       22    RRRRR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA   555555     22     RRRRRR
DD  DD       SS   LLL      BB  BB   AA  AA       55    22      RR  RR
DD  DD       SS   LLL      BB  BB   AA  AA       55   22  22   RR  RR
DDDDDD   SSSSSS   LLLLLL   BBBBBB   AA  AA   555555   222222   RR  RR
DDDDD    SSSSSS   LLLLLL   BBBBB    AA  AA   55555    222222   RR  RR


Copy or move messages from one queue to another.
HELP as the only parameter prints a description.


DSLBA52R_001I : DSLBA52R started by user HEG at 26. Nov. 1999 17:23:43

DSLBA52R_003I : Runtime parameters:
                1. FUNCTION - Function ........ : COPY
                2. QUEUE1   - Input queue name  : L1DE0
                3. QUEUE2   - Output queue name : L1SRT
                4. BUSY     - BUSY disposition  : SKIP
                5. FROMQSN  - From QSN ........ :
                6. LOGLEVEL - Log level ....... : 2
                7. SORT     - Sort order ...... : K21
                8. TOQSN    - To QSN .......... :

DSLBA52R_006I : MERVA ID is MHEG, MERVA name is MERVAESA.

DSLBA52R_007I : Queue management routine defined is DSLQMCNV.
                The access features defined are VXBD.
```

Figure 36. DSLBA52R (Copy or Move Messages) Sample Printout (Part 1 of 3)

```
DSLBA52R_013I : Detailed statistical data
                (Copy from input queue L1DE0 to output queue L1SRT)

                                            GET
                Err  In no.   Input QSN   rc  BUSY  Key 1
                ---  -------  ----------  --  ----  -----------------------
                     Out no.  Output QSN  PUT_rc    Key 2
                     -------  ----------  --        -----------------------

                           1  0000000001  ok        TRN-0001
                           1  0000000501  ok        991010AUD1000,00

                           2  0000000002  ok        TRN-0002
                           9  0000000509  ok        991010CHF1000,00

                           3  0000000003  ok        TRN-0003
                          17  0000000517  ok        991010DEM1000,00

                           4  0000000004  ok        TRN-0004
                          25  0000000525  ok        991010GBP1000,00

                           5  0000000005  ok        TRN-0005
                          33  0000000533  ok        991010USD1000,00

                           6  0000000006  ok        TRN-0006
                           2  0000000502  ok        991010AUD2000,00

                           7  0000000007  ok        TRN-0007
                          10  0000000510  ok        991010CHF2000,00

                           8  0000000008  ok        TRN-0008
                          18  0000000518  ok        991010DEM2000,00

                           9  0000000009  ok        TRN-0009
                          26  0000000526  ok        991010GBP2000,00

                          10  0000000010  ok        TRN-0010
                          34  0000000534  ok        991010USD2000,00

                          11  0000000011  ok        TRN-0011
                           3  0000000503  ok        991010AUD3000,00

                          12  0000000012  ok        TRN-0012
                          11  0000000511  ok        991010CHF3000,00
.
.
.
DSLBA52R_014I : Overview statistical data

                No. of elements matching QSN range   : 50

                No. of GETs with intrc ' ' ......... : 50
                - thereof BUSY ..................... :  0
                No. of GETs with intrc 01 .......... :  0
                No. of GETs with intrc 02 .......... :  0
                No. of GETs with rc <= -2 .......... :  0

                No. of PUTs with intrc ' ' ......... : 50
                No. of PUTs with intrc 01 .......... :  0
                No. of PUTs with intrc 02 .......... :  0
                No. of PUTs with rc <= -2 .......... :  0
```

Figure 36. DSLBA52R (Copy or Move Messages) Sample Printout (Part 2 of 3)

```
DSLBA52R_015I : Number of (matching) input messages : 50
                Number of messages GET ok ......... : 50
                - thereof BUSY .................... : 0
                Number of messages GET failed ..... : 0
                Number of messages PUT ok ......... : 50
                Number of messages PUT failed ..... : 0

DSLBA52R_010I : DSLBA52R ended with return code 0 - successful.

DSLBA52R_016I : DSLBA52R ended at 26. Nov. 1999 17:23:47
```

Figure 36. DSLBA52R (Copy or Move Messages) Sample Printout (Part 3 of 3)

Listing Fields

The 'Detailed statistical data' of the listing contains the following information:

Err
A '>' indicates an error with the message, for example:
- The message could not be read
- The message is currently in use by another user ('BUSY')
- The copy or move failed

In no.
Running number of messages in input queue (before sort)

Out no.
Running number of messages in output queue (after sort)

Input QSN
Input queue QSN

Output QSN
Output queue QSN after a successful COPY or MOVE

GET rc
Return code of API function GET - 'ok' indicates that the message was successfully read from the input queue

PUT(B)_rc
Return code of API function PUT/PUTB - 'ok' indicates that the message was successfully copied/moved from the input queue to the output queue

BUSY
Shows that the message is currently in use by another user ('BUSY')

Key 1
Key 1 of the message

Key 2
Key 2 of the message.

DSLBA53R - Scan a Queue for ‘Old’ Messages

You can use the batch utility DSLBA53R to scan a MERVA ESA queue for messages that are as old or older than a specified number of days. Those messages can either be:
- Flagged in the output listing
- Deleted from the queue
- Moved to another queue.

This can be used, for example, to move old messages to an archive queue, or to delete old messages from a protocol queue. As the program bypasses the normal message flow as defined by the function table and the routing tables, the influence on security considerations has to be evaluated carefully.

To get the age of a message, DSLBA53R computes the number of elapsed days from the date of the first or the last MSGTRACE entry of the message until today. The 2-digit year date of the MSGTRACE is interpreted according to the SWIFT rules. If a message has no MSGTRACE entry, a warning message is issued.

Dependencies: MERVA ESA must be active.

Job Control Statements

The following figure shows the MVS JCL to scan a MERVA ESA queue for ‘old’ messages.

```
//.......  JOB  ...........
//REXXB    EXEC PGM=DSLAREXX,REGION=0K,PARM=DSLBA53R
//*
//*           .. RUNTIME PARAMETERS
//SYSTSIN  DD  *
  * Comments start with '*' and ';'
  * HELP
  * -- Required parameters --
  FUNCTION  = cccccc       ; Function (DELETE, LIST, or MOVE)
  BACKDAYS  = nnnn         ; Number of days
  QUEUE1    = cccccccc     ; Input queue name
  QUEUE2    = cccccccc     ; Output queue name (for MOVE)
  * -- Optional parameters --
  BUSY      = ccccccc      ; BUSY disposition (PROCESS or SKIP)
  FLDMODIF  = ccccc        ; FIRST or LAST msgtrace entry
  FROMQSN   = nnnnnnnnnn   ; From QSN
  LOGLEVEL  = n            ; Log level
  TOQSN     = nnnnnnnnnn   ; To QSN
/*
//*
//*           .. MERVA ESA LOAD LIBRARY
//STEPLIB  DD DSN=loadlib,DISP=SHR
//*
//*           .. ON THIS PDS: DSLBA53R
//SYSEXEC  DD DSN=samplib,DISP=SHR
//*
//*           .. LISTING DATASET (VB136)
//SYSTSPRT DD DSN=listds,DISP=OLD
//
```

Figure 37. DSLBA53R (Search for ‘Old’ Messages) Sample JCL (MVS)

Runtime Parameters

The runtime parameters are passed to DSLBA53R via SYSTSIN under MVS and via SYSIPT under VSE. They have the form KEYWORD = VALUE. Each pair must be coded on a separate line. The input is folded to uppercase and leading and trailing blanks are stripped off from the specified keyword value. Lines starting with a '*' are treatened as comments, a ';' starts a line comment.

Required parameters:

Keyword	Descr.	Possible values
FUNCTION	Function to be performed	**DELete** Delete the queue elements from the input queue **LIST** List the queue elements **MOVE** Move the queue elements from the input queue QUEUE1 to the output queue QUEUE2 This parameter is required.
BACKDAYS	Number of days	Only queue elements as old or older than *nnn* days are further processed. To determine the age of a message, the first or the last MSGTRACE entry (see parameter FDLMODIF) found is compared with today's date. With FUNCTION DELETE and MOVE the entered parameter is checked against the value specified in customization variable *backdays_limit*. The default value used is 7, that is, with FUNCTION DELETE and MOVE any BACKDAYS value less than 7 is rejected. This parameter is required.
QUEUE1	Input queue name	The name of the input queue. The messages of this queue will be processed. The entered input queue name can be further checked in user exit USEREXIT_Q1 of DSLBA53R. This parameter is required.
QUEUE2	Output queue name	The name of the output queue for FUNCTION MOVE. To this queue the messages that are 'too old' are written (moved). The entered output queue name can be further checked in user exit USEREXIT_Q2 of DSLBA53R. This parameter is required with FUNCTION = MOVE.

Optional parameters:

Keyword	Descr.	Possible values
BUSY	BUSY dispo sition	Specifies what happens when a message is BUSY. If specified, the keyword value must be one of: **PROCess** The message is processed (deleted or moved) **SKIP** The message is not processed. This parameter is optional. If omitted, the default value used is SKIP.

Keyword	Descr.	Possible values
FLDMODIF	MSGTRACE field READ modifier	**FIRST** The first MSGTRACE of a message is used to determine the age **LAST** The last MSGTRACE of a message is used to determine the age This parameter is optional. If omitted, the default value used is FIRST.
FROMQSN	From QSN	Only messages with a QSN greater than or equal to this QSN will be processed. This parameter is optional. If omitted, the default value used is 0. See also TOQSN.
LOGLEVEL	Log level	**1** Only overview data is shown in the listing. **2** Detailed data for each message is shown. **3** The key-1 and key-2 value for each message are shown. **4** Should be used in case of problems only. This parameter is optional. If omitted, the default value used is 2.
TOQSN	To QSN	Only messages with a QSN less than or equal to this QSN will be processed. This parameter is optional. If omitted, all messages are processed. See also FROMQSN.

HELP as the only parameter prints a description of the runtime parameters.

Data Set Names

In the JCL, the lowercase data set names have the following meanings:

loadlib	The name of the load library containing the MERVA ESA programs.
samplib	The name of the library containing the program DSLBA53R.
listds	The name of the listing data set. Must be preallocated, record format VB, logical record length 136 recommended.

Customization

In the MERVA ESA customization module DSLPRM you can set the following parameter:

prtname
Your institution name as it should appear in the printout of (most) REXX batch utilities.

In the CUSTOMIZATION SECTION of DSLBA53R you can set the following installation-specific variable:

backdays_limit
With FUNCTION DELETE and MOVE the entered value for the runtime parameter BACKDAYS is checked against the variable 'backdays_limit'. If the entered value is lower than the limit set, the entered parameter is rejected, and DSLBA53R will not start. The IBM supplied default value is 7 days.

You can use the following routines in DSLBA53R to reject entered runtime parameters:

1. USEREXIT_Q1 can be used to reject the entered value for runtime parameter QUEUE1, input queue.
2. USEREXIT_Q2 can be used to reject the entered value for runtime parameter QUEUE2, output queue for MOVE.

Sample Printout of DSLBA53R

The following figure shows the information printed after the execution of the DSLBA53R utility.

