IMS

Open Transaction Manager Access Guide and Reference

Version 9

IMS

Open Transaction Manager Access Guide and Reference

Version 9

Note

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 153.

Quality Partnership Program (QPP) Edition (June 2004) (Softcopy Only)

This QPP edition replaces or makes obsolete the previous edition, ZES1-2354-00. This edition is available in softcopy format only. The technical changes for this version are summarized under “Summary of Changes” on page xiii.

Contents

Figures

Tables

About This Book

This information is available in PDF and BookManager formats, and also as part of the IMS Version 9 QPP Information Center. To get the most current versions of the PDF and BookManager formats, go to the IMS Library page at www.ibm.com/software/data/ims/library.html. To get the most current versions of these books for the information center, go to the IMS V9 Vendor and Quality Partnership Program Library page at www6.software.ibm.com/dl/ims02/imsv9lib-p, where you can find updated plug-ins and instructions on how to install them in your IMS Version 9 QPP Information Center.

This book is for IMS system and Transaction Manager administrators responsible for installation, design, customization, operation, and recovery procedures for Open Transaction Manager Access (OTMA) servers or clients. It provides reference information for IMS system definition, customization, application programming, data communication and system administration for OTMA. It also provides information on writing an OTMA client.

How This Book Is Organized

This book introduces OTMA, describes OTMA clients, discusses changes to IMS to support OTMA, and gives an overview of XCF for OTMA. This book contains the following chapters:

- Chapter 1, “Introduction to OTMA,” on page 1 introduces the OTMA environment.
- Chapter 2, “The OTMA Client,” on page 13 describes what an OTMA client is.
- Chapter 3, “Using IMS with OTMA,” on page 39 describes how IMS tasks change when using OTMA.
- Chapter 4, “OTMA Diagnostic Information,” on page 63 describes OTMA sense codes returned with negative acknowledgement messages.
- Chapter 5, “OTMA Message Prefix,” on page 69 contains the format of the OTMA message prefix.
- Chapter 6, “OTMA Architected Transaction Attributes,” on page 95 contains the syntax of architected command output.
- Chapter 7, “OTMA Callable Interface,” on page 99, which was an appendix in previous releases.
- “Frequently Asked Questions about OTMA,” on page 151, which contain frequently asked questions about OTMA.
- “Bibliography” on page 157 contains a list of related publications (other than those in the IMS library).

Prerequisite Knowledge

IBM offers a wide variety of classroom and self-study courses to help you learn IMS. For a complete list, see the IMS Web site at: www.ibm.com/ims.

Before using this book, you should understand:

- Basic IMS concepts
- z/OS XCF programming
- The IMS environment
- Your installation’s IMS system
- Your installation’s networks

- Administration of the IMS system and Transaction Manager

For definitions of terminology used in this manual and references to related information in other manuals, see the *IMS Version 9: Master Index and Glossary*.

How to Send Your Comments

Your feedback is important in helping us provide the most accurate and highest quality information. If you have any comments about this or any other IMS information, you can do one of the following:

- Go to the IMS Library page at www.ibm.com/software/data/ims/library.html and click the Library Feedback link, where you can enter and submit comments.
- Send your comments by e-mail to imspubs@us.ibm.com. Be sure to include the title, the part number of the title, the version of IMS, and, if applicable, the specific location of the text you are commenting on (for example, a page number in the PDF or a heading in the Information Center).

Summary of Changes

Changes to the Current Edition of This Book for IMS Version 9

The following information has changed significantly:

- Added new **Auto-One** option to Table 20 on page 85.
- Description of **One Only** option is changed in Table 20 on page 85.
- Added client flag to Table 13 on page 80.
- Added explanation of Send Only Message and Reroute Request to “Explanation of OTMA State Data Fields” on page 85.
- Added new section “Protecting Transactions with OTMA” on page 27.

Changes to This Book for IMS Version 9

This edition is a draft version of this book intended for use during the Quality Partnership Program (QPP). Contents of this book are preliminary and under development.

This edition contains the following changes:

- Information related to the OTMA ACEE aging value setting, which is described in Table 12 on page 69 and “Explanation of OTMA Message-Control Information Fields” on page 73.
- The section “OTMA Program-to-Program Switch Processing” on page 55, formerly titled IMS Program-to-Program Switch Processing, has been significantly rewritten.
- The section “OTMA C/I Hints and Tips” on page 102 provides new information for using OTMA’s callable interface.

Library Changes for IMS Version 9

Changes to the IMS Library for IMS Version 9 include the addition of new titles, the change of one title, and a major terminology change. Changes are indicated by a vertical bar (|) to the left of the changed text.

New and Revised Titles

The following list details the major changes to the IMS Version 9 library:

- *IMS Version 9: HALDB Online Reorganization Guide*

  The library includes new information: *IMS Version 9: HALDB Online Reorganization Guide*. This information is available only in PDF and BookManager formats.
- *IMS Version 9: An Introduction to IMS*

  The library includes new information: *IMS Version 9: An Introduction to IMS*.
- The information formerly titled *IMS Version 8: IMS Java User’s Guide* is now titled *IMS Version 9: IMS Java Guide and Reference*.
- The library includes new information: *IMS Version 9: IMS Connect Guide and Reference*. This information is available only in PDF and BookManager formats.

Terminology Changes

IMS Version 9 introduces new terminology for IMS commands:

type-1 command
A command, generally preceded by a leading slash character, that can be entered from any valid IMS command source. In IMS Version 8, these commands were called *classic* commands.

type-2 command
A command that is entered only through the OM API. Type-2 commands are more flexible and can have a broader scope than type-1 commands. In IMS Version 8, these commands were called *IMSplex* commands or *enhanced* commands.

Accessibility Enhancements

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products. The major accessibility features in z/OS products, including IMS, enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

User Assistive Technologies

Assistive technology products, such as screen readers, function with the IMS user interfaces. Consult the documentation of the assistive technology products for specific information when you use assistive technology to access these interfaces.

Accessible Information

Online information for IMS Version 9 is available in BookManager format, which is an accessible format. All BookManager functions can be accessed by using a keyboard or keyboard shortcut keys. BookManager also allows you to use screen readers and other assistive technologies. The BookManager READ/MVS product is included with the z/OS base product, and the BookManager Softcopy Reader (for workstations) is available on the IMS Licensed Product Kit (CD), which you can download from the Web at www.ibm.com.

Keyboard Navigation of the User Interface

Users can access IMS user interfaces using TSO/E or ISPF. Refer to the *z/OS V1R1.0 TSO/E Primer*, the *z/OS V1R1.0 TSO/E User's Guide*, and the *z/OS V1R1.0 ISPF User's Guide, Volume 1*. These guides describe how to navigate each interface, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Chapter 1. Introduction to OTMA

IMS™ Open Transaction Manager Access (OTMA) is a transaction-based, connectionless client/server protocol. Though easily generalized, its implementation is specific to IMS in a z/OS™ sysplex environment. The domain of the protocol is restricted to the domain of the z/OS Cross-System Coupling Facility (XCF).

OTMA addresses the problem of connecting a client to a server so that the client can support a large network, or a large number of sessions, while maintaining high performance.

Other solutions available today use network-based protocols, such as Systems Network Architecture (SNA). These protocols require a great amount of overhead because they are not transaction based.

In this chapter

What Is OTMA?

OTMA has similarities to network protocols. There are several architectural models for networks. Figure 1 shows two. The simplified four-layer model shown on the right is often used in descriptions of UNIX® networks. In the open systems interconnection (OSI) model, shown on the left, OTMA is the session layer. Both models have a Transport, Network, and Data Link layer. The OSI model also includes layers for Application, Presentation, and Session, and the simplified model includes a process layer. In the four-layer model, OTMA is the process layer.

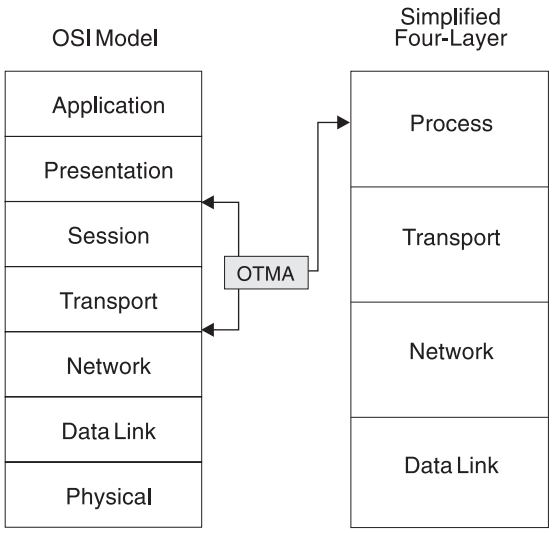

Figure 1. Network Architecture Models

OTMA, however, does not exactly conform to the OSI model, because OTMA can process several sessions simultaneously using a single transport connection, if the following are true:

- The z/OS Cross-System Coupling Facility (XCF) is the transport layer.

- A session is the connection between IMS and a client.
- A client or server only creates a single XCF connection.

OTMA performs some of the basic functions of the OSI transport layer (those not performed by XCF), so it is simplest to think of OTMA as a combined session and transport layer, with the transport layer comprised of both XCF and OTMA.

Although you can think of OTMA as a session and transport layer in a network architecture model, OTMA is designed to be a high-performance comprehensive protocol that allows z/OS programs to access IMS applications.

Definitions: A *z/OS program* in this case means any z/OS application that is a member of an XCF group that includes IMS. The XCF group members that IMS communicates with are called *OTMA clients*.

Related Reading: For more information on OTMA clients, see Chapter 2, "The OTMA Client," on page 13.

By using OTMA, each client (z/OS application) can submit transactions to IMS or issue IMS commands and receive output from IMS application programs and from IMS itself.

Definition: Because IMS can communicate with, or serve, many OTMA clients, IMS is called the *server*. However, OTMA only operates in the following IMS environments:

- IMS TM and DB (the IMS DB/DC environment)
- IMS TM with DB2® (the IMS DCCTL environment)

Capabilities of OTMA

The capabilities of OTMA include:

- Existing IMS application programs can run without modification and interact with OTMA clients. APPC/IMS application programs that use IMS `SETO` calls might need some modification.

  Related Reading: For more information on this restriction, see "OTMA Restrictions" on page 49.
- An OTMA client can issue most IMS commands and receive responses as a result of those commands.

  Related Reading: For information on the IMS commands that are supported, see the *IMS Version 9: Command Reference*.
- An OTMA client can indicate that no security checking is to be done for its messages, thereby minimizing security-processing overhead.
- The OTMA message flow and synchronization point protocols can be modified by an OTMA client for each transaction. In other words, the transaction-processing protocol used is not dependent on the current session.
- The IMS `/DISPLAY TRANSACTION` command output is in the form of an OTMA message returned to the client in the application-data section of the message prefix.
- OTMA-initiated transactions are identified to z/OS Workload Manager using the OTMA transaction-pipe name, which identifies the logical connection between IMS and OTMA.

Benefits of Using OTMA

Using OTMA provides the following benefits:

- Full-duplex processing provides an environment in which transactions and output messages are sent and processed in parallel.
- You can implement IMS device support outside IMS.

  You can also implement device support for your IMS subsystem that is different from what IMS provides, or enable device support that IMS does not provide.

  Figure 2 illustrates how IMS can communicate with a device, shown here as a workstation, using device support implemented within an OTMA client. IMS device support using VTAM® is shown for comparison.

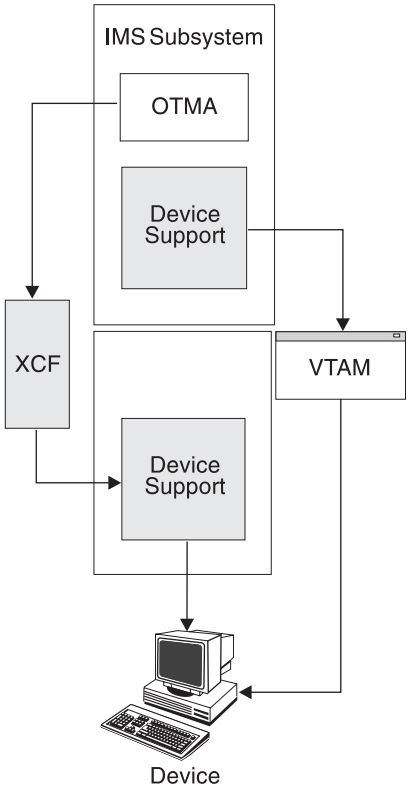

Figure 2. IMS communicates with a device using device support implemented within an OTMA Client

- Flow-control and transaction-processing attributes are dynamically bound to the transaction.
- Clients have high-performance access to IMS:
  - OTMA uses the z/OS XCF API.
  - OTMA does not use VTAM and IMS device-dependent support.
- Transactions based on different protocols (that is, that have different processing requirements such as being recoverable or irrecoverable) can be associated with a single transaction pipe.

  Related Reading: For more information on using transaction pipes, see “Using Transaction Pipes with OTMA” on page 8.
- You can connect up to 255 clients to the OTMA group.

- Messages can be extended using the user-data section of the message prefix, allowing additional user information to be sent with the transaction.
- User information and transaction pipe name are included within the messages themselves.
- Different clients can specify the same transaction pipe names, instead of needing to use uniquely named resources.
- No changes to existing IMS application programs are required.

  Exception: APPC/IMS application programs that use the SETO call might require changes.
- You do not need to use networking architectures, such as SNA (Systems Network Architecture).

Advantages of the OTMA Protocol

OTMA treats transactions as data objects that have attributes independent of application-, session-, or transport-layer considerations. OTMA is, in effect, a transaction layer, independent of other layers. As a unique layer, OTMA offers flexibility, simplicity, and performance that other solutions do not offer. OTMA provides the following transaction-specific services to the client:

- Grouping of transactions using transaction pipes.
- Security options (for example, the client can verify security or let the server verify the user ID).
- Dynamically-bound flow control and processing. The client can decide how transaction requests and responses are to be processed by the server.
- The ability for the client to query the server for transactions that the server supports.
- Treating transactions as objects. The client can include any pertinent user data with the transaction, and allow that data to stay with all messages generated by the transaction.
- The ability for the client to specify a client token with each transaction to correlate input with output.
- The ability for the client to control transaction processing performed by the server, in terms of:
  - Performance (the client can eliminate security-checking that the server performs).
  - Transaction grouping, using the transaction-pipe token.
- Client routing. An IMS exit routine can reroute an output message that is inserted to an alternate PCB to any OTMA client or to IMS.
- Architected command output. The client can use the IMS `/DISPLAY TRANSACTION` command to query the server's transaction attributes and receive the reply in a structured format. Therefore, the need for automated operator scripting to control processing is reduced.
- Unlike APPC, when using message flow through transaction pipes, no concept exists of a session that contains the flow-control parameters for all transactions and associated output data for the session.

How IMS Messages Flow in an OTMA Environment

Definition: The key to message flow for OTMA is the *transaction pipe*, the logical connection between the server and the OTMA client. An OTMA client includes the transaction-pipe name in the message-control information section of the message prefix for the input message. IMS then associates application output for an OTMA client with a specific transaction pipe.

Related Reading: For more information on transaction pipes, see "Using Transaction Pipes with OTMA" on page 8.

Basic OTMA Message Flow

The basic message flow is:

1. The client submits a transaction or command to IMS.
2. IMS accepts IMS transactions as input from any client.

   The IMS transaction code is specified in the application-data section of the input message.

   If the client is submitting an IMS command, the command is included in the application-data section of the input message.
3. The input message is processed.

   An IMS transaction is enqueued to the appropriate application program using an IMS scheduler message block (SMB).

   An IMS command is processed by IMS. The output is sent to the client synchronously or asynchronously, depending on the type of request.
4. Application output is sent to the client.

   Generation of output and commit are coordinated based on the commit mode specified in the state-data section of the message prefix for the input message.

   The application output is enqueued to a dynamically created IMS transaction-pipe structure (specific to that client) before being sent to the client.

   For an OTMA-submitted transaction, IOPCB output is returned to the OTMA client. By default, all alternate PCB output is also sent to the OTMA client. You can change this by coding the OTMA Prerouting exit routine (DFSYPRX0) or the client's OTMA Destination Resolution exit routine (DFSYDRU0). You can also use these exit routines to route alternate PCB output from non-OTMA-submitted transactions to OTMA clients.

   IMS delivers segmented messages in order, even though XCF does not guarantee sequential delivery of messages.

Figure 3 on page 6 shows an example of the message flow in an OTMA environment. Two clients are shown side by side in the example; they can be a TCP/IP client, an MQSeries® Queue Manager client, or a client of any other network type. Message flow starts with the client, goes through the XCF group, and to IMS. Within the IMS address space, a control region contains OTMA; the message flow ends at a transaction-pipe. The IMS application program issues a Get Unique (GU) call in the dependent region.