```
MERVA ESA V4.1 DSLBA53R                       1. Jul. 1999  12:26:22
(C) Copyright IBM Corp. 1999

+ ----------------------------------------------------------- +
|         S A M P L E   B A N K   B o e b l i n g e n         |
+ ----------------------------------------------------------- +


DDDDD    SSSSSS   LLL      BBBBB     AAAA    555555    3333    RRRRR
DDDDDD   SSSSSS   LLL      BBBBBB   AAAAAA   555555   333333   RRRRRR
DD  DD   SS       LLL      BB  BB   AA  AA   55       33  33   RR  RR
DD  DD   SS       LLL      BB  BB   AA  AA   55           33   RR  RR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA   55555     3333    RRRRR
DD  DD   SSSSSS   LLL      BBBBBB   AAAAAA   555555    3333    RRRRRR
DD  DD       SS   LLL      BB  BB   AA  AA       55       33   RR  RR
DD  DD       SS   LLL      BB  BB   AA  AA       55   33  33   RR  RR
DDDDDD   SSSSSS   LLLLLL   BBBBBB   AA  AA   555555   333333   RR  RR
DDDDD    SSSSSS   LLLLLL   BBBBB    AA  AA   55555     3333    RR  RR


Scan a MERVA queue for 'old' messages.
HELP as the only parameter prints a description.


DSLBA53R_001I : DSLBA53R started by user HEG at 1. Jul. 1999 09:42:12

DSLBA53R_003I : Runtime parameters:
                1. FUNCTION - Function ........ : DELETE
                2. BACKDAYS - Number of days .. : 10
                3. QUEUE1   - Input queue name  : L1DE0
                4. QUEUE2   - Output queue name :
                5. BUSY     - BUSY disposition  : SKIP
                6. FLDMODIF - MSGTRACE position : FIRST
                7. FROMQSN  - From QSN ........ :
                8. LOGLEVEL - Log level ....... : 2
                9. TOQSN    - To QSN .......... :

DSLBA53R_005I : MERVA ID is MHEG, MERVA name is MERVAESA.

DSLBA53R_006I : Queue management routine defined is DSLQMCNV.
                The access features defined are VXBD.

DSLBA53R_102W : The message 8 with QSN 258 is currently in use by
                another user (BUSY).
                As you specified runtime parameter BUSY = SKIP,
                DSLBA53R will skip it (will not delete / move it).

DSLBA53R_107W : The message no. 31 with QSN 281 does not have a
                MSGTRACE field.
```

Figure 38. DSLBA53R (Search for 'Old' Messages) Sample Printout (Part 1 of 2)

```
DSLBA53R_010I : Detailed statistical data

                KEY1 and KEY2 are printed in the second line with log level >= 3.
                The full MSGTRACE entry is printed separately with log level 4.

                     Too                               MSGTRACE         DELE
                Err  Old  Number  QSN         BUSY  YYYY/MM/DD  Days   rc
                ---  ---  ------  ----------  ----  ----------  ----   --
                               1  0000000251        1999/06/29     2
                     DEL       2  0000000252        1999/06/17    14   ok
                     DEL       3  0000000253        1999/06/17    14   ok
                     DEL       4  0000000254        1999/06/17    14   ok
                     DEL       5  0000000255        1999/06/17    14   ok
                     DEL       6  0000000256        1999/06/17    14   ok
                     DEL       7  0000000257        1999/06/17    14   ok
                 >   YES       8  0000000258  BUSY  1999/06/17    14
                     DEL       9  0000000259        1999/06/17    14   ok
                     DEL      10  0000000260        1999/06/17    14   ok
                              11  0000000261        1999/06/30     1
                              12  0000000262        1999/06/30     1
                              13  0000000263        1999/06/29     2
                     DEL      14  0000000264        1999/06/17    14   ok
                     DEL      15  0000000265        1999/06/17    14   ok
                              16  0000000266        1999/06/29     2
                     DEL      17  0000000267        1999/06/17    14   ok
                     DEL      18  0000000268        1999/06/17    14   ok
                     DEL      19  0000000269        1999/06/17    14   ok
                     DEL      20  0000000270        1999/06/17    14   ok
                     DEL      21  0000000271        1999/06/17    14   ok
                     DEL      22  0000000272        1999/06/17    14   ok
                     DEL      23  0000000273        1999/06/17    14   ok
                     DEL      24  0000000274        1999/06/17    14   ok
                     DEL      25  0000000275        1999/06/17    14   ok
                              26  0000000276        1999/06/30     1
                              27  0000000277        1999/06/30     1
                              28  0000000278        1999/06/29     2
                     DEL      29  0000000279        1999/06/17    14   ok
                     DEL      30  0000000280        1999/06/17    14   ok
                 >            31  0000000281
                              32  0000000282        1999/07/01     0

DSLBA53R_012I : Overview statistical data

                No. of messages within From/To QSN      : 32
                No. of messages with MSGTRACE entry     : 31
                - thereof younger than BACKDAYS         :  9
                - thereof as old or older than BACKDAYS : 22
                  -- thereof successfully deleted       : 21
                  -- thereof not deleted because BUSY   :  1
                  -- thereof delete from QDS failed     :  0
                No. of messages without MSGTRACE entry  :  1

DSLBA53R_008I : DSLBA53R ended with return code 4 - Warning.

DSLBA53R_013I : DSLBA53R ended at 1. Jul. 1999 09:42:15
```

Figure 38. DSLBA53R (Search for ‘Old’ Messages) Sample Printout (Part 2 of 2)

Listing Fields

The ‘Detailed statistical data’ of the listing contains the following information:

Err
: A ‘>’ indicates an error with the message, for example:
- The message has no MSGTRACE entry
- The message could not be deleted or moved, because it is currently in use by another user (‘BUSY’)
- The delete or move failed

Too Old
: DEL, MVD, or YES to show that the message is as old or older than BACKDAYS days:

DEL	The message was successfully deleted from the input queue
MVD	The message was successfully moved from the input queue to the output queue
YES	The message is as old or older than BACKDAYS days in function LIST or message could not be deleted or moved

Number	Running number
QSN	Input queue QSN
BUSY	Shows that the message is currently in use by another user (‘BUSY’)
MSGTRACE YYYY/MM/DD	The date of the first or the last MSGTRACE entry
Days	Number of days between the MSGTRACE entry and today
DELE rc	Return code of API function DELE - ‘ok’ indicates that the message was successfully deleted from the input queue
PUTB rc	Return code of API function PUTB - ‘ok’ indicates that the message was successfully moved from the input queue to the output queue
Output QSN	Output queue QSN after a successful MOVE
Key 1	Key 1 of the message. Is printed in the second line with log level ≥ 3
Key 2	Key 2 of the message. Is printed in the second line with log level ≥ 3
MSGTRACE	Full first or last MSGTRACE entry of the message. Is printed with a separate message with log level ≥ 4.

Appendix D. Field-Level Access Fields

This appendix lists the various fields in the MERVA internal structures that can be accessed using the FLDG and FLDP DSLAPI services. Refer to “Field-Level Access for Exit Routines” on page 33 for a description of the field-level access services, and to the definitions of the FLDG 105 and FLDP services 107.

Note that the FLDG and FLDP services are Product-Sensitive Programming Interfaces.

Fields in the following structures can be accessed:

DSLAPI	MERVA ESA API customization fields
DSLCOM	MERVA ESA service communication area
DSLPRM	MERVA ESA customization parameter module
TUCB	Terminal user control block
DSLMFS PL	MFS parameter list
DSLMFS PS	MFS permanent storage
DSLMFS TS	MFS temporary storage
DSLMFS FLDREF	MFS field reference
DSLNIC	Nucleus intertask communication parameter list
DSLUSR	User file record
DSLCWA	MERVA fields in the CICS CWA
DSLFNT	MERVA ESA function table entry.

You can use the following Assembler code to generate a listing of the Assembler definitions of these structures. Mappings in high-level languages are not supplied (except for the MFS parameter list, structure MFSL, in copybook DSLMFSPL).