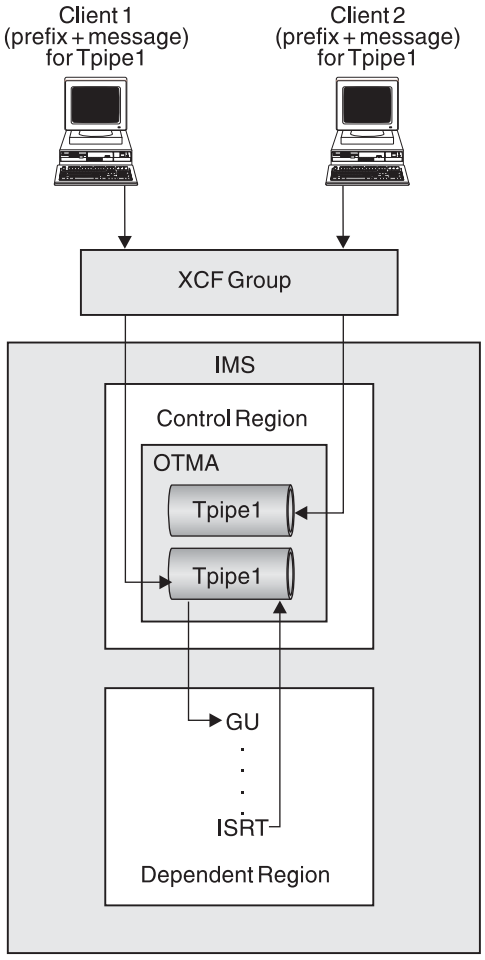

Figure 3. IMS Message Flow in an OTMA Environment

The notes in Figure 3 are as follows:

1. The message prefix is always attached to the input transaction, even in the case of segmented input. This prefix contains important information, such as the transaction-pipe name and the client token.

   A client application program can send several transactions specifying the same transaction-pipe name. The client token must always be present in the prefix, so that the client application program knows how to process the IMS output it receives.
2. OTMA clients do not need to predefine transaction pipes. Two different clients can use the same transaction-pipe name (as shown in Figure 3). Although many clients can use the same transaction-pipe name, each transaction pipe is unique. (In Figure 3, `client 1` and `client 2` both use `Tpipe1`, yet each is a unique transaction pipe.)

   A client can create and use as many transaction pipes as it needs.
3. The transaction-pipe structure is created dynamically when OTMA receives output and is used as an anchor for the application output.
4. The IMS application program has no knowledge of the OTMA message prefix when it issues the `GU` call.

IMS supports a full-duplex message flow for a client/server session. The client can instead request a half-duplex message flow, but this flow must be implemented and managed by the client itself:

- A correlator token in the state-data section of the message prefix can be used to uniquely identify a transaction. IMS maintains this field in the message prefix for a transaction.
- The client can set the response-requested flag in the message-control information section of the message prefix to receive a response for a message.
- Any unsolicited output from IMS is easily identified by a client, because the message prefix specifies only the transaction-pipe name. The client can ask IMS to discard the output.

  Unsolicited output should not interfere with half-duplex processing. That is, the client must be prepared for full-duplex flows while still maintaining a half-duplex flow on a user-token level. Contention should not be an error condition.

Sample Commit-Then-Send Transaction Processing Flows

Figure 4 shows a non-OTMA environment: a secondary logical unit type 2 (SLU 2) device communicates with IMS using VTAM and IMS device support (DDMs). The transactions are enqueued to the IMS message queues. Transaction output is returned to the SLU 2 device.

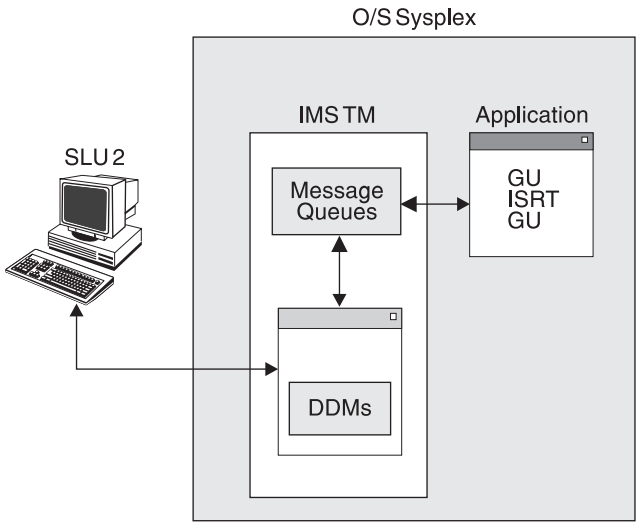

Figure 4. Standard SLU 2 Transaction Flow

Figure 5 on page 8 shows the same transaction flow in an OTMA environment. The transaction still comes from a SLU 2 device, but the device communicates with IMS using an OTMA client, through an XCF group, rather than VTAM.

Figure 5 on page 8 only shows the input flow, which begins with the SLU 2 device, goes to the OTMA client, through the XCF group, and ends at the OTMA server. The transaction is placed on the message queue, and the application issues get unique, insert, and get unique calls. Output follows the same path, in reverse. Of course, if a client is to send output to the SLU 2 device, the SLU 2 device must be defined to the client, and the client must be able to drive that device.

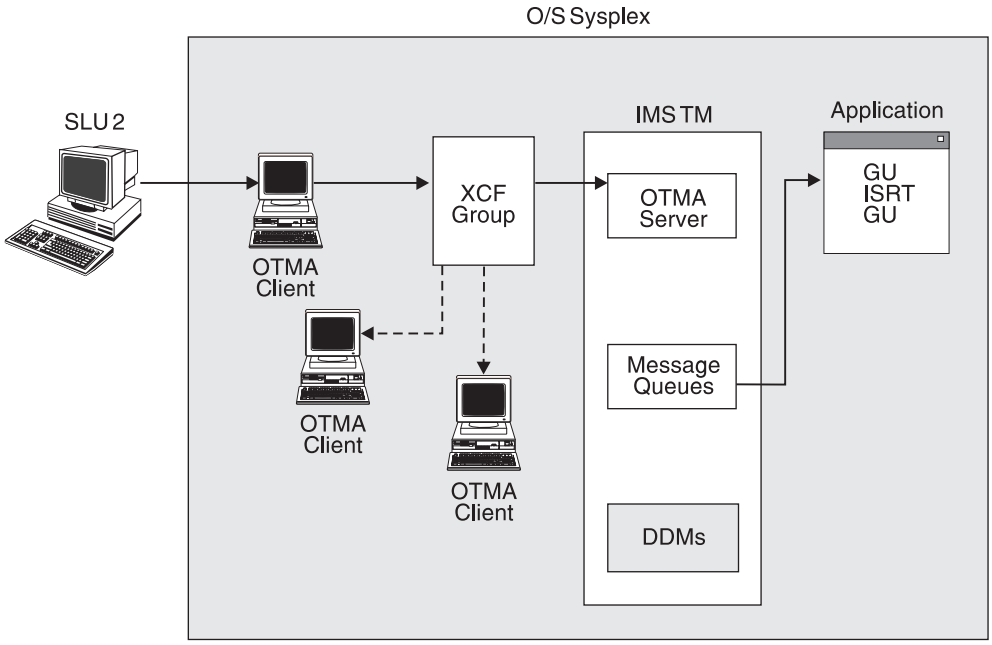

Figure 5. SLU 2 Transaction Flow Using OTMA

It might seem that the OTMA flow is more complex, and for a SLU 2 device, perhaps it is. But you can use OTMA to allow any type of device to communicate with IMS, not just VTAM-supported devices. An OTMA client can also act as a gateway for another network, such as a TCP/IP network.

Using Transaction Pipes with OTMA

An IMS transaction represents a request for IMS to do some work. Many transactions also require a response, after IMS has completed the work. So, each transaction has a source (the requester) and often a destination (for the response).

IMS uses the concept of a logical terminal (LTERM) to ensure that responses are associated with the correct requesters. An LTERM uses a queue where the transaction output is kept before it is returned to the requester.

Definition: For each LTERM, IMS maintains a connection between the queue and the physical node that receives the output. OTMA does not use an LTERM but still must maintain a connection between the client and IMS. This connection is the *transaction pipe*, or Tpipe.

Transaction pipes allow a client to associate its transactions with a transaction-pipe name. IMS uses the transaction-pipe name to associate all input and output with a particular client. The association between the transaction output and its ultimate destination (the originating device) is not made within IMS (as is the case with LTERMs), but is the responsibility of the client.

By using a transaction pipe, IMS does not know anything about the actual user of the transaction, often a user of the client application. Because IMS does not know anything about the actual user, the client has complete control over the output of transactions.

OTMA's use of transaction pipes provides:

- **Flexibility**

  Many transaction outputs can flow through the same transaction pipe.
- **Performance**

  Transaction pipes give the client the ability to specify and distinguish transactions based on their message-flow control and synchronization.
- **Resynchronization between a client and IMS**

  Transaction pipes can be either synchronized or non-synchronized. For a synchronized transaction pipe, all output messages are serialized through a single process, and sequence numbers can be assigned to messages. By logging these serialized messages, IMS and the client can resynchronize in the event of an outage.

  No resynchronization is required for a non-synchronized transaction pipe.
- **Object orientation**

  A transaction can be thought of as an object because OTMA keeps the transaction message information (such as user data and transaction-pipe name) within the message.

Figure 6 illustrates how transaction pipes fit in an OTMA client/server environment. As shown in Figure 6, transaction-pipe structures reside in the OTMA layer only for the server. XCF, which resides in the transport layer, can be thought of as an interprocess communication layer, because it provides communication between the client process and the server process.

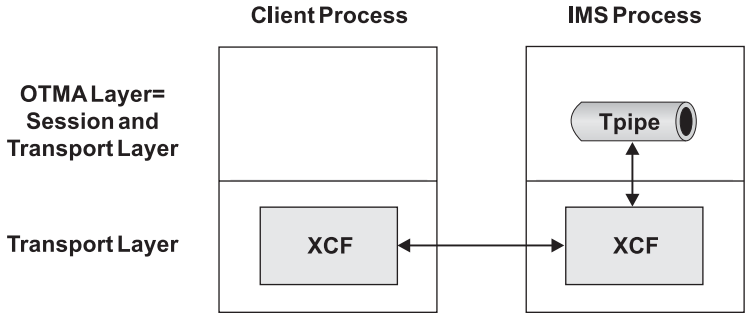

Figure 6. How Transaction Pipes Fit in an OTMA Client/Server Environment

Differences in Transaction Pipes

IMS LTERMs and UNIX pipes both provide a one-way flow for message traffic. An OTMA transaction pipe provides a two-way flow.

The concept of a transaction pipe is applicable to any protocol. In a general way, the transaction pipe replaces the IMS LTERM because:

- Processing is full duplex.
- Multiple flow-control mechanisms are possible.
- The logical output entity (in other words, the LTERM) is dissociated from the node of the actual user.
- The transaction pipe is implemented as a protocol rather than as an API, which facilitates a client/server architecture.
- The transaction pipe sets up a data-control mechanism independent of session characteristics, and is therefore transaction specific.

Message Flow Using Transaction Pipes

The flow control of transactions is handled by the client. The client dynamically binds flow-control parameters to the transaction by querying the transaction attributes in the server. Transaction pipes are not usually associated with flow control (except for synchronized transaction pipes using half-duplex processing).

Figure 7 shows the basic message flow between a client and a server, using XCF. The order of processing is:

1. The client sends a transaction as input to the server (IMS).
2. The server returns transaction output messages to the client.

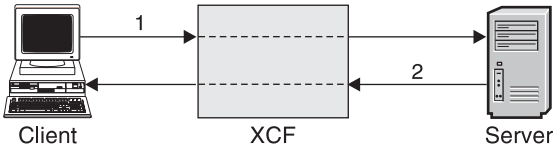

Figure 7. Basic Transaction-Pipe Message Flow

Within the server, the input transaction and the output messages are organized and synchronized using IMS queues, as shown in Figure 8 on page 11. The figure illustrates a commit-then-send transaction flow for a non-Fast Path environment.

The order of processing is:

1. The client sends a transaction to the server, and the server enqueues the transaction on a message queue.
2. The transaction is submitted to an application program for processing.
3. The application program prepares any output for the transaction and commits the output during sync-point processing.
4. The output is returned to the client.

Related Reading: For information on commit-then-send transactions, see “OTMA Commit Processing” on page 16.

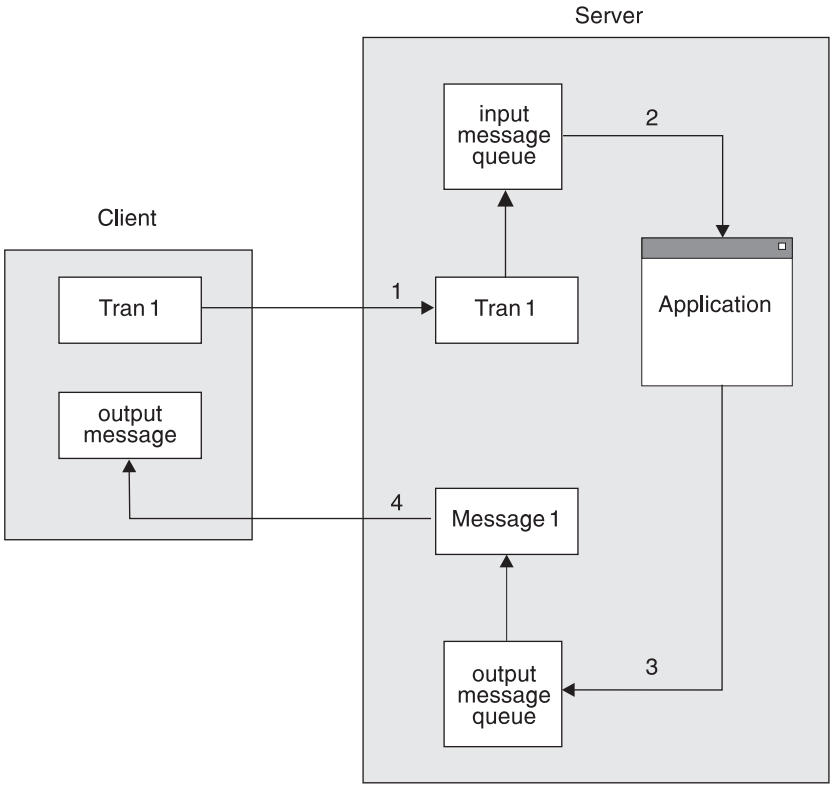

Figure 8. Use of Queues in the Transaction-Pipe Message Flow

In a full-duplex environment, transactions and output messages are being sent and processed in parallel, as shown in Figure 9 on page 12. This parallelism can be maximized by creating a process for every transaction and output message. The order of processing is:

1. The client sends a transaction (`Tran 1`) to the server, and the server's transaction pipe enqueues the transaction.
2. The transaction (`Tran 1`) is submitted to an application program for processing.
3. The application program enqueues any output (`Message 1`) for the transaction (`Tran 1`).
4. The client sends a second transaction (`Tran 2`) to the server and the server's transaction pipe enqueues the transaction.
5. The second transaction (`Tran 2`) is submitted to an application program for processing.
6. The output (`Message 1`) for `Tran 1` is returned to the client.
7. The application program enqueues the output (`Message 2`) for the second transaction (`Tran 2`).
8. The output (`Message 2`) for `Tran 2` is returned to the client.

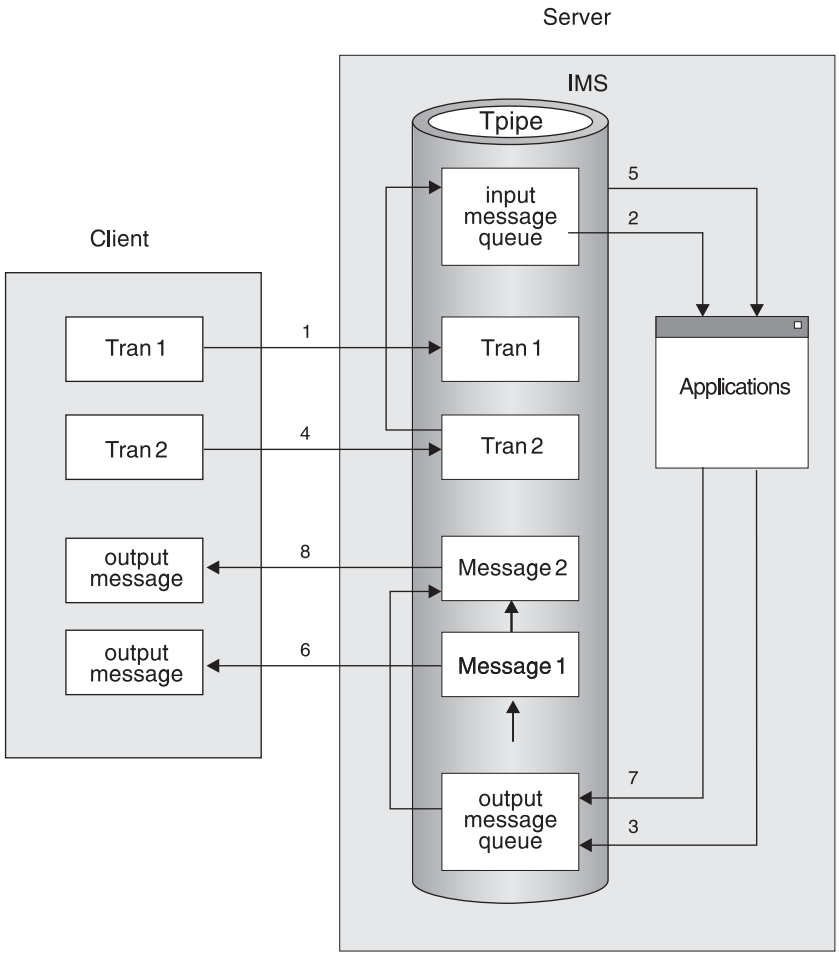

Figure 9. Transaction-Pipe Flow in Full-Duplex Environment

Chapter 2. The OTMA Client

The OTMA environment includes a server and one or more clients. This chapter explains how a client interacts with the server to process IMS transactions.

In this chapter :

What Is an OTMA Client?

Definition: An *OTMA client* is a z/OS application program that sends transactions to an IMS server and receives output. The application program must be a member of an XCF group and use the OTMA protocol.

Heterogeneous (non-z/OS) networks can connect with z/OS in many ways. Figure 10 shows some of the possible applications that use XCF. These include:

- MQSeries applications
- OEM applications
- IMS Connect applications
- DCE/RPC applications
- Other IBM® applications

Any of these can connect to an OTMA client to communicate with IMS.

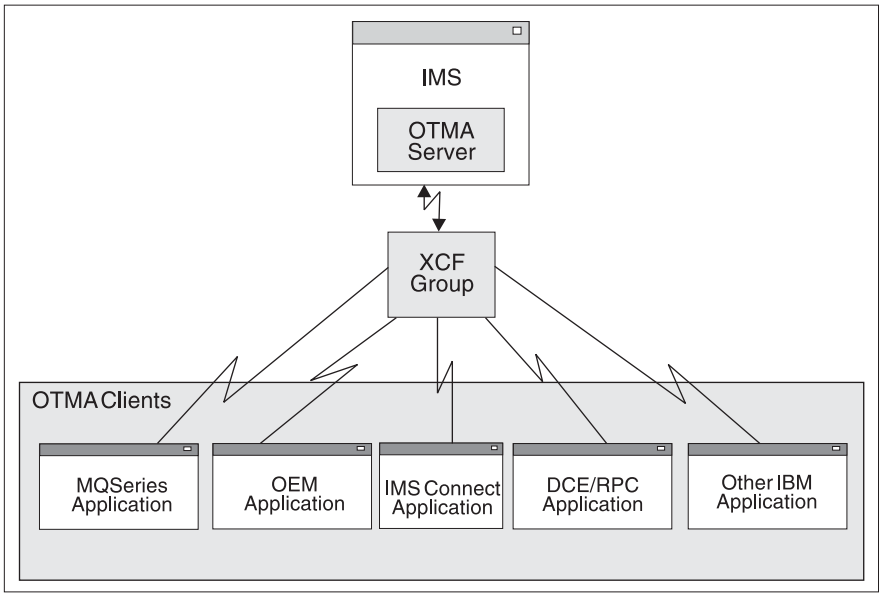

Figure 10. Applications that use XCF to connect to IMS on z/OS

An OTMA client is the gateway by which transactions from outside IMS can enter IMS.

OTMA processing involves:

1. A client sends a transaction or command to the server (IMS).
2. The server returns output to the client.

OTMA Naming Conventions

When naming either a client or a transaction pipe, you must adhere to the following conventions. The name:

- Must be character type A (A-Z, 0-9, @, $)
- Must begin with a non-blank character
- Must be padded with blanks if shorter than the maximum length (16 for a client name, 8 for a transaction-pipe name)
- Cannot contain embedded blanks
- Cannot be a reserved word (for example, “TO” or “SECURITY”)
- Cannot begin with “DFS” or “DBCDM”
- Cannot be an IMS keyword (for example, “LINE” or “NODE”)

In addition, transaction-pipe names cannot:

- Duplicate an IMS transaction name.
- Have the same name as the z/OS system console (for example, “WTOR”), IMS MTO, or secondary MTO.

IMS does not perform uppercase translation. If lowercase characters are used, the client receives a negative acknowledgment (NAK) response from the server.

Messages Sent by OTMA Clients

An OTMA client communicates with IMS by sending messages. First, a user enters application data using a device or program that is connected to the client. Next, the client adds some information (the message prefix) and sends the message to IMS. Output from IMS is sent to the client as a message, and the client uses the message prefix to route the data to the correct device or program.

Parts of the OTMA Message Prefix

The OTMA message prefix has the following sections:

- Message-control information

  This section includes the transaction-pipe name, message type, sequence numbers (if any), and various flags and indicators.
- State data

  This section includes a destination override (if any), map name, synchronization level, commit mode, tokens, and server state.
- Security data

  This section includes the user ID, user token, and security flags.
- User data

  This section includes any special information needed by the client.

Following the message prefix is the application-data section of the message. This section contains either the data to be sent to IMS for processing or the IMS response.

Related Reading: For more information on the OTMA message prefix, see Chapter 5, “OTMA Message Prefix,” on page 69.

OTMA Prefix Rules

Because a message can have a single segment or multiple segments, the following rules apply to OTMA message prefixes:

- Single-segment messages can have the full prefix (message-control information, state data, security data, and user data).
- Only the first segment of multi-segment messages has the full prefix. Subsequent segments are sent with only the message-control information and application-data sections.
- Acknowledgment (ACK or NAK) messages sent by IMS only return the first input buffer. This message carries the full prefix, and the application-data section (if it is included in the client request).

Sequence Numbers Used by OTMA

OTMA uses two types of sequence numbers for messages: send-sequence numbers and recoverable sequence numbers. Send-sequence numbers and recoverable sequence numbers are used differently in OTMA.

Using Send-Sequence Numbers

Send-sequence numbers are used for input and output messages. Send-sequence numbers should be incremented by a client for every input message. When IMS sends output to a client, the send-sequence numbers in the output message are also incremented. The send-sequence numbers are used for all the OTMA input/output messages. The send-sequence numbers in the input messages are also used to identify multi-segments.

Example: There is a two-segment OTMA input message. The first segment message will have `send-sequence number=XXX` and `segment number=1`. The second segment message should have the same `send-sequence number=XXX` and `segment number=2`. OTMA chains the two-segment message together because the send sequence numbers are the same.

OTMA uses send-sequence numbers in the following ways:

- All ACK and NAK messages from IMS use the send-sequence numbers submitted by the client on input.
- All OTMA commands that IMS sends to the client have send-sequence number 0 (zero). And, except for the resynchronization flows, these OTMA commands are all single segment.
- Send-sequence numbers for IMS error messages and IMS transaction output are set for each transaction pipe. The send-sequence number for a given transaction pipe is incremented by one for each message, and it is never 0 (zero). When the sequence number exceeds 4 294 967 295 (the 32-bit maximum), it is reset to 1.

Using Recoverable Sequence Numbers

Recoverable sequence numbers are used only to control resynchronization. If a client does not support resynchronization, `recoverable sequence number=0 (zero)`. Resynchronization is only valid for synchronized `TPIPE` and commit-then-send input/output. The recoverable sequence numbers are also incremented for every input/output message. Resynch support has an added logic to check if the recoverable sequence numbers are properly incremented. If the sequence numbers are not properly incremented, a NAK is sent. Because the resynch is dependent on

the recoverable sequence numbers, the resynch must be correct for every input/output. Recoverable sequence numbers apply to transaction pipes, which use them to control resynchronization.

Related Reading: For more information on resynchronization, see “Client/Server Resynchronization with OTMA” on page 28.

OTMA Commit Processing

OTMA can control how IMS commits transactions: they can be either commit-then-send or send-then-commit.

Definitions:

- For *commit-then-send* transactions (the IMS standard flow), IMS processes the transaction and commits the data before sending a response to the OTMA client.
- For *send-then-commit* transactions, IMS processes the transaction and sends a response to the OTMA client before committing the data.

Related Reading: For more information on these two commit modes, see “Sample OTMA Commit Processing Flows” on page 17.

For an OTMA transaction, a client can receive one of the following from IMS:

- An ACK message for the input, followed by any output messages.

  In addition send-then-commit transactions will also receive an ACK message followed by a “deallocate” flow (indicated when the commit-confirmation flag in the message-control information section of the message prefix is set to either Committed or Aborted).
- A NAK message with a sense code.
- A NAK message with the processing flag set to `Error Message Follows` in the message-control information section of the message prefix. The subsequent message has the same message prefix as the NAK message and has the IMS error message in the application-data section of the message prefix.

Summary of OTMA Commit Processing

Table 1 summarizes the differences between commit-then-send and send-then-commit processing. Several variables are listed in the first column; the differences between processing options are described in the next two columns. Following Table 1 are some usage notes to be aware of.

Table 1. Commit-Then-Send versus Send-Then-Commit Processing

Variables	Commit-then-send	Send-then-commit
Conversational	Client receives a NAK message.	Supported.
Fast Path	Client receives a NAK message.	Supported.
Non-conversational and non-Fast Path transactions	IMS commits after enqueuing the output to the client. The output is delivered later.	IMS sends output to the client and then commits.
Enqueue the input?	Yes.	Yes.
Enqueue the output?	Yes.	No.
Synchronized transaction pipe specified?	Supported.	Client receives a NAK message.

Notes:

- IMS conversations cannot use the commit-then-send commit mode.
- Send-then-commit input and output is irrecoverable.
- For irrecoverable output (send-then-commit), IMS requests an acknowledgement if the synchronization level is set to Confirm.
- For a recoverable transaction, IMS always requests an acknowledgement for an output message.
- For commit-then-send transactions, IMS always requests an acknowledgement.
- Synchronized transaction pipes can only be used for commit-then-send transactions.

Sample OTMA Commit Processing Flows

In order to explain the differences between the two commit modes, this section shows sample flows of data between IMS and clients for each commit mode.

Related Reading: For more detailed message flows, see “Sample OTMA Message Flows” on page 21.

Commit-Then-Send Flow

The commit-then-send flow, also known as the IMS standard flow, enqueues IMS output before sending it to the client. Use this flow for standard transaction processing. To use the standard flow, specify Commit Mode 0 in the state-data section of the message prefix. This sample flow assumes the following:

- The transaction pipe is synchronized. IMS maintains sequence numbers for recoverable input and output for the transaction pipe.
- Acknowledgment is always requested (by both IMS and the client).

If NAK is received by IMS, then the output is returned to the queue and will be delivered later.

The flow is illustrated in Figure 11. Following Figure 11 is a sequential list that provides more details on the flow.

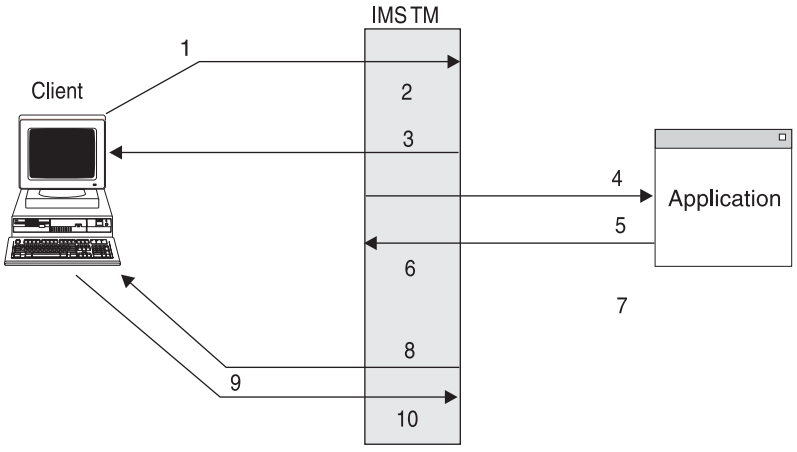

Figure 11. Commit-Then-Send (IMS Standard) Flow

The sequence of the flow illustrated in Figure 11 is:

1. Transaction initiated (response required/synchronized Tpipe)

2. Transaction is inserted to SMB
3. ACK
4. GU call followed by ISRT to IOPCB
5. Sync Point
6. Output is enqueued to Tpipe, and DB is committed
7. Transaction completes
8. Output is sent with response requested
9. ACK
10. Output is dequeued.

An example of the flow of the message activity for a single commit-then-send transaction pipe is shown in Figure 12. Following Figure 12 is a sequential list that provides more details on the flow.

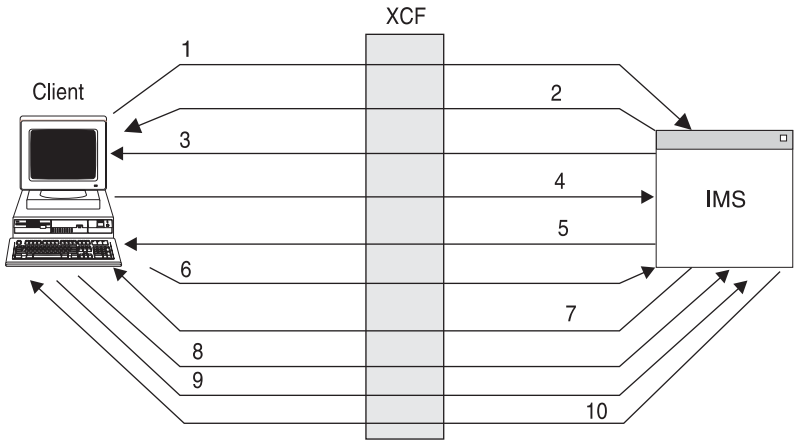

Figure 12. Sample Message Flow for Commit-Then-Send Flow

The sequence of flow shown in Figure 12 is:

1. Tran1
2. ACK to Tran1
3. Tran2
4. Output of Tran1
5. ACK to Tran2
6. Tran3
7. ACK to Tran3
8. Tran4
9. ACK to output of Tran1
10. ACK to Tran4.

Send-Then-Commit Flow

The send-then-commit flow sends IMS output before IMS completes synchronization-point (hereafter referred to as sync-point) processing. To use the send-then-commit flow, specify Commit Mode 1 in the state-data section of the message prefix. This sample flow assumes the following:

- The transaction pipe is not synchronized.
- The synchronization level is specified as `None` in the state-data section. Therefore, IMS does not request a response (an ACK) when sending output.

The flow is illustrated in Figure 13 on page 19. Following Figure 13 on page 19 is a sequential list that provides more details on the flow.

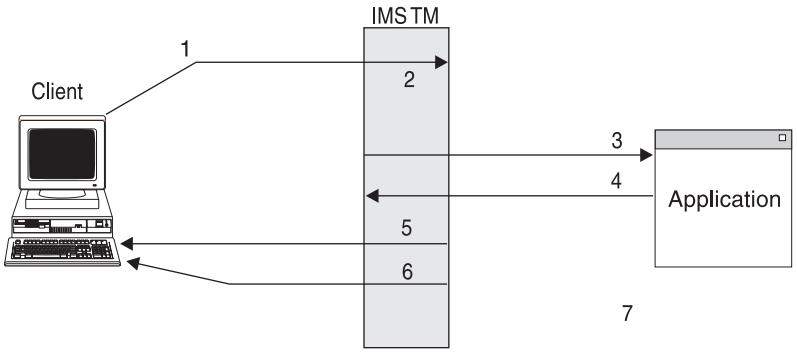

Figure 13. Send-Then-Commit Flow

The sequence of flow shown in Figure 13 is:
1. Transaction initiated
2. Transaction inserted to SMB
3. GU call followed by ISRT to IOPCB
4. Sync point started
5. Output is sent. No response is requested; response is requested only when sync=confirm is specified.
6. Commit confirmed; IMS completed sync point
7. Transaction completes

An example of the flow of the message activity for a single transaction pipe is illustrated in Figure 14. Following Figure 14 is a sequential list that provides more details on the flow.

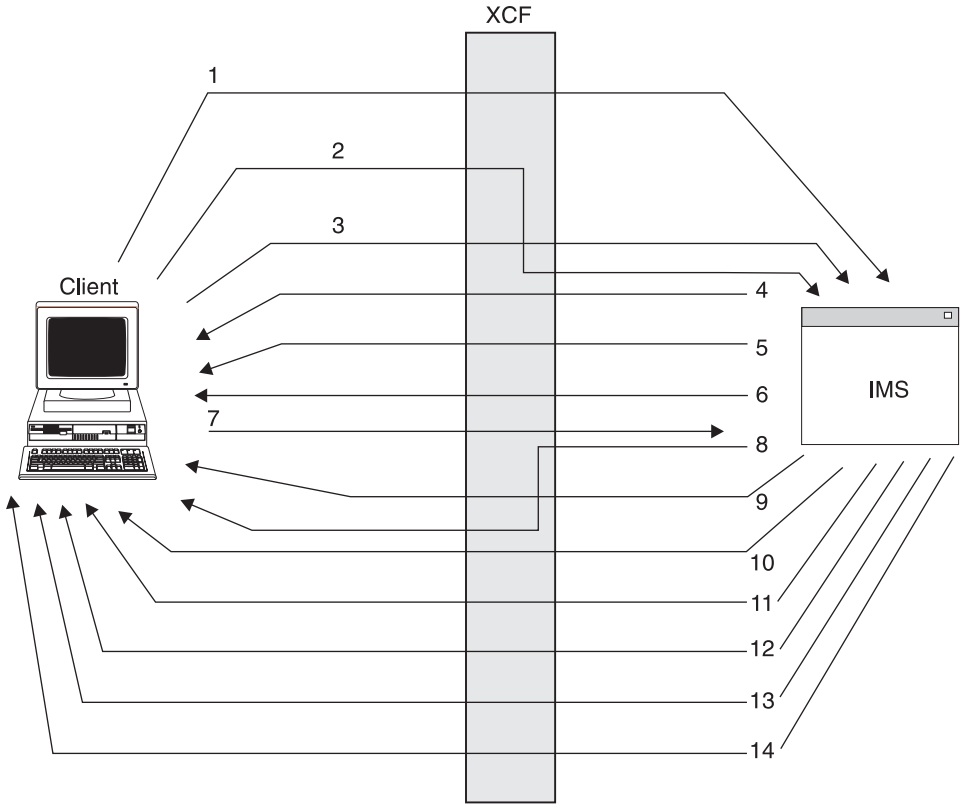

Figure 14. Sample Message Flow for Send-Then-Commit Flow

The sequence of flow shown in Figure 14 on page 19 is:
1. Tran1
2. Tran2 request/response
3. Tran3 request/response
4. ACK to Tran3
5. Output of Tran1
6. ACK to Tran2
7. Tran4
8. Output of Tran4
9. Confirm of Tran4
10. Output of Tran2
11. Output of Tran3
12. Confirm of Tran1
13. Confirm of Tran3
14. Confirm of Tran2

As shown in Figure 14 on page 19, the client can receive a confirmation for output before receiving the actual output, because XCF does not guarantee that all messages are sent in sequential order. The client must be able to handle this situation during message-receipt processing or by using the XCF Message exit routine.