```
DSLCOM DSECT=YES            Service communication area
DSLPARM TYPE=MAP            DSLPARM TYPE=MAP
DSLMFS MF=TUCB,TYPE=DSECT   TUCB
DSLMFS MF=L                 MFS parameter list
DSLMFS MF=PS                MFS permanent storage
DSLMFS MF=TS                MFS temporary storage
DSLMFS MF=FLDREF            MFS field reference
DSLNIC MF=L                 Inter-task communication parm-list
DSLUSR MF=U,DSECT=YES       User file record
DSLCWA ,                    MERVA fields in the CICS CWA
DSLFNT TYPE=MAP             Function table entry
END
```

Refer to *MERVA for ESA Macro Reference* for more information on these Assembler macros.

The following information is provided for each field:

Field The name you use to access the field

Type The type of data returned to you when you retrieve the field, or that you must provide when changing the fields value:

ADDRESS	4-byte, fullword, binary storage address.
BINARY	4-byte, fullword, binary value.
BIT	The character ‘0’ or ‘1’. The field name may define a 1-, or multiple-bit pattern. A ‘1’ indicates all bits are ones, otherwise the value is ‘0’.
BYTE	A byte in bit representation, that is, a string of 8 ‘0’ or ‘1’ characters.
CHARS	Character string.
PACKED	Packed-decimal value.

Size The length in bytes of a character string or packed value.

Write A ‘Y’ occurs in this column if the field is writeable, in other words, if you can change the field using the FLDP function.

Description
A short description of the field.

Table 19. MERVA ESA API Customization Fields, DSLAPI

Field	Type	Size	Write	Description
APICQMG	BYTE		Y	Queue management requests
APICQBIN	BIT		Y	Binary key, do not modify
APICQDIR	BIT		Y	Direct queue management (DB2 MVS only)
APICQLAZ	BIT		Y	Defer write requests (‘lazy’)
APICQWRB	BIT		Y	Set *write back* indicator
APICQMIT	BIT		Y	Commit direct DB2 queue management requests
APICMSG	BYTE		Y	Message mapping
APICMCLR	BIT		Y	Clear message buffer before mapping
APICMCHK	BIT		Y	Check message

Table 20. MERVA ESA Service Communication Area, DSLCOM

Field	Type	Size	Write	Description
COMPRMA	ADDRESS			Address of MERVA parameter table
COMSRVPA	ADDRESS			Address of DSLSRVP
COMTSVA	ADDRESS			Address of DSLTOFSV
COMFDTA	ADDRESS			Address of field definition table
COMOMSGA	ADDRESS			Address of DSLOMSG (message module)
COMMSGTA	ADDRESS			Address of operator message table
COMMFSA	ADDRESS			Address of DSLMMFS
COMMTTA	ADDRESS			Address of message type table
COMFNTA	ADDRESS			Address of function table
COMFLVPA	ADDRESS			Address of DSLFLVP
COMFLTTA	ADDRESS			Address of file table
COMMFSMA	ADDRESS			Address of MFS error-message buffer
COMXSPS	ADDRESS			HLL exit manager scratchpad storage
COMTRAPA	ADDRESS			Address of DSLTRAP

Table 20. MERVA ESA Service Communication Area, DSLCOM (continued)

Field	Type	Size	Write	Description
COMTRATA	ADDRESS			Address of trace table
COMTRAST	BYTE			Trace status
COMTRAFL	BIT			GETMAIN for trace table failed
COMTRACT	BIT			DSLTRAP is active
COMTRAJP	BIT			DSLTRAP journal put is pending
COMTRASF	BYTE		Y	Special trace switches
COMTRAMF	BIT		Y	Trace MERVA ESA MFS
COMTRATF	BIT		Y	Trace MERVA ESA TOF supervisor
The following fields form the DSLTRAP parameter list:				
COMTRAI1	BINARY		Y	First part trace ID
COMTRAI2	BINARY		Y	Second part trace ID
COMTRASE	BINARY		Y	Session ID
COMTRARE	BINARY		Y	Register 14
COMTRADA	CHARS	24	Y	Data part
COMTRARC	BINARY			DSLTRAP return code
COMTRAWK	ADDRESS			Work area address of DSLTRAP
The following fields are used by programs not linked to DSLNUC:				
COMNICPL	ADDRESS			Address of DSLNIC parameter list
COMTUCBA	ADDRESS			Address of TUCB
COMMTBA	ADDRESS			Address of MFS load table
COMERRA	ADDRESS			Address of DSLEERR
COMEPTA	ADDRESS			Address of EUD program table
The following fields are used by programs linked to DSLNUC or loaded to DSLNUC:				
COMJRNPA	ADDRESS			Address of DSLJRNP
COMNCSA	ADDRESS			Address of DSLNCS
COMNMOPA	ADDRESS			Address of DSLNMOP
COMQMGTA	ADDRESS			Address of DSLQMGT
COMTIMPA	ADDRESS			Address of DSLTIMP
COMTIME	ADDRESS			MERVA startup time
COMSTAT0	BYTE			MERVA ready status byte
COMSTAT1	BYTE			MERVA status byte
COMSTSHU	BIT			MERVA shutdown is set
COMSTCAN	BIT			MERVA cancel is set
COMSPNR	BYTE			MVS getmain subpool number
The following fields are used or reserved for network programs:				

COMDSNL	ADDRESS			Used by S.W.I.F.T Link for Address of table
COMRECON	ADDRESS			Used by reconciliation
COMDTNL	ADDRESS			Used by Telex Link
COMDWNN	ADDRESS		Y	Used by national network programs
COMUSER1	ADDRESS		Y	Reserved for user
COMUSER2	ADDRESS		Y	Reserved for user
COMUSER3	ADDRESS		Y	Reserved for user
COMUSER4	ADDRESS		Y	Reserved for user
The following field is used only under IMS:				
COMPCBLA	ADDRESS			Address of PCB-address-list
The following fields are used only under CICS:				
COMCWAA	ADDRESS			Address of DSLCWA
COMEISTG	ADDRESS			Address of exec interface storage
COMEIB	ADDRESS			Address of exec interface block
COMCOM	ADDRESS			Reserved

Table 21. MERVA ESA Customization Parameter Module, DSLPRM

Field	**Type**	**Size**	**Write**	**Description**
Module names:				
NPFDT	CHARS	8		Field definition table name
NPFNT	CHARS	8		Function table name
NPMTT	CHARS	8		Message type table name
NPMSG	CHARS	8		Operator message table name
NPTFD	CHARS	8		IMS: Terminal feature table name
Buffer sizes:				
NPEXSPA	BINARY			Size of exit SPA
NPEXTS	BINARY			Size of exit temporary storage
NPMFSPS	BINARY			Size of MFS permanent storage
NPMFSTS	BINARY			Size of MFS temporary storage
NPMFSRK	BINARY			Size of MFS retype-buffer
NPTOFSZ	BINARY			Size of TOF
NPNICBUF	BINARY			Buffer length for DSLNIC
NPNICPL	BINARY			Parm list length for DSLNIC
Other parameters:				

NPCWAOFF	BINARY			CICS: Bytes left free in CWA
NPCVTABO	BINARY			MERVA offset in the CVT extension
NPNQE	BINARY			Number of QDS queue elements
NPUSER	BINARY	2		Max. number of active users
NPMCBNUM	BINARY	2		Number of MCBs
NPDM	BINARY	2		Number of unsolicited messages
NPSVC	BINARY	2		MVS: SVC instruction
NPMERVID	CHARS	4		MERVA ID for operator messages
NPCID	CHARS	3		CICS: MERVA ID for CICS signon
NPBITS	BYTE			MERVA parameter bits -1-
NPEXSEC	BIT			External security bit
NPEXUSR	BIT			Extended origin ID check in USR function
NPEXQUE	BIT			Queue test commands allowed
NPQDS2	BIT			Secondary queue data set used
NPQCONT	BIT			Continue after failure in one QDS
NPQUETRA	BIT			Queue trace of DSLQMGT
NPQUETRL	BIT			Queue trace is ‘large’
NPNOPINT	BIT			No operator intervention with DSL090A
NPTRACE	BYTE		Y	MERVA trace status
NPTREXT	BIT		Y	External trace on
NPTRINT	BIT		Y	Internal trace on
NPRTRACE	BYTE		Y	Routing trace status
NPRTRALL	BIT		Y	Routing trace all on
NPRTRWNG	BIT		Y	Routing trace warning on
NPRTRSEV	BIT		Y	Routing trace severe error on
NPRTROFF	BIT		Y	Routing trace off
NPLANCDS	CHARS	1		Country specific decimal separator
NPLANCTS	CHARS	1		Country specific thousands separator
NPJRNBUF	BINARY			Journal buffer length
NPFLT	CHARS	8		File table name
NPFLVTS	BINARY			DSLFLVP TS size
NPLAN	CHARS	1		Language for sign-on panel
NPBITS2	BYTE			MERVA parameter bits -2-
NPCINTER	BIT			CICS: When no NIC/INTRA, use INTER
NPUCTRAN	BIT			Uppercase translation for display
NPTSAUX	BIT			CICS auxiliary TS used
NPUSFPW	BIT			Passwords not displayed in user file
NPEXAFO	BIT			Automatic force during sign-on
NPMVSSS	BIT			MERVA uses subsystem control block
NPRECON	BIT			Reconciliation active
NPPGCALL	BIT			End-user-driver can be called by pgm
NPBITS3	BYTE			MERVA parameter bits -3-

NPLSTOP	BIT			Stop if routing table loads fail
NPUMRYES	BIT			Unique message reference is used
NPUMRIMM	BIT			UMR is got immediately with MT command
NPXSTOP	BIT			Stop if (DSLN) user exit loads fail
NPEXDSP	BIT			DSLNUSR allows API display and list request
NPEXJRN	BIT			Journal display command is allowed
NPJRNSEG	BIT			Journal records are segmented
NPRECONT	BIT			Continue after Reconciliation error
NPBITS4	BYTE			MERVA parameter bits -4-
NPEXUMSK	BIT			User file data masking
NPCURFIL	BIT			File access for currency codes
NPCURTAB	BIT			File and table access for currency codes
NPDBCSON	BIT			DBCS support
NPEXUID	BIT			MERVA user ID sign-on bypassed
NPJRNLYY	BIT			Four-digit year in journal
NPSDRC	BIT			RC of DSLSDI, DSLSDO, and DSLSDY like reason code
NPNOCHK	BIT			No password check with EXSEC=YES
NPAPSMSG	BINARY			DSLAPI message buffer size
NPMSGLIM	BINARY			IMS: Message limit for transaction
NPPGBUF	BINARY			EUD-to-PGM interface data buffer size
NPOPID	CHARS	3		Master operator ID, “***”, or “///”
NPNAME	CHARS	8		Name for UMR and others
NPAPIUID	CHARS	8		User ID for DSLAPI cmd requests
NPLMSTAT	BYTE			Large message status
NPLMYES	BIT			Large messages used
NPLMSTOP	BIT			Stop MERVA if DSLQLRG fails
NPLMSIZE	BINARY			Minimum size of a large message
NPMAXBUF	BINARY			Maximum dynamic buffer length
NPTOFINC	BINARY			TOF increase value
NPTIER	BINARY			Monthly usage level (message volume)
NPEUDTRN	CHARS	8		DSLEUD transaction code
NPMRVUSR	CHARS	8		User ID when user file is empty
NPRACSVC	BINARY	2		MVS: RACF® SVC instruction no.
NPSONNUM	BINARY			Number of sign-on trials before user ID will be revoked
NPIRQNO	BINARY	2		Number of request queue elements
NPITCM1	BINARY			Intertask communication method CICS transactions
NPITCM2	BINARY			Intertask communication method non-CICS transactions
NPITCM3	BYTE			Intertask communication method modifier
NPITCM3P	BIT			Intraregion communication uses single ECB

NPITCCQN	CHARS	5		CICS TS server queue name
NPITCAC	CHARS	25		APPC requestor parameters
NPITCACS	CHARS	8		SYMDEST used by requestor
NPITCACL	CHARS	17		Partner LU Name for requestor
NPITCAS	CHARS	113		APPC server parameters