Send-Then-Commit Flow with Confirm

The send-then-commit flow assumes no synchronization for the transactions as they are processed by IMS. This section shows a flow in which all transactions are confirmed as they are received (each message requests a response). The sample illustrated in Figure 15 assumes the following:

- Commit Mode 1 is specified in the state-data section of the message prefix.
- The transaction pipe is not synchronized.
- The Synchronization Level is specified as Confirm in the state-data section.

If NAK is received by IMS, then a user 119ABEND occurs in the application and IMS issues a `DFS554` message to the client.

Following Figure 15 is a sequential list that provides more details on the flow.

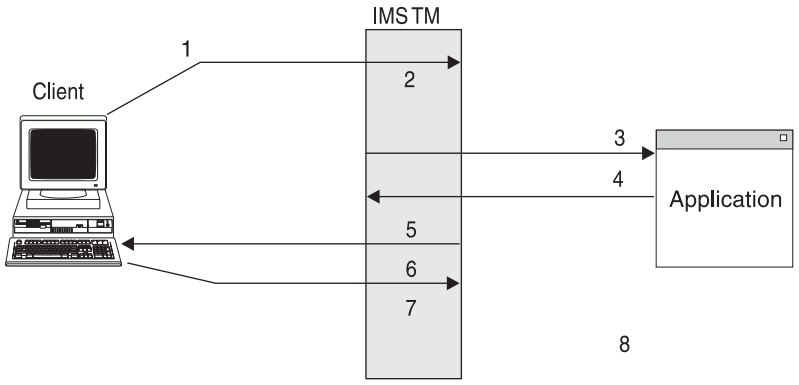

Figure 15. Send-Then-Commit with Confirm Flow

The sequence of flow shown in Figure 15 is:
1. Transaction initiated
2. Transaction inserted to SMB

3. GU call followed by ISRT to IOPCB
4. Sync point start
5. Output sent; response requested
6. ACK
7. DB is committed; commit is confirmed; IMS completed sync point
8. Transaction completed

Sample OTMA Message Flows

This section shows some sample message flows, and describes how various fields in the message prefix are set. In the figures, the following abbreviations are used for parts of the message prefix:

MC Message-control information section

SD State-data section

SE Security-data section

US User-data section

AP Application-data section

The sample flow diagrams show which parts of the prefix are mandatory for a given message and which are not applicable. Optional fields and prefix sections are enclosed in parentheses.

For transactions submitted by clients, the following principles apply:

- After IMS sends an ACK message to a client, IMS sends a commit confirmation (indicating that the transaction committed successfully or was aborted).
- The commit confirmation terminates a client transaction.

Related Reading: For examples that show what the message prefixes look like for various types of messages, see “Sample OTMA Messages” on page 92.

Client-Bid Message Flow

Figure 16 shows a client-bid flow, where the client attempts to connect to the server. This flow can occur when the client has already joined the XCF group and notices that a server has joined the group. The client-bid flow is:

1. Client-Bid: MC, SD, SE
2. ACK: MC, SD, SE

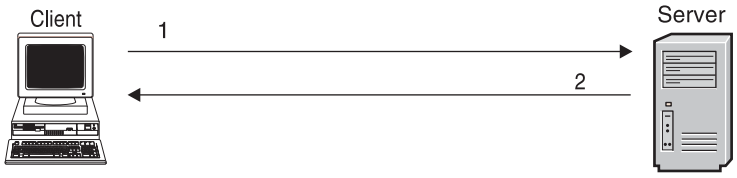

Figure 16. Client-Bid Flow

Table 2 on page 22 shows the contents of the message prefix.[1] The flow step is listed, with the message flow type, message prefix section, and associated contents of the message prefix section for the prefixes MC, SD, and SE.

1. The numbers used to show sequence (shown in Figure 16 and in the table text) are not part of the actual message prefix.

Table 2. Contents of Client-Bid Flow Message Prefix

Flow Step	Message Flow	Message Prefix Section	Contents of Prefix Section
1	Client-Bid	MC	Architecture level = 1 Message type = command Response flag = response requested Command type = client-bid Prefix flag = state data + security data
		SD	The state data format for command messages applies to these fields: Length Member name Originator's token Destination token (DRU exit name) MaxBlocksize Aging value Hash table size
		SE	(Utoken)
2	ACK	MC	Architecture level = 1 Message type = command and response Response flag = ACK Command type = client-bid Prefix flag = state data + security data
		SD	The state data format for command messages applies to these fields: Length Member name Originator's token Destination token (DRU exit name) MaxBlocksize Aging value Hash table size
		SE	Utoken

Server-Available Flow

Figure 17 on page 23 shows a Server-Available flow, where the server attempts to connect to the client. This flow only occurs when the server is already joined to the XCF group and recognizes that a client joins the group. A client should not wait for the server to recognize that it has joined the XCF group; the client should send its client-bid message as soon as it joins the group.

The client should ignore a Server-Available message after it has successfully completed its client-bid request and connected to the XCF group. The flow shown is:

1. Server-available: MC, SD, SE
2. Client-Bid: MC, SD, SE
3. ACK: MC, SD, SE

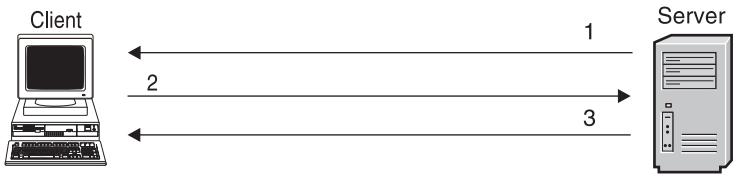

Figure 17. Server-Available Flow

Table 3 shows the contents of the message prefix. [2] The flow step is listed, with the message flow type, message prefix section, and associated contents of the message prefix section for the prefixes MC, SD, and SE.

Table 3. Contents of Server-Available Flow Message Prefix

Flow Step	Message Flow	Message Prefix Section	Content of Prefix Section
1	Server Available	MC	Architecture level = 1 Message type = command No response flag Command type = Server Available
		SD	The state data format for command messages applies to these fields: Length Member name Originator's token Destination token
2	Client-bid	MC	Architecture level = 1 Message type = command Response flag = response requested Command type = client-bid Prefix flag = state data + security data
		SD	The state data format for command messages applies to these fields: Length Member name Originator's token Destination token (DRU exit name) MaxBlocksize Aging value Hash table size
		SE	(Utoken)

2. The numbers used to show sequence (shown in Figure 17 and in the table text) are not part of the actual message prefix.

Table 3. Contents of Server-Available Flow Message Prefix (continued)

Flow Step	Message Flow	Message Prefix Section	Content of Prefix Section
3	ACK	MC	Architecture level = 1 Message type = command and response Response flag = ACK Command type = client-bid Prefix flag = state data + security data
		SD	The state data format for command messages applies to these fields: Length Member name Originator's token Destination token (DRU exit name) MaxBlocksize Aging value Hash table size
		SE	Utoken

Commit-Then-Send Transaction Flow

Figure 18 shows the flow for a commit-then-send transaction, where the client submits a transaction to the server for processing. The flow is:

1. Transaction ABC: MD, SD, SE, (US), AP
2. ACK: MC, SD, SE, (US)
3. Transaction output: MC, SD, (US), AP
4. ACK: MC, SD

Figure 18. Commit-Then-Send Transaction Flow

Table 4 on page 25 shows the contents of the message prefix. [3] The flow step is listed, with the message flow type, message prefix section, and associated contents of the message prefix section for the prefixes MC, SD, SE, US, and AP.

3. The numbers used to show sequence (shown in Figure 18 and in the table text) are not part of the actual message prefix.

Table 4. Contents of Commit-Then-Send Transaction Flow Message Prefix

Flow Step	Message Flow	Message Prefix Section	Content of Prefix Section
1	Transaction 'ABC'	MC	Architecture level = 1 Message type = transaction Response flag = response requested Transaction-pipe name Prefix flag = SD/SE/(US)/AP Send-sequence number
		SD	Length Synchronization flag = Commit Mode 0 Synchronization level = Confirm or None (Map name) (Correlator) Length of server user data = 0
		SE	Length (Security flag) Length of fields User ID length (User ID type = 02) (User ID) Profile length (Profile type = 03) (RACF® group) Utoken length (Utoken type = 00) (Utoken)
		(US)	This optional section is returned with the transaction output: Length User data
		AP	Length ZZ application data ('ABC' in the example)

Table 4. Contents of Commit-Then-Send Transaction Flow Message Prefix (continued)

Flow Step	Message Flow	Message Prefix Section	Content of Prefix Section
2	ACK	MC	Architecture level = 1 Message type = transaction and response Response flag = ACK Transaction-pipe name Prefix flag = SD/SE Send-sequence number
		SD	Length Synchronization flag = Commit Mode 0 Synchronization level = Confirm or None (Map name) (Correlator) Length of server user data = 0
		SE	Length (Security flag) Length of fields User ID length (User ID type = 02) (User ID) Profile length (Profile type = 03) (RACF group) Utoken length (Utoken type = 00) (Utoken)
		(US)	This optional section is returned with the transaction output: Length User data
3	Transaction Output	MC	Architecture level = 1 Message type = data Response flag = response requested Transaction-pipe name Prefix flag = SD/(US)/AP Send-sequence number Server token
		SD	Length Synchronization flag = Commit Mode 0 Synchronization level = Confirm or None (Map name) Server token (Correlator) Length of server user Data (Server user data)
		(US)	This optional section is returned with the transaction output: Length (User data)
		AP	Length ZZ Transaction output data

Table 4. Contents of Commit-Then-Send Transaction Flow Message Prefix (continued)

Flow Step	Message Flow	Message Prefix Section	Content of Prefix Section
4	ACK	MC	Architecture level = 1 Message type = data and response Response flag = ACK Transaction-pipe name Prefix flag = SD Send-sequence number
		SD	Length Synchronization flag = Commit Mode 0 Synchronization level = Confirm or None (Map name) Server token (Correlator) Length of server user data (Server user data)

Protecting Transactions with OTMA

This section describes protected transactions, how to use them, and the role of OTMA, APPC/IMS, and RRS in protecting transactions.

In a z/OS environment, this resource protection and recovery is managed by Resource Recovery Services (RRS), part of z/OS Recovery Resource Management Services (RRMS). RRS can apply coordinated changes across multiple mission-critical resources.

OTMA is one of two components that enable IMS to support protected transactions. The second component is APPC/IMS.

To process protected transactions, specify RRS=Y on the control region JCL. IMS then registers as a Resource Manager (RM) with RRMS using the CRGGRM service; it also sets its exits with the Context Services and RRS exit managers using the CRGSEIF service. IMS supports the following RRS exit routines:

- PREPARE
- COMMIT
- BACKOUT
- EXIT_FAILED
- ONLY_AGENT
- SUBORDINATE_FAILED

During initialization, IMS issues message `DFS0653I` to indicate that it has successfully connected with RRS and that it can now process protected transactions.

Initiating Protected Transactions from an OTMA Client

Following are the steps to initiate protected transactions from an OTMA client.

1. Specify Synclevel=2 (Syncpt) in the OTMA message prefix.
2. Obtain or reuse a context token. To obtain a context token, use the CTXBEGC service.

3. Set the context token in the OTMA message prefix.
4. Express interest in the UR using ATREINT.
5. Send the message to the OTMA client.
6. Wait for the output from the IMS transaction.
7. Send an acknowledgement (ACK) to IMS after the output is received.
8. Initiate the RRS commit (ATRACMT) or backout (ATRABCK).

The OTMA client assumes the server distributed syncpoint RM (SDSRM) role. This means that the OTMA client owns the context and is the only RM allowed to initiate or invoke the RRS commit. The flow is similar to that of an APPC/IMS-protected transaction, with the following exceptions:

- The OTMA client initiates the commit using the RRS Commit_Agent_UR service (ATRACMT).
- RRS then directly informs IMS to take a commit. As a result of the previous ATRACMT call, RRS drives the commit exits of all interested RMs.

Processing Protected Transactions in IMS

IMS receives the protected transaction and extracts the context token from the OTMA message header. IMS saves the context token in its own control block before placing the protected transaction on the IMS message queue. Also, before placing the protected transaction on the message queue, IMS expresses interest in the context using the Express_Context_Interest service (CTXEINT). IMS will therefore be informed if anything happens to the context while the protected transaction is queued on the message queue.

When IMS schedules the protected transaction into a dependent region, IMS switches the context token to the dependent region TCB using the Switch_Context service (CTXSWCH); it also expresses protected interest in the UR using ATREINT. IMS then presents the protected transaction to an application program that processes that particular transaction. The application contains the business logic (for example, update databases, send messages, and others). After the application completes its work, it reaches a commit point. IMS then sends the application output back to the OTMA client and waits for a commit or backout event from RRS.

Client/Server Resynchronization with OTMA

In order to guarantee that client transactions are processed and that they are processed only once, OTMA provides a protocol for synchronizing transactions. Using a synchronized commit-then-send (Commit Mode 0) transaction pipe, the client and IMS can regain message flow in the event of a client or IMS outage. Resynchronization occurs when either IMS or the client terminates normally or abnormally.

Transaction resynchronization achieves the following:

- Prevents data from being reprocessed
- Detects that data has not been received and causes the client to resend the data
- Detects that resynchronization might not be possible
- Allows the client to decide what actions to take in order to resynchronize

OTMA resynchronization is not symmetrical, and a system's behavior depends on its role: client or IMS. Resynchronization also does not maintain symmetry for send- or receive-sequence numbers. For example, the differences for the input and output sides of an IMS flow are:

Input IMS logs the client sequence numbers when the transaction is enqueued, and from that moment, the client has no control over dequeuing the transaction.

Output The application output is enqueued to a synchronized transaction pipe, but the output sequence numbers are not logged at that time. Only after sending the output and receiving an acknowledgment from the client does IMS finally dequeue the message and log the incremented sequence numbers.

All output using a synchronized transaction pipe is sequenced. The second output message is not sent until the ACK message from the client is received for the first output message.

Assumptions for OTMA Resynchronization

The OTMA resynchronization process is based on the following assumptions:

- Neither client nor IMS sends an ACK message until it has logged a transaction message.
- The client decides what resynchronization actions IMS should take.
- Both client and IMS can determine whether a transaction and its output messages are recoverable. The client can determine a transaction's recoverability using the architected form of the `/DISPLAY TRANSACTION` command.
- Recoverable OTMA messages include a value for the recoverable sequence number in the message-control information section of the message prefix. This value is incremented by 1 every time a recoverable message is sent using a Tpipe (see Table 5 on page 30, Table 6 on page 30, and Table 7 on page 31).
- A 0 (zero) is not a valid recoverable sequence number.
- Recoverable send- and receive-sequence numbers are maintained on a per transaction pipe basis.
- IMS does not support resynchronization for any IMS command input. If the client needs to submit IMS commands using a synchronized transaction pipe, the recoverable sequence number must be set to 0 (zero). If the recoverable sequence number is not set to 0 (zero), IMS rejects the command input with sense code X'0023'.

Recoverable OTMA Transactions

The recoverability of OTMA-initiated transactions and commands is determined by the following factors:

- Is it a recoverable or unrecoverable transaction?
- Is it a recoverable or unrecoverable command?
- Is the recoverable sequence number 0 (zero) or not?
- Is it a synchronized or nonsynchronized transaction pipe?
- Is it Commit Mode 0 (commit-then-send) or Commit Mode 1 (send-then-commit)?

A recoverable IMS transaction submitted using the send-then-commit transaction flow is not rejected. However, send-then-commit transactions are discarded during IMS restart (they are unrecoverable).

Unrecoverable OTMA Transactions

The following is true for unrecoverable transactions:

- The client must know that the transaction is unrecoverable, process it, and then forget about it.
- Send-then-commit output is unrecoverable, and it is not resynchronized.
- Send-then-commit transactions must be associated with nonsynchronized transaction pipes.

Summary Results of IMS Transactions and Commands

Table 5 summarizes the results of IMS transactions that a client submits under various processing conditions using a synchronized Tpipe. The summarization differentiates recoverable sequence numbers of zero and non-zero, and shows the differences between recoverable and unrecoverable transactions for commit modes 0 and 1 for both the zero and non-zero sequence.

Table 5. Results of IMS Transactions Using a Synchronized Tpipe

Recoverable Sequence Number	Commit-Then-Send (Commit Mode 0)		Send-Then-Commit (Commit Mode 1)	
	Recoverable Transaction	Unrecoverable Transaction	Recoverable Transaction	Unrecoverable Transaction
0 (zero)	Client receives ACK message. Output is recoverable, and no input/output recoverable sequence is updated.	Client receives ACK message. Output is not recoverable and no input/output recoverable sequence number is updated.	Client receives NAK message with sense code X'001C'.	Client receives NAK message with sense code X'001C'.
Not 0 (zero)	If the recoverable sequence number is valid, client receives ACK message. If it is not valid, client receives NAK message with sense code X'001F'. Transaction and output are recoverable.	Client receives NAK message with sense code X'0023'. Client should set recoverable sequence number to 0 (zero).	Client receives NAK message with sense code X'001C'.	Client receives NAK message with sense code X'001C'.

Table 6 summarizes the results of IMS transactions that a client submits under various processing conditions using a nonsynchronized Tpipe. The summarization differentiates recoverable sequence numbers of zero and non-zero, and shows the differences between recoverable and unrecoverable transactions for commit modes 0 and 1 for both the zero and non-zero sequence.

Table 6. Results of IMS Transactions Using a Nonsynchronized Tpipe

Recoverable Sequence Number	Commit-Then-Send (Commit Mode 0)		Send-Then-Commit (Commit Mode 1)	
	Recoverable Transaction	Unrecoverable Transaction	Recoverable Transaction	Unrecoverable Transaction
0 (zero)	Client receives ACK message. Transaction and output are recoverable.	Client receives ACK message. Transaction and output are not recoverable.	Client receives ACK message. Transaction and output are not recoverable.	Client receives ACK message. Transaction and output are not recoverable.

Table 6. Results of IMS Transactions Using a Nonsynchronized Tpipe (continued)

Recoverable Sequence Number	Commit-Then-Send (Commit Mode 0)		Send-Then-Commit (Commit Mode 1)	
	Recoverable Transaction	Unrecoverable Transaction	Recoverable Transaction	Unrecoverable Transaction
Not 0 (zero)	Client receives NAK message with sense code X'0023'. Client should set recoverable sequence number to 0 (zero).	Client receives NAK message with sense code X'0023'. Client should set recoverable sequence number to 0 (zero).	Client receives NAK message with sense code X'0023'. Client should set recoverable sequence number to 0 (zero).	Client receives NAK message with sense code X'0023'. Client should set recoverable sequence number to 0 (zero).

Table 7 summarizes the results of commands that a client issues under various processing conditions using a synchronized Tpipe or a nonsynchronized Tpipe. The summarization differentiates recoverable sequence numbers of zero and non-zero, and shows the differences between commit modes 0 and 1 for both synchronized and nonsynchronized Tpipes.

Table 7. Results of Commands that a Client Issues

Recoverable Sequence Number	Synchronized Tpipe		Nonsynchronized Tpipe	
	Commit-Then-Send (Commit Mode 0)	Send-Then-Commit (Commit Mode 1)	Commit-Then-Send (Commit Mode 0)	Send-Then-Commit (Commit Mode 1)
0 (zero)	Client receives ACK message. Command output is recoverable and output recoverable sequence number is updated.	Client receives NAK message with sense code X'001C'.	Client receives ACK message. Output is not recoverable.	Client receives ACK message. Output is not recoverable.
Not 0 (zero)	Client receives NAK message with sense code X'0023'. Client should set recoverable sequence number to 0 (zero).	Client receives NAK message with sense code X'001C'.	Client receives NAK message with sense code X'0023'. Client should set recoverable sequence number to 0 (zero).	Client receives NAK message with sense code X'0023'. Client should set recoverable sequence number to 0 (zero).

Related Reading:

- For information on the differences between recoverable and unrecoverable IMS transactions, see *IMS Version 9: Administration Guide: Transaction Manager*.
- For information on recoverability of APPC transactions in these circumstances, see “Integrity Tables” in *IMS Version 9: Application Programming: Design Guide*.

OTMA Resynchronization Protocol

OTMA resynchronization is based on the following command exchanges for each client:

CBresynch (Client_Bid resynch)
CBresynch is sent by the client to request resynchronization with IMS after both the client and IMS have successfully joined the XCF group.

SRVresynch (Server resynch)
SRVresynch must be initiated from IMS to the client after the client has successfully joined the XCF group and issued CBresynch. SRVresynch contains all synchronized Tpipe names of which IMS is aware.

REQresynch (Request resynch)
REQresynch must be issued from IMS to the client for each Synchronized Tpipe. REQresynch contains the Tpipe name, the IMS recoverable send-sequence number for the Tpipe, and the IMS recoverable receive sequence number for the Tpipe.

REPresynch (Reply resynch)
REPresynch is issued from the client for each Tpipe. REPresynch is a reply to the REQresynch request from IMS.

TBresynch (Tpipe_Bid resynch)
Tpipe_Bid resynch is issued by the client to initiate resynchronization with IMS for a particular Tpipe.

IMS keeps track of the send and receive numbers in the `TPIPE` structure. The send and receive numbers are updated for each input and output message. When resynch occurs, both the client and IMS share their send and receive numbers to verify that both sides are synchronized. The REQresynch command from IMS releases the send and receive numbers from IMS. The client accepts the numbers and does a comparison to the client's send and receive numbers. If the send and receive numbers are not the same, then the client specifies an action to IMS with the REPresynch command. If both sides have the same send and receive numbers, then the resynch completes successfully. If resynch fails, then the failing tpipe is identified and is not used.

Command message exchange for resynchronization must follow the OTMA resynchronization protocol. Normally, the sequence of events that occurs during resynchronization is:

1. The client issues CBresynch when the client attempts to resynchronize with IMS.
2. IMS sends an ACK to acknowledge receipt of CBresynch. From this point on, IMS quiesces any non-resynch type of input or output for all synchronized Tpipes. If IMS receives input while resynchronization is in progress for a synchronized Tpipe, the input is rejected with sense code X'0025'.
3. IMS builds the SRVresynch command and sends it to the client. The SRVresynch command lists all synchronized Tpipe names of which IMS is aware for that client.
4. The client receives the SRVresynch command and issues an ACK or NAK message to IMS.
5. If IMS receives an ACK message, IMS begins the resynchronization process for each Tpipe. IMS sends the REQresynch command that contains the Tpipe name, the IMS recoverable send-sequence number for the Tpipe, and the IMS recoverable receive-sequence number for the Tpipe.

   If IMS receives a NAK message from the SRVresynch command, IMS sends the DFS2393 message to the MTO and waits for a client-bid request or a CBresynch command from the client.
6. The client receives the REQresynch request. By comparing the information from the REQresynch request with its own information of the Tpipe, the client sends the REPresynch reply to IMS and informs IMS about the Tpipe

   Related Reading: For more information on REPresynch format, see "Format of OTMA State Data for REPresynch Command" on page 83.
7. IMS receives the REPresynch reply and takes actions on the Tpipe, based on the information from the client. IMS sends an ACK message to the client after it has taken actions dictated by the client. IMS enables the Tpipe to handle input

and output. If IMS cannot perform what the client has requested, IMS stops the Tpipe and sends a NAK message to the client.

8. If more than one Tpipe exists, steps 5 on page 32 to 7 on page 32 are repeated in parallel for each Tpipe. Other Tpipes that are not included in the SRVresynch request can send output in either direction anytime after step 4 on page 32.

Figure 19 illustrates the flow of nondeferred resynchronization. Following Figure 19 is a sequential list that provides high-level flow description.

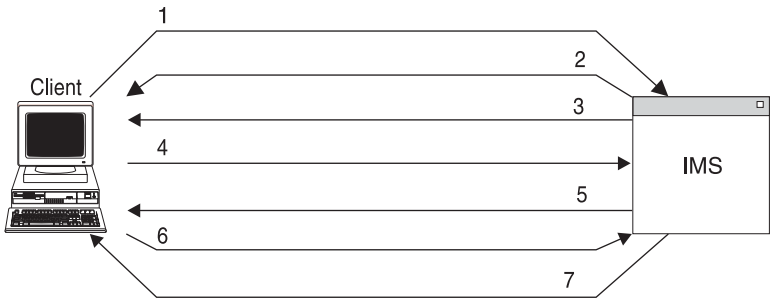

Figure 19. Flow of Resynchronization (Nondeferred)

1. Client-bid request with resynchronization
2. ACK message
3. SRVresynch command
4. ACK message
5. REQresynch command
6. REPresynch command
7. ACK or NAK message

If the client determines that resynchronization must be deferred for a particular Tpipe, the sequence of events for that Tpipe differs slightly:

In the REPresynch command, the client can set the “stop and wait for resynchronization” indicator, and can request that IMS defer any input or output while waiting for the TBresynch command from the client. Assuming steps 1 on page 32 to 4 on page 32 have completed, the events following are:

1. IMS sends the REQresynch command that contains the Tpipe name, the IMS recoverable send-sequence number and the IMS recoverable receive sequence number.
2. The client receives the REQresynch request. However, due to any product-specific reasons, the client defers resynchronization for this Tpipe by sending the REPresynch command with the “stop and wait for TBresynch” indicator on.
3. IMS sends an ACK message to acknowledge receipt of the REPresynch command and waits for TBresynch. Meanwhile, IMS quiesces input and output for the Tpipe. If IMS receives any input while waiting for TBresynch, IMS sends a NAK message to the client with sense code X’0025’.
4. The client sends the TBresynch command and requests IMS to resume resynchronization for this Tpipe.
5. IMS sends the REQresynch command that contains the Tpipe name, the IMS recoverable send-sequence number, and the IMS recoverable receive-sequence

number. If the associated Tpipe cannot be located using the client's TBresynch command, the client receives a NAK message with sense code X'0025'.

6. The client receives the REQresynch request. By comparing the information from REQresynch request with its own information about the Tpipe, the client sends the REPresynch reply to IMS and informs IMS about the Tpipe.
7. IMS receives the REPresynch reply and takes actions on the Tpipe, based on what the client has requested. IMS sends an ACK message to the client if it has taken actions dictated by the client. Otherwise, IMS sends a NAK message to the client with sense code X'0025' or X'0026'.

Figure 20 shows the flow of deferred resynchronization. Following Figure 20 is a sequential list that provides high-level flow description.

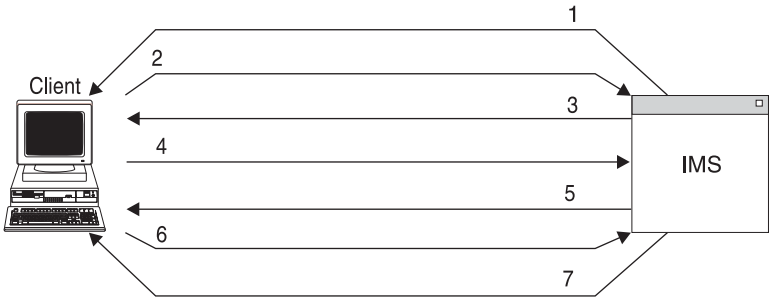

Figure 20. Flow of Resynchronization (Deferred)

1. REQresynch command
2. REPresynch command with STOP AND WAIT for TBresynch
3. ACK message
4. TBresynch command
5. REQresynch command
6. REPresynch command
7. ACK or NAK message

Sample OTMA Resynchronization Message Flow

This section provides a sample message flow for OTMA resynchronization.

Figure 21 on page 35 shows the flow of messages through a synchronized transaction pipe. Receive- and send-sequence numbers for each side (IMS and client) are represented by the letters **R** and **S**, which are set in the message-control information section of the message prefix. These numbers apply to the entire message (including multi-segment messages). **R** and **S** are not necessarily related.

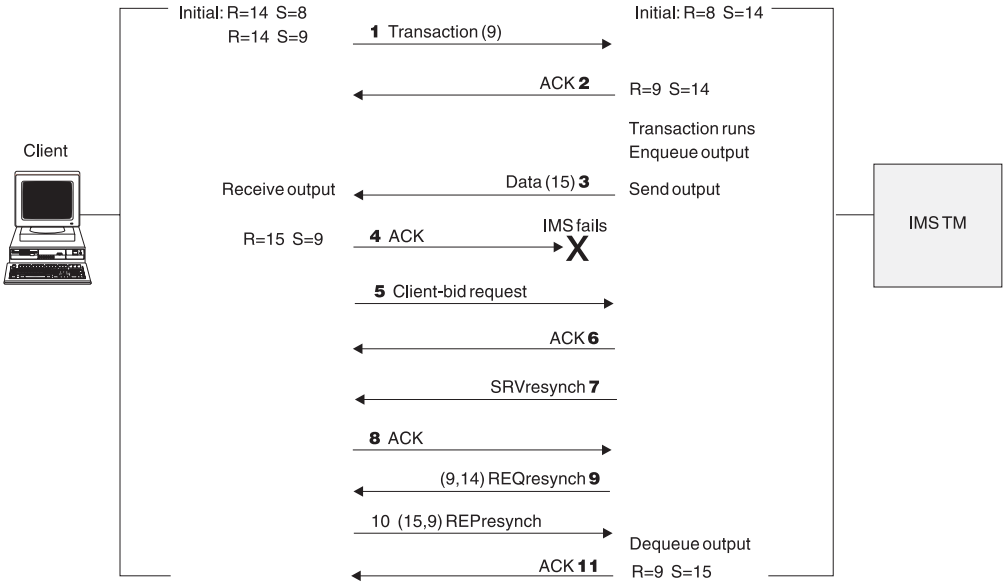

Figure 21. Sample OTMA Resynchronization Message Flow

1 After the client submits the transaction, IMS enqueues the transaction, and the transaction runs. The receive-sequence number is incremented by 1.

2 IMS sends the client an ACK message to acknowledge receiving and enqueuing the transaction.

3 IMS enqueues the output and sends the data to the client.

4 The client sends an ACK message to IMS to acknowledge receiving the output; however, IMS never receives this ACK to message 15 because of an IMS failure.

Resynchronization proceeds as follows:

5 The client sends a client-bid request to IMS to initiate resynchronization.

6 IMS sends an ACK message to the client that resynchronization will begin.

7 IMS sends the SRVresynch command to the client to begin resynchronization.

8 The client sends an ACK message to IMS to acknowledge receiving the SRVresynch command.

9 IMS sends the REQresynch command to the client to update the receive- and send-sequence numbers to (9,14).

10 The client sends the REPresynch command to IMS to update the receive- and send-sequence numbers to (15,9), and to tell IMS to dequeue the last output message. IMS dequeues message 15 and updates **S** to 15.

11 IMS sends an ACK message to the client.

Sample OTMA Resynchronization Messages

This section provides sample OTMA resynchronization messages.

Related Reading: For information on the format of these messages, see Chapter 5, “OTMA Message Prefix,” on page 69.

Figure 22 on page 36 shows the OTMA client-bid request with the resynchronization message.

```
MESSAGE CONTROL INFORMATION:

01102000 0C004040 40404040 4040A0C0     |......       .{|
00000000 00000000 00000000 00000000     |................|

STATE DATA:

0036D4D8 E7C3C6F6 40404040 40404040     |..MQXCF6        |
40400100 00010002 00010100 00020002     |  ..............|
0002C4C6 E2E8C4D9 E4F00800 00007FFF     |..DFSYDRU0....".|
FFFF0000 00650056 C3525100 5001A051     |........C...&...|
```

Figure 22. Client-Bid Request with Resynchronization Message

Figure 23 shows the ACK message to acknowledge receipt of CBresynch.

```
MESSAGE CONTROL INFORMATION:

01308000 0C004040 40404040 4040A0C0     |......       .{|
00000000 00000000 00000000 00000000     |................|

STATE DATA:

0036D4D8 E7C3C6F6 40404040 40404040     |..MQXCF6        |
40400100 00010002 00010100 00020002     |  ..............|
0002C4C6 E2E8C4D9 E4F00800 00007FFF     |..DFSYDRU0....".|
FFFF0000 00650056 C3525100 5001A051     |........C...&...|
```

Figure 23. ACK Message To Acknowledge Receipt of CBresynch

Figure 24 shows the SRVresynch command message.

```
MESSAGE CONTROL INFORMATION:

01102000 2C000000 00000000 0000A080     |................|
00000000 00000000 00000000 00010000     |................|

STATE DATA:

000AD1C2 D1F0F0F0 F0C50000 00000000     |..JBJ0000E......|
```

Figure 24. The SRVresynch Command Message

Figure 25 shows the ACK message sent by client to acknowledge receipt of SRVresynch.

```
MESSAGE CONTROL INFORMATION:

0130A000 2C000000 00000000 0000A080     |................|
00000000 00000000 00000000 00010000     |................|

STATE DATA:

000AD1C2 D1F0F0F0 F0C50000 00000000     |..JBJ0000E......|
```

Figure 25. ACK Message To Acknowledge Receipt of SRVresynch

Figure 26 on page 37 shows the REQresynch command message.

```
MESSAGE CONTROL INFORMATION:

01100000 3000D1C2 D1F0F0F0 F0C5A080     |......JBJ0000E..|
00000000 00000000 00000000 00000000     |................|

STATE DATA:

001AD1C2 D1F0F0F0 F0C50000 00020000     |..JBJ0000E......|
00020000 00000000 00000000 00000000     |................|
```

Figure 26. The REQresynch Command Message

Figure 27 shows the REPresynch command message.

```
MESSAGE CONTROL INFORMATION:

01100000 3400D1C2 D1F0F0F0 F0C5A080     |......JBJ0000E..|
00000003 00000000 00000000 00000000     |................|

STATE DATA:

001AD1C2 D1F0F0F0 F0C50000 00020000     |..JBJ0000E......|
00020000 00000000 00000000 00000000     |................|
```

Figure 27. The REPresynch Command Message

Figure 28 shows the ACK message sent by IMS to inform the client that resynchronization on a Tpipe successfully completed.

```
MESSAGE CONTROL INFORMATION:

01308000 3400D1C2 D1F0F0F0 F0C5A080     |......JBJ0000E..|
00000003 00000000 00000000 00000000     |................|

STATE DATA:

001AD1C2 D1F0F0F0 F0C50000 00020000     |..JBJ0000E......|
00020000 00000000 00000000 00000000     |................|
```

Figure 28. ACK Message for Successful Resynchronization

Chapter 3. Using IMS with OTMA

This chapter describes changes to IMS tasks for the OTMA environment, as well as how IMS operates in an OTMA environment.