NPITCASD	CHARS	8		SYMDEST used by server
NPITCALU	CHARS	8		NOSCHED local LU name
NPITCATP	CHARS	64		MERVA TP Name
NPITCAPL	CHARS	17		Partner LU name
NPITCAUI	CHARS	8		User ID
NPITCAPR	CHARS	8		Profile
NPWSAS	CHARS	90		Workstation APPC/MVS server parameters
NPWSASD	CHARS	8		SYMDEST used by server
NPWSALU	CHARS	8		NOSCHED local LU name
NPWSATPL	BINARY	2		TP name length
NPWSATP	CHARS	64		TP name
NPWSAPR	CHARS	8		Profile
NPBITS5	BYTE			MERVA parameter bits -5-
NPUGYES	BIT			Group users on
NPUGREQ	BIT			Group users is required
NPEXGRP	BIT			Extended group ID check in USR function
NPEXQUM	BIT			Queue test cmds allowed for master user
NPJRNSWS	BYTE			Journal File SWITCH status
NPJRNSWT	BIT			Journal File SWITCH status specified
NPJRNSWM	BIT			Journal File SWITCH mask
NPBITS6	BYTE			MERVA parameter bits -6-
NPSDIRDB	BIT			DSLSDI, DSLSDO, and DSLSDY use direct DB2 queue management
NPQIONM	CHARS	8		Queue management routine name
NPQIOTP	CHARS	4		Queue management access type
NPITCACT	CHARS	64		MERVA TP name used by client
NPITCACM	CHARS	8		MERVA mode name used by client
NPWSSEC	BYTE			Workstation security bits
NPWSSENC	BIT			Encryption
NPWSSAUT	BIT			Authentication
NPWSSPWD	BIT			Password scrambling
NPWSSOFF	BIT			None
NPWSTP#	BINARY	2		Workstation TCP/IP server port number
NPMQMNM	CHARS	48		MQ manager name
NPITCRTQ	CHARS	96		ITC MQ reply-to queue names
NPITCRTN	CHARS	48		Normal or model queue name
NPITCRTD	CHARS	48		Dynamic queue name

NPITCRCV	CHARS	96		ITC MQ receive queue names
NPITCRVN	CHARS	48		Normal or model queue name
NPITCRVD	CHARS	48		Reserved
NPITCSND	CHARS	96		ITC MQ send queue names
NPITCSDN	CHARS	48		Normal or model queue name
NPITCSDD	CHARS	48		Reserved
NPITCWTT	BINARY	4		ITC wait time interval
NPISCRTQ	CHARS	96		ISC MQ reply-to queue names
NPISCRTN	CHARS	48		Normal or model queue name
NPISCRTD	CHARS	48		Reserved
NPISCRCV	CHARS	96		ISC MQ receive queue names
NPISCRVN	CHARS	48		Normal or model queue name
NPISCRVD	CHARS	48		Reserved
NPISCSND	CHARS	96		ISC MQ send queue names
NPISCSDN	CHARS	48		Normal or model queue name
NPISCSDD	CHARS	48		Reserved
NPISCMID	CHARS	33		ISC MQ MERVA IDs
NPISCMQP	CHARS	16		ISC MQ MERVA ID for primary nucleus
NPISCMQL	CHARS	16		ISC MQ MERVA ID for secondary nucleus
NPISCSDS	CHARS	1		ISC queue name silable type
NPISCPRM	CHARS	1		ISC nucleus primary instance
NPISCSTT	CHARS	1		ISC nucleus instance start
NPISCXCF	CHARS	28		ISC XCF names
NPISCXGP	CHARS	8		XCF group name
NPISCXMB	CHARS	16		XCF group member name
NPISCJWT	BINARY	4		Join wait time interval
NPMFCLIC	BINARY	4		Number of client licences
NPDB2PLB	CHARS	8		DB2 plan name for batch nucleus program
NPDB2SS	CHARS	4		DB2 subsystem name for batch nucleus pgm
NPRTNAM	CHARS	60		Bank’s name in printout of batch utilities

Table 22. Terminal User Control Block, DSLMFS MF=TUCB

Field	Type	Size	Write	Description
TUCBLL	BINARY			Length
TUCBRV1	CHARS	4		Reserved for IMS
TUCBTRAN	CHARS	8		Transaction name
TUCBLTN	CHARS	8		Logical terminal name
TUCBRSV1	CHARS	2		transaction/terminal ID

The following fields concern the current MERVA function:

TUCNAME	CHARS	8		Function name

TUCROUTN	CHARS	8		Name of routing table
TUCROUTA	ADDRESS			Address of routing table

The following fields concern the current MERVA queue:

TUCHSN	BINARY			Highest QSN used in this queue
TUCFQE	BINARY			First QKTE in this queue
TUCLQE	BINARY			Last QKTE in this queue
TUCECBA	ADDRESS			ECB address for posting after put
TUCSIZE	BINARY			Size for storing messages in queue
TUCTRESH	BINARY			Threshold number of this queue
TUCKFLD1	CHARS	8		Name of key field 1
TUCKLEN1	BINARY			Length of key 1
TUCKOFF1	BINARY			Offset to start of key 1
TUCKFLD2	CHARS	8		Name of key field 2
TUCKLEN2	BINARY			Length of key 2
TUCKOFF2	BINARY			Offset to start of key 2
TUCNXTNM	CHARS	8		Name of next function
TUCHCONM	CHARS	8		Name of hard-copy function
TUCTRAN	CHARS	8		Transaction name
TUCLTE1	CHARS	8		Logical terminal name
TUCLTE2	CHARS	8		Logical terminal name
TUCNSO	BINARY			Number of signed on end users
TUCMSGST	BYTE			Message-processing status
TUCDENT	BIT			New message generation allowed
TUCPROT	BIT			Screen protected
TUCKMSG	BIT			Keep message after print
TUCPRON	BIT			Noprompt only display allowed
TUCNRKY	BIT			Retype verification allowed
TUCPROM	BIT			Noprompt processing allowed
TUCCHCK	BIT			Message check specified
TUCFRAME	BIT			Frame message ID specified
TUCQUEST	BYTE			Queue status 1
TUCIGNAC	BIT			IMS only: ignore active status
TUCHLDC	BIT			Function in ‘hold’ currently
TUCHLDI	BIT			Function in ‘hold’ initial
TUCACTIV	BIT			Function active
TUCNOTY	BIT			Notify = yes is specified
TUCDSLQ	BIT			Function with queue
TUCDUMQ	BIT			Function with dummy queue
TUCSTART	BIT			Start command executed
TUCRSI	BIT			Restart info in this queue

TUCFRAMT	CHARS	8		Name of top-frame message ID
TUCFRAMB	CHARS	8		Name of bottom-frame message ID
TUCUAPL	ADDRESS			User application action
TUCFMID	CHARS	1		Printer format ID
TUCCOMPF	CHARS	1		Compression format
TUCCOMND	BYTE			Valid commands
TUCCAUT	BIT			Authenticate command
TUCCDEL	BIT			Delete command
TUCCOK	BIT			OK command
TUCCROU	BIT			Route command
TUCXPND	BYTE			Field expansion requirements for function
TUCXPCO	BIT			Common names expansion
TUCXPPR	BIT			Private names expansion
TUCXPNC	BIT			Noprompt message conditional expansion
TUCXPPC	BIT			Prompt message conditional expansion
TUCXPCL	BIT			Clear is requested
TUCXPNO	BIT			No field expansion
TUCMLIM	BINARY			Message limit within IMS scheduled cycle
TUCPFGN	BINARY			PF-key-groupnumber
TUCMSGS2	BYTE			Message-processing status 2
TUCMS2M2	BIT			SWIFT II display mode
TUCPFKY	CHARS	8		User PF-key table name
TUCFPGM	CHARS	8		Name of DSLEUD function program
TUCCOPYQ	CHARS	8		Name of copy queue
TUCRELA1	ADDRESS			Address of first related function
TUCRELA2	ADDRESS			Address of second related function
TUCMSGID	CHARS	8		Message ID for mapping and formatting
TUCNQE	BINARY			Number of QEs in this queue
TUCQUST2	BYTE			Second queue status
TUCQ2DQF	BIT			No display with DQ filled
TUCQ2TOF	BIT			No TOF format in this queue
TUCQ2LST	BIT			Start message selection with queue list
TUCQ2STS	BIT			Store = small specified
TUCQ2STL	BIT			Store = large specified
TUCLIID	CHARS	1		ID of list MCB
TUCLILEN	BINARY			Maximum length of list data
TUCCKIND	BYTE			Checkpoint/message limit indicator
TUCMLSET	BIT			Message limit set
TUCCKSET	BIT			Checkpoint set
TUCINTQ	CHARS	8		Intermediate queue EDI-SWI conversion

The following fields address various data blocks:

TUCBPCBA	ADDRESS			Address of I/O PCB (IMS)
TUCBPCBL	ADDRESS			Address of PCB list (IMS)
TUCBCMLA	ADDRESS			Address of command table list
TUCBCMPA	ADDRESS			Address of command parser buffer
TUCBUSRA	ADDRESS			Address of user file record
TUCBLDSA	ADDRESS			Address of LDS buffer
TUCBRKYA	ADDRESS			Address of retype buffer
TUCBTOFA	ADDRESS			Address of TOF buffer
TUCBMFSP	ADDRESS			Address of MFS permanent storage
TUCBMFST	ADDRESS			Address of MFS temporary storage
TUCBNCPL	ADDRESS			Address of DSLNIC-parmlist

The following fields concern the end-user session:

TUCBTRML	BINARY			Size of terminal buffer
TUCBROWN	BINARY			Rows per page/screen
TUCBCOLN	BINARY			Columns per page/screen
TUCBDEVC	BYTE			Device capabilities
TUCBHIL	BIT			Extended highlight indication
TUCBCOLR	BIT			Extended color indication
TUCBALTC	BIT			Alternate screen size to be used
TUCBUCTR	BIT			Uppercase translation required
TUCBILUC	BIT			Input lines underscored
TUCBDVTR	BIT			Bracket translation
TUCBDEV	BYTE			Device type
TUCBSCR	BIT			Screen
TUCBHCP	BIT			Terminal printer
TUCBSCS	BIT			SCS printer
TUCBSYS	BIT			System printer
TUCBACTP	BINARY			Actual page
TUCBACTL	BINARY			Actual line
TUCBACTN	BINARY			Actual nesting ID
TUCBACTG	BINARY			Actual field group
TUCBACTO	BINARY			Actual occurrence
TUCBACTF	CHARS	8		Actual field name
TUCBACTD	BINARY			Actual data area
TUCBACTI	CHARS	1		Actual option indicator
TUCBACTS	CHARS	1		Actual scroll mode P or S
TUCBACTA	CHARS	1		Actual occurrence action indicator A or ' '
TUCBCOMP	CHARS	1		Actual compression mode
TUCBWNDW	CHARS	3		Actual window, values: MSG BOT TOP
TUCBCRS	BYTE		Y	

TUCBCRSI	BIT		Y	Cursor selection indicator
TUCBPFKI	BIT		Y	PF-key has been pressed
TUCBENTR	BIT		Y	Enter key was pressed
TUCBNOND	BIT		Y	Suppress automatic end
TUCBCRSR	BINARY		Y	Row of cursor receive/send
TUCBCRSC	BINARY		Y	Column of cursor receive/send
TUCBCRST	CHARS	15	Y	TOF reference of cursor receive/send
TUCBPCTR	BYTE		Y	Screen paging control
TUCBCHEC	BIT		Y	Simulate checking error on page
TUCBPCRS	BIT		Y	Put cursor on TOF field
TUCBXCRS	BIT		Y	Put cursor on row/column specification
TUCBPIEP	BIT		Y	Beep indicator
TUCBSUPS	BIT		Y	Suppress scrolling
TUCBDYPR	BIT		Y	Dynamic protection of windows
TUCBPNPL	BIT		Y	Put cursor on prefix area
TUCBPNPI	BIT		Y	Put cursor on noprompt line
TUCBMFRE	BINARY			MFS reason code (I/O-buffer --> TOF)
TUCBMIDT	CHARS	8		Top window message ID
TUCBMIDM	CHARS	8		Message window message ID
TUCBMIDB	CHARS	8		Bottom window message ID
TUCBUSER	CHARS	8		User identification
TUCBORIG	CHARS	34		User origin
TUCBLID	CHARS	1		User language
TUCBNID	CHARS	1		User line format
TUCBMSTA	CHARS	3		Message status, values: msg.fun
TUCBMSGC	CHARS	5		Message generation count
TUCBPFKN	CHARS	8	Y	User PF-key table name
TUCBPFGR	BINARY		Y	PF-keyset-group number
TUCBCNTR	BYTE		Y	Control flags for end-user process
TUCBCMDL	BIT		Y	Set cursor on command-line
TUCBEMSG	BIT		Y	Erase DSLMSG data areas
TUCBSPMG	BIT		Y	Suppress generating check messages
TUCBPFDY	BIT		Y	PF-key set dynamically
TUCBMTCM	BIT		Y	Message generation command active
TUCBSON	BIT		Y	Sign-on in process
TUCBLSTP	BIT		Y	Last page is reached