In this chapter :

Installing OTMA

OTMA is not installed using the IMS INSTALL/IVP Dialog. To enable IMS to use OTMA, specify the XCF group name during system definition. To start OTMA, you can use the `OTMA=Y` startup parameter in the IMS procedure during IMS system definition.

Related Reading: For more information on IMS system definition, see *IMS Version 9: Installation Volume 2: System Definition and Tailoring*.

Specifying OTMA-Related Parameters

These are the OTMA-related parameters that you must specify in IMS PROCLIB member DFSPB*xxx*:

GRNAME=

Specifies the XCF group IMS is to join. The group name is one to eight uppercase alphanumeric characters or other valid characters ($, @).

IMS joins the XCF group either during IMS initialization (if `OTMA=Y` is specified) or as a result of an IMS `/START OTMA` command. If `GRNAME=` is not specified and `OTMA=N` is specified, IMS cannot join the XCF group.

If you specify `GRNAME=` and OTMA is started, you can use the `/DISPLAY OTMA` command to display the XCF status. You are not required to define any XCF information.

All OTMA clients must know the XCF group name. The group name for the OTMA clients and the IMS server must be the same.

If you use RACF for security, the `IMSXCF.group.member` (client member name) must be defined in the RACF FACILITY class.

OTMA=

Specifies whether OTMA is to be enabled. Values are `Y|N`. The default is `N`.

If you specify a valid group name for `GRNAME=`, you can use the `/START OTMA` command to enable OTMA later, even if you specify `OTMA=N` during system definition.

OTMAASY=

Specifies that a non-response transaction originating from a program-to-program switch be scheduled asynchronously. This parameter is for send-then-commit messages only. A DFS2082 message is not issued for a transaction scheduled asynchronously.

OTMAASY= can also be used in a multiple program-to-program switch environment to ensure that only the response transaction be scheduled synchronously.

The default is OTMAASY=N.

OTMANM=

Specifies the XCF member name that IMS uses for the group when IMS is not using XRF or RSR. The member name is 1 to 16 uppercase alphanumeric characters or other valid characters ($, @).

The OTMANM name can be specified in the IMS procedure or in the DFSPB*xxx* member. If OTMANM is not specified, IMS uses the IMS APPLID for the member name.

If IMS is using XRF or RSR, the XCF member name that IMS uses comes from the USERVAR name specified in the IMS procedure, in the DFSPB*xxx* member, or in the DFSHSB*xx* member. The OTMANM name is not used in this case.

Recommendation: Do not change the group name or the IMS member name during an IMS emergency or normal restart.

OTMAMD=

Specifies whether you want to have the member override function in the DFSYPRX0 user exit for a transaction invoked from an OTMA client. The parameter values are Y (Yes) or N (No). N means that for transactions that are invoked from a non-OTMA lterm, you can use the 16-byte member override field of the user exit parameter list to specify the OTMA client member name. You cannot, however, use the member override field for transactions that are invoked from an OTMA client. N is the default setting for the OTMAMD parameter. Y means that you can use the member override field of the user exit parameter list for both OTMA and non-OTMA invoked transactions.

OTMASP=

Using the OTMASP parameter provides the same function as setting the output flag of the OTMA destination resolution exit DFSYDRU0. Using an output flag in DFSYDRU0 indicates whether a synchronized tpipe needs to be created for OTMA output. If the only reason you code DFSYDRU0 is to set that output flag, you can use the OTMASP parameter instead.
The OTMASP parameter values are Y (Yes) or N (No). N, which is the default, means that non-synchronized tpipes are to be created for the OTMA output. Y means that synchronized tpipes are to be created for OTMA output.
If your organization uses both the DFSYDRU0 exit and the OTMASP parameter to control the kind of tpipe that gets created for OTMA output, the following table shows when synchronized tpipes will get created. Table 8 describes which Tpipes are created when the DFYDRU0 exit is used, and when OTMASP=Y or OTMASP=N.

Table 8. Tpipes created when both OTMASP parameter and DFYDRU0 exit used

DFSYDRU0 is set to...	If OTMASP=Y, result is...	If OTMASP=N, result is...
Create synchronized tpipe	Synchronized tpipe	Synchronized tpipe
Create non-synchronized tpipe	Synchronized tpipe	Non-synchronized tpipe

OTMASE=

Specifies the type of OTMA RACF security that you want to use, if any. The parameter values are as follows:

C OTMA RACF security is CHECK. IMS commands are checked against the CIMS class. IMS transactions are checked against the TIMS class.

F OTMA RACF security is FULL. The same type of security as CHECK, but additional checking is performed against dependent regions. F is the default value for the OTMASE parameter.

N OTMA RACF security is NONE. No calls to RACF are made.

P OTMA RACF security is PROFILE. Each OTMA message defines the level of security checking to be done.

Important: The `/SECURE OTMA` command overrides the value specified in the OTMASE parameter.

Specifying OTMA Descriptors

OTMA descriptors relate the client name with the OTMA Destination Resolution exit routine to be used. OTMA descriptors are optional. If no client descriptor is specified, the default exit routine, DFSYDRU0, is used. DFSYDRU0 can be overridden during a client-bid request.

OTMA descriptors are built during IMS initialization. The descriptors are included in DFSYDT*x* members of IMS.PROCLIB ("*x*" is the IMS nucleus suffix). All parameters are delimited with a blank. The following is the format of a descriptor:

Column	Contents
1	Descriptor type. *M* is for an OTMA descriptor.
2	Blank.
3-18	1- to 16-character client name, left-justified and padded with blanks if necessary. This is a required and positional parameter. Duplicate names are not allowed. The client name should follow the resource naming conventions. **Related Reading:** For more information on the naming conventions, see "OTMA Naming Conventions" on page 14.
19	Blank.
20-23	DRU=
24-31	OTMA Destination Resolution exit routine name. This is a required parameter. Duplicate exit routine names are allowed.
73-80	Sequence numbers. These columns are ignored by IMS.

If you have multiple clients at the same IMS system, list each client name on a separate line. Ensure that columns 3-18 are different for each client.

Customizing IMS for OTMA

This section describes the exit routines that you can use to customize IMS for OTMA.

OTMA-Supported Exit Routines

Three new exit routines support OTMA:

- OTMA Destination Resolution (DFSYDRU0)
- OTMA Input/Output Edit (DFSYIOE0)
- OTMA Prerouting (DFSYPRX0)

In addition, the following exit routines are supported by OTMA:

- Command Authorization (DFSCCMD0)
- Input Message Routing (DFSNPRT0)
- Queue Space Notification (DFSQSPC0)
- Transaction Authorization (DFSCTRN0)

Related Reading: For more information on the exit routines, see *IMS Version 9: Customization Guide*.

Using DFSYPRX0 and DFSYDRU0 OTMA Exit Routines to Determine Destination

OTMA allows transaction pipe names to be the same as IMS LTERM names or APPC/IMS TP names. To clarify whether a destination is for OTMA, IMS provides OTMA exit routines that can specify where IMS should look to resolve the destination names. The exit routines cannot change the actual destination name.

Determining the destination for an OTMA message requires two phases. In each phase, an OTMA exit routine can be called:

Phase 1 The Prerouting exit routine (DFSYPRX0) is called to determine the initial destination for the output.

The exit routine can determine whether the message should be directed to an OTMA client or to IMS TM for processing. The exit routine cannot determine the final destination (because insufficient parameters are passed to it).

Phase 2 The Destination Resolution exit routine (DFSYDRU0) is called to determine the final destination for the output. Each client can specify a separate Destination Resolution exit routine.

Both of these exit routines receive control when an IMS application program issues an `ISRT` call to an alternate PCB, or issues `CHNG` or `PURG` calls. But if the destination is the name of an IMS scheduler message block (SMB), the Destination Resolution exit routine does not receive control. Figure 29 on page 43 illustrates the two phases of message destination determination.

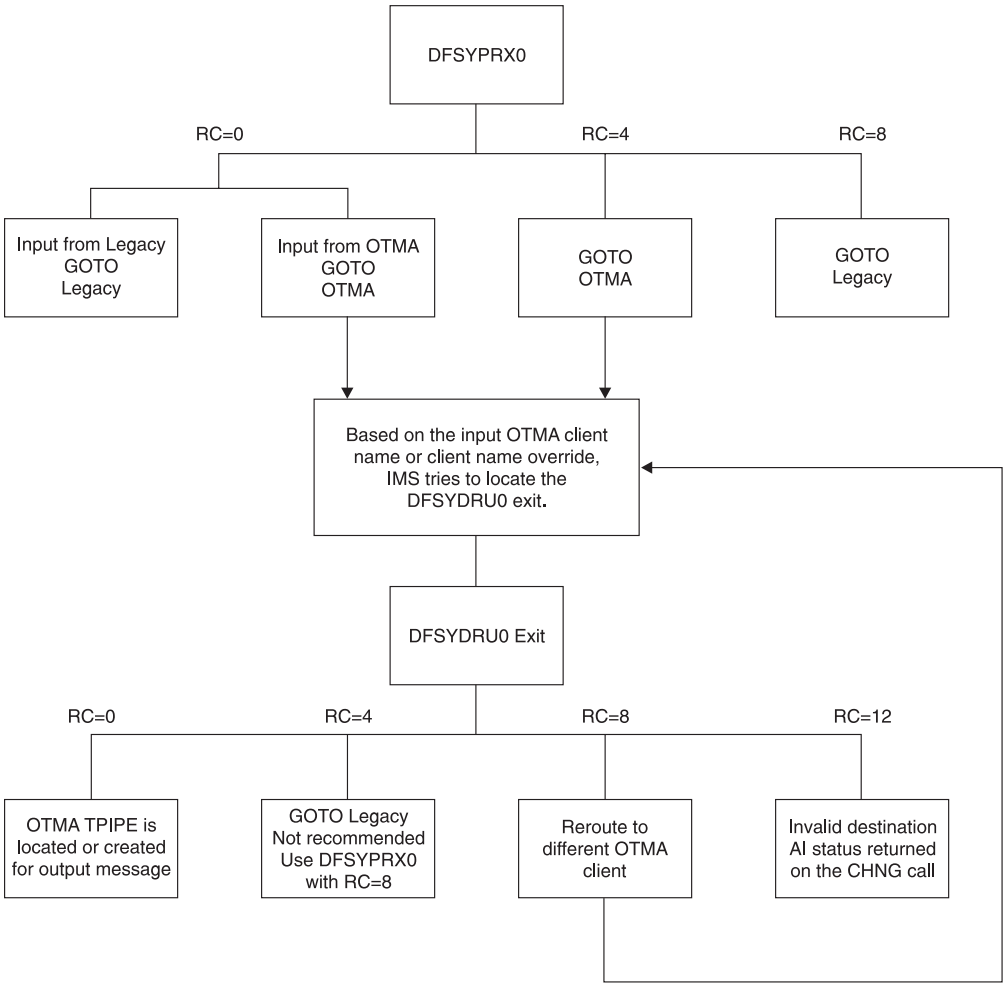

Figure 29. How DFSYPRX0 and DFSYDRU0 Determine Message Destination

Recommendations:

- Because of the potential conflict with the SMB name, OTMA clients should avoid using a transaction pipe name as either the transaction name or the routing key.
- DFSYPRX0 and DFSYDRU0 exits should be the same for the front-end and back-end IMS systems within a shared queues group. If the exit routines are different for one or more back-end IMS systems, asynchronous output might be sent to different destinations, depending on which back-end IMS system processed the input.

  To ensure prompt delivery of the output, enable OTMA on every back-end IMS system in the shared queues group. If a back-end IMS system does not have OTMA enabled, any asynchronous OTMA output that is inserted into an alternate PCB is simply queued and not delivered until the operator issues a `/STA OTMA` command.
- In an IMS subsystem, you can have many Destination Resolution exit routines, but only a single Prerouting exit routine.
- Specifying OTMAMD=Y in the IMS PROCLIB member DFSPB*xxx* can direct your OTMA message from the DFSYPRX0 exit to a different DFSYDRU0 exit without rerouting.
- Specifying OTMASP=Y in the IMS PROCLIB member DFSPB*xxx* always creates a SYNC tpipe for the ALT-PCB output message.

- The name of the DFSYDRU0 exit is determined by the user or an OTMA client. Each client can have its own dedicated DFSYDRU0 exit.
- The SCD and PST addresses are available in the input parameter for both OTMA user exits.
- The address of the first segment of the output message is not passed to either user exit.

Administering IMS for OTMA

This section describes IMS administration considerations for OTMA.

Related Reading: For more information on administering IMS, see *IMS Version 9: Administration Guide: Transaction Manager* and *IMS Version 9: Administration Guide: System*.

IMS Conversations and OTMA

OTMA-to-IMS conversations are send-then-commit and are nonrecoverable.

IMS creates a unique conversation ID for each transaction pipe that is saved in the Server token field in the OTMA state-data section of the message. This ID must be passed to IMS with all subsequent client input for the conversation using that transaction pipe.

The server-state flag is set to `Conversational State` in the state-data section of the message prefix when IMS acknowledges the transaction. This flag must be set for all subsequent client input during the conversation.

OTMA does not support the `/EXIT` command. If a client wants to terminate the IMS conversation, it sends a message with the commit-confirmation flag set in the message-control information section of the message prefix (indicating a deallocate flow). IMS then terminates the IMS conversation. A deallocate flow must specify `Conversational State` for the server-state flag in the state-data section of the message prefix; it must also set the server token field.

MSC and OTMA Transactions

You can use IMS Multiple Systems Coupling (MSC) with OTMA transactions. All IMS subsystems in the MSC network that process OTMA transactions must run under IMS/ESA® Version 5 or later; if you are using IMS shared queues, all IMS subsystems in the MSC network must run under IMS/ESA Version 6 or later. MSC processing in an OTMA environment is similar to MSC processing in an APPC/IMS environment:

- A client sends a transaction to IMS. If the transaction is defined as remote for that IMS system, it is sent to a remote IMS system for processing. If the transaction is defined as a local transaction and performs a message switch to another IMS system, the switched message is sent to that remote IMS system for processing.
- Output from the remote application program is returned to the originating IMS.
- IMS recognizes that the data is OTMA data and uses the transaction pipe to send the data to the client.

If the remote application inserts to an alternate PCB that is a remote destination, the data is not routed to an OTMA destination. The remote destination does not route the output message to the OTMA client, even though the message has a prefix. If the message is to be properly routed back to the original client, the remote

IMS must insert to a remote transaction. That transaction (at the original IMS site) must then send the message to the OTMA client using an alternate PCB and a Prerouting exit routine.

You can use MSC with OTMA in a shared-queues environment, as long as the MSC link exists in the front-end IMS subsystem that is connected to the OTMA client.

Fast Path and OTMA Transactions

Fast Path transactions must run as send-then-commit transactions. Any parameters with the OTMA transaction that contradict this commit mode cause the transaction to be rejected. Existing Fast Path application programs can run with OTMA if the client-entered transaction is properly defined.

IMS Restart Processing and OTMA

If an IMS subsystem connected to an XCF group must be restarted, IMS reconnects to the group during restart, but all clients must send new client-bid requests to IMS.

XRF Processing and OTMA

Recommendation: The active and alternate IMS subsystems should be in the same XCF group. If they are not in the same XCF group, after an XRF takeover, the client must connect to the new XCF group. If they are in the same XCF group, after an XRF takeover, the client is automatically reconnected to the XCF group.

The active IMS joins the XCF group specifying the IMS USERVAR name as its member name. The alternate IMS joins the group only after an XRF takeover, and connects using the same USERVAR name as the member name. If the active and XRF alternate subsystems are in the same XCF group, during XRF takeover, the new active IMS:

- Ensures that the old IMS member is disconnected
- Rejoins the group
- Begins delivering any queued output destined for an OTMA client, following client-bid processing for that client

IMS XRF does not track the `/START OTMA` and `/STOP OTMA` commands. The IMS procedures for both the active and alternate IMS should both specify `OTMA=Y` in order to ensure that IMS automatically rejoins the XCF group during XRF takeover.

In an XRF environment, OTMA clients are equivalent to Class 3 terminals and are not automatically reconnected to IMS. The clients detect that IMS has left the XCF group and wait for IMS (or the XRF alternate) to rejoin the group. Then the clients send new client-bid requests to IMS.

The alternate system in an XRF complex does not track when resynchronization has begun or the resynchronization of a particular Tpipe. If IMS fails during resynchronization, the client should detect the IMS failure, and try to resynchronize with the new XRF active system.

Queue Control Facility and OTMA

The IMS Queue Control Facility (QCF) replaces Message Requeuer (MRQ). MRQ processes messages in the background based on criteria that you provide;

however, it is accessible only with control statements issued in a BMP environment. You can access QCF both online with an ISPF interface and with control statements in a BMP environment.

The IMS QCF supports OTMA messages. You can use QCF to switch between all supported IMS releases, or between Shared Queues and non-Shared Queues. `TMEMBER` and `TPIPE` are the operands for the `INCLUDE` and `EXCLUDE` control statements.

TMEMBER A 1- to 16-character OTMA transaction member (client) name. You can generically specify groups of names that begin with the same characters by using an asterisk (*) after the characters that are the same. An asterisk as the first character will include or exclude **all** OTMA transactions.