TUCBCMDH	BIT		Y	Intensified display for command line
TUCBLINE	BINARY			Top line in message
TUCBESPA	ADDRESS			Address SPA area of DSLE...-exits
TUCBFUN	CHARS	8		Actual (new) function name
TUCBCNTS	BYTE		Y	Control flags 2 for end-user process
TUCBEOMC	BIT		Y	EOM command active

TUCBDAEN	BIT		Y	Data checking modus for page
TUCBQUCC	BIT		Y	Queue test command in process
TUCBACCN	BINARY			Current nesting ID
TUCBACCO	BINARY			Current occurrence
TUCBACCD	BINARY			Current data area
TUCBACSL	BINARY			Slot length
TUCBACSO	BINARY			Slot offset
TUCBACHL	CHARS	8		Default panel for help
TUCBPFGS	BINARY		Y	PF group of message window
TUCBCMPS	CHARS	1	Y	Save compression
TUCBMSTS	CHARS	3	Y	Save status msg/fun
TUCBMNFL	BYTE		Y	MERVA Link flag
TUCBRMNA	BIT		Y	Resume Merva Link application
TUCBACTM	CHARS	8		Actual message ID
TUCBACTR	BINARY			Actual representative sequence number
TUCBNPRL	BINARY			Noprompt line number for CRS
TUCBCRSO	BINARY		Y	Offset cursor in screen slot
TUCBWS1A	ADDRESS			Address free space in storage pool 1
TUCBWS1L	BINARY			Length free space pool 1
TUCBWS2A	ADDRESS			Address free space in storage pool 2
TUCBWS2L	BINARY			Length free space pool 2
TUCBWS3A	ADDRESS			Address free space in storage pool 3
TUCBWS3L	BINARY			Length free space pool 3
TUCBOCPS	CHARS	4	Y	Cursor position previous panel (row/column)
TUCBPSNO	CHARS	4	Y	Sign-on sequence no (feedback DSLNUSR)
TUCBHDCT	BYTE			Message header display format
TUCBHDS2	BIT		Y	Header type 2 display
TUCBOTTO	CHARS	1	Y	EUD : TOF trace
TUCBOTMF	CHARS	1	Y	EUD : MFS trace
TUCBOTPG	CHARS	1	Y	EUD : PGM trace
TUCBOTRN	CHARS	12	Y	Online trace setting reserved

Table 23. MFS Parameter List, DSLMFS MF=L

Field	Type	Size	Write	Description
MFSLTYP	BINARY		Y	Type code
MFSLMED	BINARY		Y	Medium-code
MFSLOPT1	BYTE		Y	
MFSLO1NM	BIT		Y	OPT=NOMSG
MFSLO1FN	BIT		Y	OPT=FUNC
MFSLO1CH	BIT		Y	OPT=CHECK
MFSLO1RT	BIT		Y	OPT=RETRY
MFSLO1CN	BIT		Y	OPT=CONT

Table 23. MFS Parameter List, DSLMFS MF=L (continued)

Field	Type	Size	Write	Description
MFSLO1PT	BIT		Y	OPT=PROT
MFSLO1BT	BIT		Y	OPT=BATCH
MFSLO1EM	BIT		Y	OPT=ERRMSG
MFSLOPT2	BYTE		Y	
MFSLO2NI	BIT		Y	OPT=NXTNI
MFSLO2DE	BIT		Y	OPT=DEEDIT
MFSLO2CL	BIT		Y	OPT=CLRPERM
MFSLO2DY	BIT		Y	OPT=DYNBUF
MFSLO2DL	BIT		Y	OPT=DELETE
MFSLO2WR	BIT		Y	OPT=WRITE
MFSLO2RD	BIT		Y	OPT=READ
MFSLRET	BYTE		Y	Return code
MFSLREAS	BINARY		Y	Reason code function dependent
MFSLMODN	BINARY		Y	Module message number
MFSLMILF	CHARS	1	Y	Line format for network
MFSLWORK	BYTE		Y	Work indicator
MFSLCECI	BIT		Y	MFS exit invoked under CICS
MFSLXMGR	BIT		Y	MFS exit invoked as an HLL exit
MFSLCOMA	ADDRESS		Y	Address of MERVA ESA communication area
MFSLPERM	ADDRESS		Y	Address of permanent storage
MFSLTEMP	ADDRESS		Y	Address of temporary storage
MFSLENVA	ADDRESS		Y	Address of environment string
MFSLMSG	ADDRESS		Y	Address of message ID/load module name
MFSLTOF	ADDRESS		Y	Address of DSLTOF
MFSLFLD	ADDRESS		Y	Address of field reference
MFSLIBUF	ADDRESS		Y	Address of input buffer
MFSLOBUF	ADDRESS		Y	Address of output buffer

Table 24. MFS Permanent Storage, DSLMFS MF=PS

Field	Type	Size	Write	Description
MFSPP	BINARY			Storage length
MFSPENV	CHARS	8	Y	External environment string
MFSPIENV	CHARS	8	Y	Internal environment string
MFSPIND	BYTE			Permanent indicator
MFSPDYNP	BIT			Dynamic permanent storage
MFSPINRV	BIT			Reserved
MFSPPEMS	BIT		Y	Pass error message to caller
MFSPTOFU	BIT			TOF full during MFS cycle
MFSPUSED	BIT			Save area in use (recursive test)
MFSPUEMS	BIT			Process error message (recursive)

Table 24. MFS Permanent Storage, DSLMFS MF=PS (continued)

Field	Type	Size	Write	Description
MFSPINXM	BIT			Exit manager init by DSLMMFS
MFSPBAT	BIT			DC or batch process (BATCH=YES)
MFSPINDX	BYTE			MFS disable indicator
MFSPRCTR	BINARY			MFS recursion counter
MFSPRASF	BYTE			Save trace status from COMTRASF
MFSPTCTR	BINARY			Number of dynamic GETMAINs
MFSPTDAM	ADDRESS			Maximum dynamic area size
MFSPTDAA	ADDRESS			Current dynamic area size
MFSPTS	ADDRESS			Address of temporary storage pool
MFSPTSC	ADDRESS			Free space pointer
MFSPTSE	ADDRESS			End address of pool
MFSPTSA	ADDRESS			Address of actual temporary storage
MFSPMTBA	ADDRESS			Address of internal load table
MFSPUCOM	ADDRESS		Y	User exit communication field
MFSPCCOM	CHARS	16	Y	Check exit communication field
MFSPSCOM	CHARS	16	Y	Separation exit communication field
MFSPMODN	CHARS	8		Module/MCB/PFK table name
MFSPMODY	CHARS	8		Module entry name
MFSPMODA	ADDRESS			Load module address
MFSPMODE	ADDRESS			Module entry address
MFSPMCBM	ADDRESS			MCB message descriptor address
MFSPMCBD	ADDRESS			MCB device descriptor address
MFSPMIDN	CHARS	8		Message identification
MFSPMTTE	ADDRESS			Address of current MTT entry
MFSPPFKN	CHARS	8		PF-key table name
MFSPPFKA	ADDRESS			PF-key table address
MFSPMPTE	ADDRESS			Address of current MPT entry
MFSPEMSB	BINARY			MFS error message (highest severity)
MFSPEMSA	BINARY			Actual message length
MFSPEMSR	BINARY			Error message reason
MFSPEMSG	CHARS	80		Message
MOMSRETC	BYTE			Return code
MOMSRSNC	BYTE			Reason code
MOMSTAB	ADDRESS			Address of message table
MOMSLAN	CHARS	1		Language ID
MOMSMID	CHARS	7		Message ID
MOMSWORK	CHARS	8		Work area

The following fields are the MFS Error message substitution variables:

MFSPOMRC	CHARS	2	Y	@0 MFS return code
MFSPOMRS	BINARY		Y	@1 MFS reason code
MFSPOMID	CHARS	8	Y	@2 Message ID
MFSPOMOD	CHARS	8	Y	@3 MCB/module name
MFSPOMFL	CHARS	8	Y	@4 Name of TOF field
MFSPOMLV	BINARY		Y	@5 Nesting identifier/module number
MFSPOMSQ	BINARY		Y	@6 Field group index
MFSPOMOC	BINARY		Y	@7 Repeatable sequence index
MFSPOMDA	BINARY		Y	@8 Field data area
MFSPOML1	BINARY		Y	@9 Page/line number
MFSPOML2	BINARY		Y	@10 Line number 2
MFSPOMTS	BINARY		Y	@11 Sub-function reason code
MFSPOMTC	BINARY		Y	@12 Sub-function return code
MFSPOMCN	CHARS	8	Y	@13 Screen command name
MFSPOMPN	BINARY		Y	@14 Parameter number
MFSPOMST	CHARS	24	Y	@15 String/key
MFSPOMPP	CHARS	3	Y	@16 Prefix command
MFSPOMFI	CHARS	1	Y	@17 Language id/form
MFSPTBL	BINARY			Buffer length for message initialization
MFSPTD	CHARS	20		Permanent area for message initialization
MFSPPSND	CHARS	8		Send status DSLMPSXX
MFSPPBL	BINARY			Buffer length for window control
MFSPPDL	BINARY			Data length in buffer +4

Table 25. MFS Temporary Storage, DSLMFS MF=TS

Field	Type	Size	Write	Description
MFSTUSLL	BINARY			Storage length
MFSTUS00	BINARY		Y	Reserved ('DY' = dynamic indicator)
MFSTUSNX	ADDRESS			Address of next level temporary storage
MFSTUSPR	ADDRESS			Address of prev level temporary storage
MFSTIENV	CHARS	8	Y	Internal environment information
MFSTPLSV	ADDRESS		Y	Parmlist save word
MFSTPRET	ADDRESS		Y	Return save word
MFSAVBAK	BINARY		Y	Backward chain pointer
MFSAVFOR	BINARY		Y	Forward chain pointer
MFSAVRET	BINARY		Y	Return location
MFSAVENT	BINARY		Y	Entry address
MFSAVR0	BINARY		Y	r0-r12

Table 26. MFS Field Reference, DSLMFS MF=FLDREF

Field	Type	Size	Write	Description
FLDNI	BINARY		Y	Field nesting identifier

Table 26. MFS Field Reference, DSLMFS MF=FLDREF (continued)

Field	Type	Size	Write	Description
FLDFG	BINARY		Y	Field group index
FLDRS	BINARY		Y	Field repeatable seq. index
FLDNAME	CHARS	8	Y	Field name
FLDDA	BINARY		Y	Field data area index
FLDOPT	CHARS	1	Y	Field option indicator
FLDSTAT	BYTE		Y	Field status
FLDSTEX	BIT		Y	Sub-field extension exists
FLDSTEM	BIT		Y	Data area empty
FLDINIT	BIT		Y	TOF request type was init
FLDCHEK	BIT		Y	TOF checking required
FLDSTX2	BIT		Y	2nd extension exists

The following sub-field fields exist if FLDSTEX is 1:

FLDNAME0	CHARS	8	Y	Field master name
FLDOFF	BINARY		Y	Offset of subfield
FLDLEN	BINARY		Y	Length of subfield
FLDLENM	BINARY		Y	Maximum length of subfield

The following nested repeatable sequence fields can exist if FLDSTX2 is 1:

FLDRSXNN	BINARY		Y	Number of RS indexes used (following)
FLDRSXO1	BINARY		Y	Occurrence index in first rep seq
FLDRSXO2	BINARY		Y	Occurrence index in nested rep seqs
FLDRSXO3	BINARY		Y	Occurrence index in nested rep seqs
FLDRSXO4	BINARY		Y	Occurrence index in nested rep seqs
FLDRSXO5	BINARY		Y	Occurrence index in nested rep seqs
FLDRSXO6	BINARY		Y	Occurrence index in nested rep seqs
FLDRSXO7	BINARY		Y	Occurrence index in nested rep seqs
FLDRSXO8	BINARY		Y	Occurrence index in nested rep seqs
FLDRSXO9	BINARY		Y	Occurrence index in nested rep seqs

Table 27. Nucleus intertask communication parameter list, DSLNIC MF=L

Field	Type	Size	Write	Description
NICICB	ADDRESS			Address of ICB
NICTIME	PACKED	4		MERVA startup time (0hhmmssF)
NICCOM	ADDRESS			Address of DSLCOM
NICECB	ADDRESS			Communication ECB
NICNAME	ADDRESS			Address of servicing module name (REQ)
NICPL	ADDRESS			Address of PARMLIST
NICBUF	ADDRESS			Address of buffer (REQ,RESP)

Table 27. Nucleus intertask communication parameter list, DSLNIC MF=L (continued)

Field	Type	Size	Write	Description
NICTYPE	BYTE			Type of request
NICOPT	BYTE			Option byte
NICNTS	BIT			Interregion from DSLNTS only
NICINTER	BIT			Interregion from requestor
NICINTRA	BIT			Intraregion from requestor
NICVER3	BIT			Error info in NICPL
NICDYNB	BIT			Dynamic buffer allowed
NICRC	BYTE			Return code
NICERRIP	BINARY			Error info parm list buffer
NICERRIB	BINARY			Error info buffer

Table 28. User File Record, DSLUSR MF=U

Field	Type	Size	Write	Description
USRULBM	BINARY			Maximal buffer length
USRULB	BINARY			Buffer length
USRUKEY	CHARS	8		Key (user ID)
USRUSCPW	CHARS	8		Scrambled password
USRUNAME	CHARS	18		User's name
USRUORID	CHARS	34		Origin ID
USRUDATE	CHARS	8		ISO date of last update
USRUTIME	CHARS	8		Time of last update
USRUUUID	CHARS	8		Update user ID
USRUDATP	CHARS	8		ISO date of last password change
USRUTIMP	CHARS	8		Time of last password change