TPIPE A 1- to 8-character OTMA transaction pipe name. You can generically specify groups of names that begin with the same characters by using an asterisk (*) after the characters that are the same.

To selectively recover OTMA messages, use the `INCLUDE` and `EXCLUDE` control statements. The format of the `INCLUDE` and `EXCLUDE` statements with OTMA operands is:

```
INCLUDE  operand(,)

EXCLUDE  operand(,)
```

operand must start in column 10 and is one of the following:

- `TMEMBER=tmember`
- `TPIPE=tpipe`

Example: To select all OTMA messages using transaction pipe name `S4A1BV6`, specify:

```
INCLUDE  TMEMBER=*,
         TPIPE=S4A1BV6
```

All messages with the same TMEMBER are grouped together, and the count is reported by the TPIPE name.

Example:

```
**** MESSAGES INSERTED BY DESTINATION ****
       BY OTMA DESTINATION
       TMEMBERNAME
         TPIPE1                          count
         TPIPE2                          count
```

If a client-bid request changes the name of the current OTMA Destination Resolution exit routine, any transactions enqueued before IMS terminates that are then reprocessed by the Message Requeuer might not use the changed exit routine name. Inserts to alternate PCBs use the exit routine name in the client descriptor.

With QCF, you can identify a category of message as well as the message type. Table 9 on page 47 describes category parameters, and the supported message types and keywords associated with the parameter.

Table 9. Selecting Messages by Category Type

Category parameter	Description	Supported message types and keywords
DESTYPE	Checks the destination of a message for a selected message type	APPC, LTERM, MSC, OTMA, LTRAN, RTRAN, TRANS, VSP
SRCETYPE	Checks the source of the message for a selected message type	APPC, MSC, OTMA, VSP
MSGTYPE	Checks the source or destination of the message for the selected message type	APPC, LTERM, MSC, OTMA, VSP

Related Reading: For more information about message selection by category in QCF, refer to *IMS Queue Control Facility for z/OS: User's Guide and Reference*.

Using Shared Queues with OTMA

This section describes general information about using IMS shared queues with OTMA.

To ensure delivery of alternate PCB processing, enable OTMA on all IMS systems; assign each IMS system in the shared queue group a unique XCF member name.

Use the `/DISPLAY TRANS ALL QCNT` to view all the OTMA transactions currently in the shared queue group waiting to be processed.

As the result of a temporary shortage in the HIOP storage pool, you might receive message `DFS1269E`, which notifies you of an internal IMS failure to register a shared queue resource. To re-register the shared queue resource for OTMA, issue the IMS commands `/STOP OTMA` and `/START OTMA`.

Related Reading: For more information on message `DFS1269E`, see *IMS Version 9: Diagnosis Guide and Reference* and *IMS Version 9: Messages and Codes, Volume 2*.

Related Reading: For more information on how IMS determines whether a message is for OTMA, see "Using DFSYPRX0 and DFSYDRU0 OTMA Exit Routines to Determine Destination" on page 42.

OTMA Commit-Then-Send Messages

OTMA commit-then-send (commit mode 0) messages can be processed on any IMS system in the shared-queues group. Program-to-program switches can also be run on any IMS.

OTMA Unsolicited Messages

OTMA clients must connect to every IMS system in the shared queue group in order to receive unsolicited messages. The OTMA client connections are necessary because transactions that might cause unsolicited messages can run on any IMS within the shared-queues group.

If the IMS that processes an unsolicited message (the backend system) is a different IMS than the one that receives the message, the unsolicited message is delivered by the back-end system. Therefore, OTMA must also be enabled on the

backend IMS. For example, message DFS555I, which notifies you that an application program abend occurred during transaction processing, is an unsolicited message that might be delivered by the back-end system.

OTMA Send-Then-Commit Messages

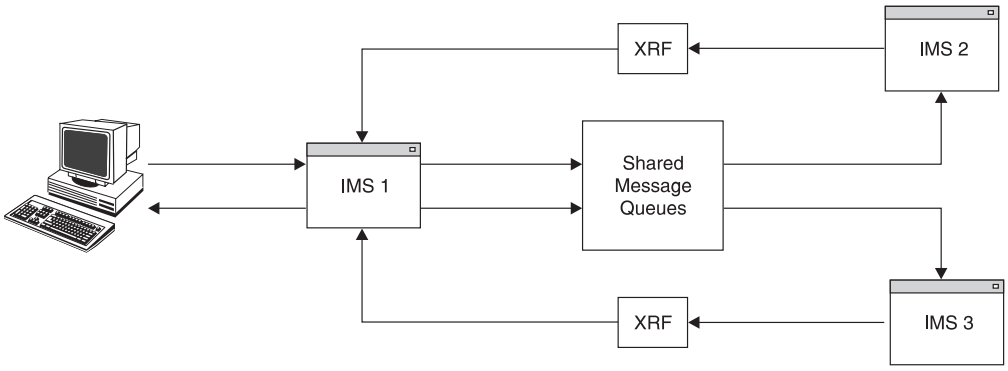

Figure 30. OTMA Messages Being Processed on Multiple IMS Systems in a Shared-Queues Group

OTMA send-then-commit messages can also be processed on any IMS system in the shared queue group. This is shown in Figure 30. Synchronous transactions created by a program-to-program switch always run on the same IMS system as the transaction that initiated the program-to-program switch. If the program-to-program switch initiates an asynchronous transaction, it can run on any IMS system.

In addition, program-to-program switching is not allowed for protected conversations (sync level 2). For more information on program-to-program switching, see “OTMA Program-to-Program Switch Processing” on page 55.

The commit levels for synchronous and asynchronous transactions are shown in Figure 31.

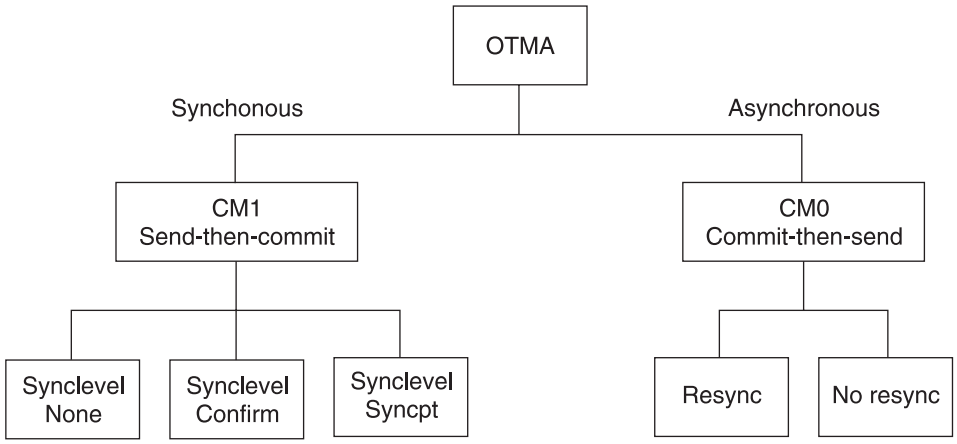

Figure 31. Synchronous and Asynchronous Transactions and Their Respective Commit Levels

All of the IMS systems in the shared queue group must be on IMS Version 8 or 9, with z/OS® Version 1 Release 2 or above, to enable the shared queue function for send-then-commit messages.

Use the DBRC INIT.RECON MINVERS(81) or CHANGE.RECON MINVERS(81) command to ensure that all of the IMS systems are Version 8 or above.

Use the /DISPLAY ACTIVE command to determine whether the shared queue function for OTMA send-then-commit is active.

Using Other IMS Commands

The IMS command /DISPLAY TMEMBER *membername*TPIPE *tpipename* QCNT shows the tpipe status and the output message queue count in a shared queue for a particular IMS system.

OTMA Restrictions

The following are general restrictions for the OTMA environment:

- The maximum total length of all prefixes for an OTMA message is 4096 bytes. This length does not include any application data.
- Existing IMS application programs that use SETO calls might not run as expected. APPC/IMS application programs using SETO calls might require modification to use implicit OTMA support.
- IMS conversational and Fast Path transactions must be defined as send-then-commit. Existing Fast Path applications can run with OTMA.
- A transaction from an IMS terminal (for example, a SLU 2 terminal) cannot route output directly to a client, but must use an OTMA Prerouting exit routine (DFSYPRX0).
- OTMA does not support the IMS Message Format Service (MFS). However, the MFS message output descriptor (MOD) name can be specified by the client in the prefix of an OTMA message.
- OTMA does not support IMS Front-End Switch.
- OTMA messages cannot be encrypted.
- All user IDs must be verified by RACF, unless the client specifies no security checking in the security-data section of the message prefix.
- IMS modules that contain XCF macros must be reassembled for new releases of IMS.
- OTMA has read only access to MSDB. No update access is available to MSDB from OTMA.
- OTMA does not operate in the IMS DBCTL environment.
- OTMA does not allow IMS terminal control commands like but not limited to /EXIT, /FORMAT, /HOLD, /RCL, and /SIGN commands.

Managing System Resources and OTMA

In an IMS-OTMA environment, several things can influence how IMS system resources are used. This section describes these system resource considerations.

IMS Message Queue Data Set Size and OTMA

Messages entering IMS from OTMA contain both the OTMA message prefix and other existing IMS message prefixes. The OTMA message prefix is variable in length. Excluding the user data section, the OTMA message prefix can become very large, sometimes over 200 bytes in length. The OTMA message prefix, including the user data section, is stored on IMS message queue data sets, which increases usage of the queue buffer pool.

Recommendation: Because of this increase in queue buffer pool usage, try to increase the size of the message queue data sets.

Buffer Pool Usage for OTMA

If an IMS-OTMA environment has heavy OTMA traffic, a significant increase in LUMP and HIOP pool usage can occur. Because LUMP and HIOP pools are allocated from private storage, you might need to increase the size of the IMS control region. Also, certain OTMA control blocks are allocated from ECSA, another limited resource.

Recommendation: Increase the ECSA size according to your workload. For example, if a client is sending more than 20 messages over 100 Tpipes within a few seconds, try increasing the IMS control region size to 200MB or more, and increase the ECSA size to 50MB or more. If you cannot increase the IMS control region size or the ECSA size, try balancing your workload to allow IMS to reuse its buffers more effectively.

Tpipe Number Recommendations for OTMA

Because Tpipes consume significant amounts of IMS resources and processing time, try to limit the number of Tpipes for each Tmember.

Recommendation: Restrict the number of Tpipes to 100 or less for each Tmember.

Dependent Region Occupancy and OTMA

A send-then-commit transaction remains in a dependent region while the output is being sent (before a sync point occurs).

Recommendations:

- If many of your transactions are send-then-commit transactions, increase the number of dependent regions to improve throughput performance.
- Use as many commit-then-send OTMA transactions as possible.

OTMA Security Overhead

You can reduce security overhead in IMS by letting the clients perform all security checking. If you set the no-security-checking flag in the security-data section of the message prefix, you can avoid much of the RACF overhead and improve IMS performance.

Related Reading: For more information on security, see “Establishing Security for OTMA.”

Establishing Security for OTMA

This section describes usage information for the /SECURE OTMA command, what the four OTMA security levels are, and lists some general security considerations for OTMA.

Using the /SECURE OTMA Command

This section describes usage information for the `/SECURE OTMA` command. Complete information for how to use this command is provided in *IMS Version 9: Command Reference*.

- Add transactions or commands that must be protected to the TIMS or CIMS classes, respectively. If the transaction is not in the TIMS class, or the command is not in the CIMS class, the transaction is allowed regardless of the option you set with the /SECURE OTMA command.

  When you enter an OTMA command, OTMA issues a RACHECK to validate the command. OTMA passes only the command verb to DFSCCMD0 for verification, not the entire CVB control block.
- The client-bid request for a client includes:
  - Access control environment element (ACEE) aging value.
  - The client user token.

    The user token is optional (except when RACF security is used) and is identified in the security-data section of the message prefix.
  - After RACF returns an ACEE address for a verified user ID to IMS, the ACEE is used in a subsequent (or second) call to RACF. It is used to determine the user ID's authorization to the IMS command or IMS transaction requested in the input message. The ACEE for each user ID and the ACEE expiration value are saved in an OTMA table. The ACEE expiration value is specified during the client-bid time. If a user ID is revoked and the ACEE is not expired, you must issue /STOP and /START OTMA to rebuild the ACEE table.
- Use the IMS startup/execution parameter of OTMASE to globally specify the OTMA security level for all the OTMA clients. Alternately, issue the /SEC OTMA command to specify the security after IMS is started. However, you cannot set the OTMA security level for an individual OTMA client.
- If you specify /SECURE OTMA NONE, IMS does not use RACF for security verification, regardless of what security is specified by the class for a client-bid request or for transactions. The REFRESH option allows you to dynamically refresh cached security ACEE values for user IDs of one individual OTMA member or multiple OTMA members. Therefore, the only commands that can be used are the following default commands:
  - BRO
  - LOC
  - LOG
  - RDI
  - UNL

Selecting an OTMA Security Level

This section describes the five security levels common to OTMA. There are five OTMA security levels, but only 1 OTMA security level can be in effect at any point in time. The OTMA security levels are NONE, PROFILE, CHECK, FULL, and REFRESH. They are described below:

NONE A system-wide security level. RACF is not called for messages received through OTMA. Specifically:
- RACF is not called when IMS receives the connection request (client-bid) from MQSeries or IMS Connect.
- RACF is not called to verify that the user ID in the incoming message is a valid user ID (one that has been defined to RACF).
- RACF is not called to verify that the user ID in the incoming message is authorized to the IMS command or IMS transaction requested in the message.
- The user ID caching scheme is not used.

PROFILE A message-by-message security level. In other words, each

incoming message entered through OTMA is checked to determine whether or not RACF will be called. Specifically:

- Messages entered from IMS Connect will contain a 1-byte security flag field. The value in this field determines whether or not RACF is called.
- Messages entered from the MQSeries-IMS Bridge application will contain a SecurityScope field in the MQIIH structure. The value in this field will determine whether or not RACF is called.

When you set the security level for OTMA to PROFILE, IMS checks each incoming message independently to see if the security value is set to NONE, CHECK, or FULL. Consider using the PROFILE security level for situations when application developers set the RACF security level as N (NONE), C (CHECK), or F (FULL) in each incoming message. In this case, the security level set in each message determines whether IMS calls RACF for security checking related to that message. You might not want application programmers deciding on the security for IMS commands and IMS transactions. RACF is called when IMS receives the connection request (client-bid) from MQSeries or IMS Connect.

CHECK A system-wide security level, which means that RACF will be called for messages received through OTMA. Specifically:

- RACF is called for client-bid connection requests. A cache, or hash table, is built for each OTMA client if the client-bid is successful. (Use of the hash table is described below).
- RACF is called to VERIFY that the user ID in the incoming message is a valid user ID (one that has been defined to RACF).

  If the OTMA client (IMS Connect or MQSeries for z/OS) supplied a UTOKEN in the incoming message, IMS supplies the address of the UTOKEN on the VERIFY call to RACF. Use of the UTOKEN in VERIFY processing improves performance. RACF returns an ACEE security control block to IMS for verified user IDs.
- A user ID caching scheme is used in IMS/OTMA environments. The caching scheme also improves authorization checking performance.

  A cache, or hash table, is used to store previously verified user IDs. Each OTMA client (IMS Connect, MQSeries for z/OS, etc.) has a hash table created in the IMS control region after a successful client bid. Use of the hash table minimizes the number of calls to RACF to VERIFY user IDs. This way, if the same user ID enters multiple messages destined for IMS/OTMA, IMS can check the hash table for a valid entry for the user ID and might be able to avoid the VERIFY call to RACF. The entry for the user ID in the hash table contains a pointer to the ACEE for the user ID. The ACEE that is pointed to can be used for resource (command and transaction) FASTAUTH calls to RACF.
- RACF is called to verify that the user ID in the incoming message is authorized to the IMS command or IMS transaction requested in the message. The address of the ACEE, previously built by RACF during verify authorization processing, is supplied by IMS on the FASTAUTH call to RACF.
- RACF is called to verify that the user ID in the incoming message is authorized to the IMS transaction code set as the

destination on a DL/I CHNG or AUTH call. However, an existing ACEE is not used for these calls; therefore, another call is made to RACF to dynamically build an ACEE for the CHNG or AUTH call. If you know the application will issue many CHNG or AUTH calls, consider using a different OTMA security level to overcome the performance impact.

FULL A system-wide security level, which means that RACF will be called for messages received through OTMA.

FULL has the same characteristics as CHECK, with two exceptions:

1. During the verify processing, RACF is called a second time to build an additional ACEE security control block in the dependent region.
2. If the application program that processes the OTMA-entered transaction issues one or more CHNG calls (or AUTH calls with the destination set to a different transaction code), the security environment already exists in the dependent region and does not have to be dynamically built. Alternately, if the application does not issue these calls, a security level of FULL is not needed, and you might consider an OTMA security level of CHECK or PROFILE.

Related Reading: For more information on IMS security, see *IMS Version 9: Administration Guide: System*.

For more information on the values `NONE`, `FULL`, or `CHECK`, see the `/SEC OTMA` command in *IMS Version 9: Command Reference*.

General OTMA Security Considerations

- If you use RACF (or an equivalent product) for security, define the `IMSXCF.group.client_member_name` in the FACILITY class.

  If you define the `IMSXCF.group.client_member_name` in the FACILITY class, and if IMS security is not set to `NONE`, the user token for the client-bid request must be valid and the user must have READ access to the FACILITY class.