USRUPFKS	CHARS	8		PF-key setname
USRULID	CHARS	1		Language ID
USRUNLIF	CHARS	1		Noprompt line format
USRUDNW	CHARS	1		Default network (for msg. type)
USRUAUT	CHARS	1		Record authorization (U = not auth.)
USRUFTAB	CHARS	18×8		Allowed functions (max 18)
USRUAMSG	CHARS	24×8		Message types assigned to user
USRUNOCM	CHARS	60		Commands forbidden for user ID
USRUUFLM	CHARS	8		FLM administrator
USRUIDTA	CHARS	17		Internal data area / reserved
USRUDATS	CHARS	8		ISO date of last sign-on
USRUIMRX	BINARY			Traffic Reconciliation user class
USRUGRP	CHARS	8		Group ID
USRUUSON	BINARY			Number of rejected sign-ons
USRUUTYP	CHARS	1		User type (B,K,L,..)
USRUUDTA	CHARS	48		User-data area 1

Table 28. User File Record, DSLUSR MF=U (continued)

Field	Type	Size	Write	Description
USRUUDTB	CHARS	48		User-data area 2

The following fields comprise the user file pending area:

USRUPW1P	CHARS	8		Scrambled password
USRUPW2P	CHARS	8		Scrambled password
USRUNAMP	CHARS	18		User's name
USRUORGP	CHARS	34		Origin ID
USRUUSRP	CHARS	8		Update user ID
USRULIDP	CHARS	1		Language ID
USRUNOPP	CHARS	1		Noprompt line format
USRUNETP	CHARS	1		default network (for msg. type)
USRUFCTP	CHARS	18×8		Allowed functions (max 18)
USRUPFKP	CHARS	8		PF-key-set name
USRUMTPP	CHARS	24×8		Message types assigned to user
USRUUCDP	CHARS	60		Commands forbidden for user ID
USRUFLMP	CHARS	8		FLM administrator
USRUDT1P	CHARS	48		User-data area 1
USRUDT2P	CHARS	48		User-data area 2
USRUAUT2	CHARS	1		Record authorization 2 (U = not auth.)
USRUUTPP	CHARS	1		User type (B,K,L,..)
USRUUSNP	BINARY			Number of rejected sign-ons
USRUGRPP	CHARS	8		Group ID
USRUIMRP	BINARY			Traffic Reconciliation user class
USRUFILL	CHARS	29		For expansion
USRUWKEY	CHARS	8		Alternate scrambling key

Table 29. CICS Common Work Area (CWA), DSLCWA

Field	Type	Size	Write	Description
CWATIME	PACKED	4		MERVA startup time (0hhmmssF)

MERVA ESA Function Table

A field of the function table is accessed by specifying the qualified name '**xxxxxxxx.yyyyyyyy**' as the field name in the FLDG call. Alternatively, it can be specified as '**xxxxxxxx.nnnnnnnn**'.

xxxxxxxx	is the field name as shown in the table below
yyyyyyyy	stands for the actual function name
nnnnnnnn	is a decimal number indicating the position of the requested function table entry in the function table.

The field name, the function name, and the number can be specified in their actual length with a maximum of 8 bytes. Padding with blanks or leading zeroe is not necessary.

An FLDG call for **FNTNAME.nnnnnnnn** returns the name of a function in an 8-byte character field.

An FLDG call for **FNTINDEX** returns the number of functions defined in the function table in form of a binary value.

An FLDG call for **FNTINDEX.yyyyyyyy** returns the index of the named function table entry in form of a binary value.

The function table to be evaluated is accessed in the address space where DSLAPI is running. If DSLAPI is running in a different address space from MERVA ESA nucleus, this function table is not necessarily identical to the one used by the nucleus.

Table 30. Function Table Entry (FNT), DSLFNT

Field	Type	Size	Write	Description
FNTNAME	CHARS	8		Function name
FNTROUTN	CHARS	8		Name of routing table
FNTROUTA	ADDRESS			Address of routing table
FNTHSN	BINARY			Highest QSN used in this queue
FNTFQE	BINARY			First QKTE in this queue
FNTLQE	BINARY			Last QKTE in this queue
FNTECBA	ADDRESS			Address of trigger ECB
FNTSIZE	BINARY			Size for storing messages in queue
FNTTRESH	BINARY			Threshold number of this queue
FNTKFLD1	CHARS	8		Name of key field 1
FNTKLEN1	BINARY			Length of key 1
FNTKOFF1	BINARY			Offset to start of key 1
FNTKFLD2	CHARS	8		Name of key field 2
FNTKLEN2	BINARY			Length of key 2
FNTKOFF2	BINARY			Offset to start of key 2
FNTNXTNM	CHARS	8		Name of next function
FNTHCONM	CHARS	8		Name of hardcopy function
FNTTRAN	CHARS	8		Transaction name
FNTLTE1	CHARS	8		Logical terminal name
FNTLTE2	CHARS	8		Logical terminal name
FNTNSO	BINARY			Number of signed–on end users
FNTMSGST	BYTE			Message processing status
FNTDENT	BIT			New message generation allowed
FNTPROT	BIT			Screen protected
FNTKMSG	BIT			Keep message after print
FNTPRON	BIT			Noprompt only display allowed
FNTNRKY	BIT			Retype verification allowed

Table 30. Function Table Entry (FNT), DSLFNT (continued)

Field	Type	Size	Write	Description
FNTPROM	BIT			Noprompt processing allowed
FNTCHCK	BIT			Message check specified
FNTFRAME	BIT			Frame MSGID specified
FNTQUEST	BYTE			Queue status 1
FNTIGNAC	BIT			IMS only: ignore active status
FNTHLDC	BIT			Function in HOLD currently
FNTHLDI	BIT			Function in HOLD initial
FNTACTIV	BIT			Function in ACTIVE status
FNTNOTY	BIT			Notify = YES is specified
FNTDSLQ	BIT			Function with queue
FNTDUMQ	BIT			Function with dummy queue
FNTSTART	BIT			Start command executed
FNTRSI	BIT			Restart info in this queue
FNTFRAMT	CHARS	8		Name of top-frame MSGID
FNTFRAMB	CHARS	8		Name of bottom-frame MSGID
FNTUAPL	BINARY			User application action
FNTFMID	CHARS	1		Printer format ID
FNTCOMPF	CHARS	1		Compression format
FNTCOMND	BYTE			Valid commands
FNTCAUT	BIT			AUT command
FNTCDEL	BIT			DELETE command
FNTCOK	BIT			OK command
FNTCROU	BIT			ROUTE command
FNTXPND	BYTE			Field expansion request for function
FNTXPCO	BIT			Common names expansion
FNTXPPR	BIT			Private names expansion
FNTXPNC	BIT			Noprompt msg conditional expansion
FNTXPPC	BIT			Prompt msg conditional expansion
FNTXPCL	BIT			Clear is requested
FNTXPNO	BIT			No field expansion
FNTMLIM	BINARY			Message limit within IMS schedule cycle
FNTPFGN	BYTE			PF-key groupnumber
FNTMSGS2	BYTE			Message processing status 2
FNTMS2M2	BIT			SWIFT II display mode
FNTMS2FE	BIT			Four–eyes principle on
FNTMS2FQ	BIT			Check against all queues
FNTMS2FC	BIT			Any command forbidden
FNTCKAUT	BIT			Checkaut = Yes is specified
FNTXKEYS	BIT			Queue with extra keys
FNTPFKY	CHARS	8		User PF-key table name

Table 30. Function Table Entry (FNT), DSLFNT (continued)

Field	Type	Size	Write	Description
FNTFPGM	CHARS	8		Name of DSLEUD function program
FNTCOPYQ	CHARS	8		Name of copy queue
FNTRELA1	BINARY			Address of first related function
FNTRELA2	BINARY			Address of second related function
FNTMSGID	CHARS	8		MSGID for mapping and formatting
FNTNQE	BINARY			Number of queue elements in this queue
FNTQUST2	BYTE			Second queue status
FNTQ2DQF	BIT			No display with DQ filled
FNTQ2TOF	BIT			No TOF format in this queue
FNTQ2LST	BIT			Start msg sel with queue list
FNTQ2STS	BIT			Store = Small specified
FNTQ2STL	BIT			Store = Large specified
FNTQ2NM1	BIT			Key 1 = NOMOD
FNTQ2NM2	BIT			Key 2 = NOMOD
FNTQ2MQI	BIT			MQI attachment transaction
FNTLIID	CHARS	1		ID of list MCB
FNTLILEN	BINARY			Maximum length of list data
FNTCKIND	BYTE			Checkpoint/msglim indicator
FNTMLSET	BIT			Msglim set
FNTCKSET	BIT			Checkpoint set
FNTQUST3	BYTE			Third queue status
FNTQ3AUT	BIT			Function with automatic start
FNTINTQ	CHARS	8		Intermediate queue EDI-SWIFT conversion
FNTDESCR	CHARS	128		Descriptive text

Appendix E. 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

Trademarks

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-MQI Attachment User's Guide*, SH12-6714
- *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

- *CICS/ESA V4.1 Application Programming Guide*, SC33-1169
- *CICS/ESA V4.1 Application Programming Reference*, SC33-1170
- *CICS Transaction Server for OS/390 V1.3 Application Programming Reference*, SC33-1688
- *CICS Transaction Server for OS/390 V1.3 Application Programming Guide*, SC33-1687
- *CICS/VSE V2.3 Application Programming Guide*, SC33-0712
- *CICS/VSE V2.3 Application Programming Reference*, SC33-0713
- *IBM C/370 V2 Programming Guide*, SC09-1384
- *IBM C/370 V2 Programming Guide for VSE*, SC09-1399
- *IBM Language Environment for MVS & VM Programming Guide*, SC28-1939
- *IBM Language Environment for MVS & VM Programming Reference*, SC28-1940
- *IMS/ESA V5 Application Programming: Design Guide*, SC26-8016
- *IMS/ESA V6 Application Programming: Design Guide*, SC26-8728
- *OS PL/I V2 Programming Guide*, SC26-4307
- *TSO Extensions V2 REXX/MVS User's Guide*, SC28-1882
- *TSO Extensions V2 REXX/MVS Reference*, SC28-1883
- *VS COBOL II Application Programming Guide for MVS and CMS*, SC26-4045
- *VS COBOL II Application Programming Guide for VSE*, SC26-4697.

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

MERVA Requirement Request

Use the form overleaf to send us requirement requests for the MERVA product. Fill in the blank lines with the information that we need to evaluate and implement your request. Provide also information about your hardware and software environments and about the MERVA release levels used in your environment.

Provide a detailed description of your requirement. If you are requesting a new function, describe in full what you want that function to do. If you are requesting that a function be changed, briefly describe how the function works currently, followed by how you are requesting that it should work.

If you are a customer, provide us with the appropriate contacts in your organization to discuss the proposal and possible implementation alternatives.

If you are an IBM employee, include at least the name of one customer who has this requirement. Add the name and telephone number of the appropriate contacts in the customer's organization to discuss the proposal and possible implementation alternatives. If possible, send this requirement online to MERVAREQ at SDFVM1.

For comments on this book, use the form provided at the back of this publication.

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.

Send the fax to:

```
To: MERVA Development, Dept. 5640              Fax Number: +49-7031-16-4881
    Attention: Gerhard Stubbe                  Internet address:
                                               mervareq@de.ibm.com
    IBM Deutschland Entwicklung GmbH
    Schoenaicher Str. 220
    D-71032 Boeblingen
    Germany
```

MERVA Requirement Request

```
To: MERVA Development, Dept. 5640               Fax Number: +49-7031-16-4881
    Attention: Gerhard Strubbe                  Internet address:
                                                mervareq@de.ibm.com
    IBM Deutschland Entwicklung GmbH
    Schoenaicher Str. 220
    D-71032 Boeblingen      Germany
```

Page 1 of ______

Customer's Name	______
Customer's Address	______
	______
	______
Customer's Telephone/Fax	______
Contact Person at Customer's Location	______
Telephone/Fax	______
MERVA Version/Release	______
Operating System Sub-System	______
Version/Release	______
Hardware	______
Requirement Description	______
	______
	______
	______
	______
	______
	______
Expected Benefits	______
	______
	______

Readers' Comments — We'd Like to Hear from You

MERVA for ESA
Application Program Interface Guide
Version 4 Release 1

Publication No. SH12-6374-01

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
SH12-6374-01

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, Dept. 0446
Schoenaicher Strasse 220
71032 Boeblingen
Germany

Fold and Tape **Please do not staple** Fold and Tape

SH12-6374-01

Cut or Fold
Along Line

Program Number: 5648-B29

Printed in Denmark by IBM Danmark A/S

SH12-6374-01

Spine information:

IBM MERVA for ESA API Guide Version 4 Release 1