  If the user token for a client-bid request fails RACF verification, the client receives a NAK message from the server.
- Authorize the XCF client for z/OS.
- If you define your OTMA applications with full security, the security environment is kept until the application ends.
- After IMS receives messages received from OTMA, when OTMA security is activated, IMS calls RACF to verify that the user ID in the incoming message is a valid RACF user ID. IMS is not passed a password for the user ID, so the call to RACF is to verify the user ID only. If the user's password has not been validated before IMS receives the message, the password cannot be validated.
- IMS uses the UTOKEN in the input message in the call to RACF not only to verify the user ID, but also to create a security control block in the IMS control region to represent each verified user ID. The security control blocks built in the IMS control region, representing verified RACF user IDs, are called accessor control elements or ACEEs.

Using DL/I Calls in an OTMA Environment

This section describes the DL/I calls that are used in an OTMA environment:

- CHNG call

  If a CHNG call is issued from an OTMA submitted transaction, the destination is assumed to be the same OTMA client (the TPIPE name is set by the CHNG call). This behavior can be altered by the OTMA Prerouting and Destination Resolution exit routines.

  An IMS application program that issues a CHNG call to an alternate PCB (specifying an options list) does not cause IMS to call the OTMA Prerouting and Destination Resolution exit routines to determine the destination. However, an IMS application program that issues a CHNG call to an alternate PCB (specifying an APPC descriptor) does cause IMS to call the OTMA exit routines to determine the destination.

  The application program can still issue ISRT calls to the I/O PCB to send data to an OTMA destination.

  OTMA application programs can use CHNG and ISRT calls for APPC destinations.

- INQY (null) call

  An INQY call issued for an OTMA destination returns the following information: the transaction pipe name, the client XCF member name, the user ID, the group name, and the synchronization levels.

- PURG call

  An IMS application program that issues a PURG call causes IMS to call the OTMA Prerouting and the Destination Resolution exit routines to determine the destination.

- SETO call

  An IMS application program that issues a SETO call does not cause IMS to call the OTMA Prerouting and the Destination Resolution exit routines to determine the destination.

  Existing IMS application programs that issue SETO calls might not run as expected because a return code is returned to the program if it is processing an OTMA-originated transaction. APPC/IMS application programs that issue SETO calls might need modification if they require implicit OTMA support.

  One way to make these application programs work is to use an INQY call before issuing the SETO call. The application program can use the output from the INQY call to determine if a transaction originated from an OTMA client, and not issue the SETO call.

For those DL/I calls that cause IMS to call one of the OTMA exit routines, IMS only calls the exit routines if the destination has not yet been set (for example, by another DL/I call).

Related Reading: For more information on these calls, see *IMS Version 9: Application Programming: Transaction Manager*.

To initiate protected conversations (such as accessing multiple resource managers' resources under one unit of recovery in an RRS/MVS environment), the client-adapter code (OTMA user) must acquire and own a private context and provide the context ID in the state-data section of the message prefix.

Definition: A *context* is a z/OS entity under which resource managers perform work; a private context is required in this environment.

During message traffic between IMS and the client, if the context-ID field in the message header is non-zero, protected conversation processing occurs.

OTMA Program-to-Program Switch Processing

This section describes how OTMA program-to-program (P2P) message switches occur. Two types of message switch occur in OTMA: commit-then-send, and send-then-commit. Each type is described briefly below. This section focuses primarily on the send-then-commit message switch and provides usage scenarios for different send-then-commit message switches.

For OTMA commit-then-send input messages (also called asynchronous or CM0 messages), the program switch always results in another CM0 message. For OTMA send-then-commit input messages (also called synchronous or CM1 messages), the program switch results vary, depending on whether:

- there is an ISRT call to the I/O PCB
- an express PCB is used for the switch
- there is a switch to multiple programs
- the IMS start-up parameter OTMAASY=Y is specified
- the transaction is protected

A P2P switch for a CM1 input message, therefore, could be another CM1 message, a `DFS2082` message, or a CM0 message. In addition, some OTMA clients, for example MQSeries, can accept a CM0 output message for a CM1 input message; others, however, may not.

The usage scenarios that follow apply only to send-then-commit messages.

OTMA Single-Stream Program Switch

This basic switch is shown in Figure 32. Program A switches to program B, and Program B switches to Program C, which then inserts back to the I/O PCB. This model of program flow delivers the send-then-commit output message successfully. Single-stream means that the program switches occur one after another. No express PCBs are used in the P2P message switches.

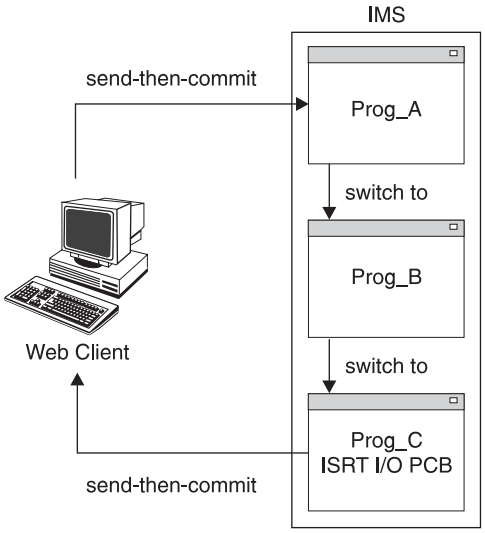

Figure 32. Single-Stream Program Switch

OTMA Program Switch without ISRT to I/O PCB

When several switches occur in sequence and none inserts back to the I/O PCB, message DFS2082 is sent back to the OTMA client. The last program switched to, Program C, does not insert back to the I/O PCB. IMS therefore generates message DFS2082 for the OTMA client. If program C runs in a remote IMS through MSC and does not insert back to the I/O PCB, the remote IMS does not issue DFS2082. However, in this case, the OTMA client program might hang.

OTMA Program Switch with Express PCB

A P2P message switch with an express PCB can lead to a commit-then-send output message. Program A uses an express PCB rather than a non-express PCB to perform the P2P message switch. The output from Program B is commit-then-send because using the express PCB forces Program B to be processed asynchronously. When a program is processed asynchronously and inserts back to the I/O PCB, the output message is sent as a commit-then-send message. However, if Program A also switches to Program C using a non-express PCB, Program C then inserts back to the I/O PCB. The output from C will be a send-then-commit message.

OTMA Program Switch to Multiple Programs

After a program inserts back to the I/O PCB, the rest of the P2P message switch, if any, is processed asynchronously. Program A switches to Program B, which inserts back to the I/O PCB. The output from Program B will be a send-then-commit message. Program B then switches to program C, which will be processed asynchronously.

A “race” condition can occur when a program switches to multiple programs. Program A switches to multiple programs using non-express PCBs. Only one switched-to program, the one scheduled first, is processed synchronously. The rest of the switched-to programs are processed asynchronously. If the program processed synchronously inserts back to the I/O PCB, the output message is a send-then-commit message.

In some cases, one of the multiple programs could be a remote program. This program flow is shown in Figure 33 on page 57. Program A switches to remote Program B through MSC. Program B first launches a new program, Program C, in the local IMS and then inserts a response to the OTMA client through the local IMS. Depending on what happens first (scheduling of Program C or the processing of the response for the OTMA client) in the local IMS, an unwanted DFS2082 message could be sent to the client. This is also a race condition. If Program C gets processed first in the local IMS, a DFS2082 message is sent. If the response is processed first, the expected output from Program B is delivered synchronously using send-then-commit.

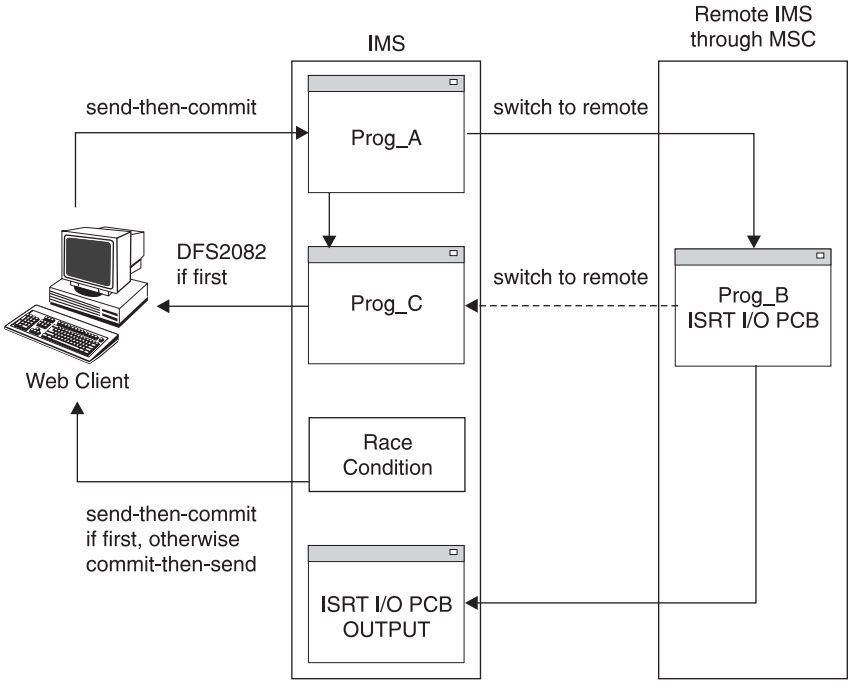

Figure 33. Race condition resulting from program switch to multiple programs

To avoid creating the race condition in these circumstances, you can do any one of the following:

- Modify your programs to avoid multiple program-to-program switches within the same transaction for send-then-commit (CM1).
- Use commit-then-send (CM0) input when performing multiple program-to-program switches within the same transaction.
- Use the IMS start-up parameter OTMAASY to serialize P2P message switch processing, described in “OTMA Program Switch with OTMAASY Option.”

OTMA Program Switch with OTMAASY Option

The IMS start-up parameter OTMAASY can be used to serialize P2P message switch processing to prevent a potential race condition. To avoid the race condition OTMAASY is used to create a program switch model similar to the single-stream model. Program B, for which the response mode transaction is processed synchronously, can deliver the send-then-commit (synchronous) output message. Program C, which is running with a non-response mode transaction, processes the message asynchronously. With this method, the race condition described above can be avoided.

OTMA Program Switch for Protected Transactions

For a non-conversational program that performs a P2P message switch for a protected transaction, ABENDU0711, with reason code 1D, will be returned. For a conversational program, the program receives an X6 status code.

Other OTMA Program Switch Considerations

The following considerations also apply to P2P switching:

- The P2P message switch is not supported for OTMA protected messages (send-then-commit input with synclevel = SyncPt).

- If a non-conversational program performs a P2P message switch to a program in a Shared Queues environment, the program in the SQ environment must be running on the same IMS where the first program gets scheduled.

IMS Commands Using OTMA

OTMA clients can enter IMS commands.

Restrictions: OTMA clients cannot enter IMS commands from the following:

- An IMS Remote Site Recovery (RSR) tracking subsystem
- An IMS Extended Recovery Facility (XRF) alternate subsystem
- A CICS®-IMS DBCTL subsystem

Only certain commands are valid from OTMA.

Related Reading: For more information on supported IMS commands, see *IMS Version 9: Command Reference*.

OTMA Terminology

The following terms apply to this section :

Tpipe Is analogous to an LTERM. It is a logical structure that represents an anchor point for client transactions and output. The Tpipe name is unique within a client structure.

TMEMBER Is the name of a client that connects to an OTMA group. The group is created by the first member to join it, usually IMS. All members are clients, except IMS, which is the server. Clients communicate with IMS using the XCF interface by sending transactions to IMS. IMS then returns the output to those clients.

OTMA Represents the logical collection of all XCF members associated with a given group name.

The initial letter “T” in Tpipe and Tmember represents “Transaction.”

Modified Commands for OTMA

The following commands have been modified for OTMA.

- `/DISPLAY ACTIVE`

  This command displays OTMA group status. Only one OTMA group can be active at a time.

- `/DISPLAY OTMA`

  This command displays the following information for OTMA clients and servers:

  – Each member in the XCF group
  – The XCF status for each member
  – The user status for each member
  – The security status for each server

  If the IMS application program uses the alternate PCB, the exit routine determines the destination, and the member name is displayed, even when it is not in the same group. The status can be either active or not defined.

  After the client-bid request is received from a Tmember, IMS creates a Tmember control block for that Tmember. If the command is issued prior to the client-bid request, the Tmember is not displayed.

- /DISPLAY SHUTDOWN STATUS

  This command displays shutdown status of OTMA processing. The following OTMA information is displayed:

  – The current phase of OTMA shutdown processing
  – Whether the transaction is commit-then-send or send-then-commit
  – Which clients and transaction pipes are in progress and thus preventing shutdown from completing

- /DISPLAY STATUS TMEMBER

  This command displays all transaction pipes that are stopped.

- /DISPLAY TMEMBER ALL

  This command displays the following information for OTMA clients and servers:

  – Each member in the XCF group
  – The XCF status for each member
  – The user status for each member
  – The security status for each server

- /DISPLAY TMEMBER QCNT

  This command displays the number of asynchronous output messages on the global queue for the specified OTMA member. The QCNT keyword is valid only in a shared-queues environment.

- /DISPLAY TMEMBER *tmembername* TPIPE *tpipename* | ALL

  This command displays TPIPE status for each tmember, including:

  – Enqueue and dequeue counts
  – Current queue count
  – Current status

- /DISPLAY TMEMBER *tmembername* QCNT TPIPE *tpipename*

  This command displays the number of asynchronous output messages on the global queue for the specified OTMA member and transaction pipe. The QCNT keyword is valid only in a shared-queues environment.

- /DISPLAY TRACE TMEMBER

  This command displays all the transaction pipes that are currently being traced for the specified client.

- /DISPLAY TRANSACTION

  This command, when issued by an OTMA client, sends its output directly to the client, not to the IMS master terminal. Depending on the setting of the extended-response-requested flag in the message-control information section of the message prefix, the output is either in an architected format (only supported for this command) or in the standard IMS format.

  Related Reading:

  – For more information on the message prefix, see Chapter 5, “OTMA Message Prefix,” on page 69.
  – For information on the transaction attributes provided in the application-data area, see Chapter 6, “OTMA Architected Transaction Attributes,” on page 95.

- /DEQUEUE TMEMBER TPIPE

  This command dequeues messages from the transaction-pipe structure associated with the transaction pipe.

- /SECURE OTMA

This command is used to control the RACF security level for OTMA client-originated transactions. You can specify that RACF be used or bypassed. If RACF is bypassed and NONE is specified, you can issue only the following default unsecured commands:
– BRO
– LOC
– LOG
– RDI
– UNL

You can also use the PROFILE keyword to specify or bypass RACF security for individual transactions. This can be important if you do not want the performance overhead of processing security for your entire transaction workload. You specify the security for each transaction by setting the values in the security-data section of the OTMA message prefix.

The /DISPLAY OTMA command shows the security option currently in effect.

- /START OTMA

  This command causes IMS to join the XCF group (and create the group if necessary). All clients must know the group name, which is specified on the GRNAME= parameter in the IMS PROCLIB member DFSPB*xxx*. The XCF group is created and joined according to the following rules:
  – In the IMS procedure, if OTMA=N and GRNAME= is set to blanks, the XCF group can be neither created nor joined.
  – If OTMA=N and GRNAME= is set to a valid value, the XCF group is created and joined when you issue the /START OTMA command.
  – If OTMA=Y, the XCF group is both created and joined as part of IMS initialization. In this case, the /START OTMA command is used to rejoin a group after a /STOP OTMA command has caused IMS to leave the group.

  It is assumed that all clients know the names of the servers with which they are to communicate.

  /START OTMA initiates the following processing:

  1. IMS joins the XCF group.
  2. After a successful client-bid request, IMS sends an ACK message to the client.
  3. IMS begins sending all commit-then-send output to the client.

- /STOP OTMA

  This command causes IMS to leave the XCF group (but it does not stop the group itself). All clients must know the group name, which is specified on the GRNAME= parameter in the IMS procedure.

- /START TMEMBER *tmembername* TPIPE *tpipename* | ALL

  This command causes IMS to send one of the following OTMA commands to the client:
  – “resume input for *tpipename*”
  – “resume input for all *tpipename*s”

  IMS then resumes sending output to the client.

- /STOP TMEMBER *tmembername* TPIPE *tpipename* | ALL

  This command causes IMS to send one of the following OTMA commands to the client:.
  – “suspend input for *tpipename*”
  – “suspend input for all *tpipename*s”

IMS then does not send any more output to the client.

- `/TRACE SET ON|OFF TABLE OTMT`

  This command starts or stops online tracing to the OTMA trace table (OTMT).

- `/TRACE SET ON|OFF TMEMBER TPIPE`

  This command starts or stops online tracing of OTMA client activity and transaction-pipe activity for clients.

  A temporary Tpipe is created when you issue a `/TRACE Tpipe` or `/STOP Tpipe` command against a Tpipe that does not exist. A temporary Tpipe is converted to a permanent Tpipe if an input message reaches IMS through the Tpipe or if an output message is queued to the output queue of the Tpipe.

  When you issue a `/DISPLAY Tpipe` command against a temporary Tpipe, the status of `TMP` is displayed.

Generally, an OTMA client should use the send-then-commit flow to process IMS commands. However, to process any of the following commands, the client must use the commit-then-send flow:

- `/DBDUMP DATABASE`
- `/DBRECOVERY AREA|DATABASE`
- `/START AREA|DATABASE`
- `/START REGION`
- `/STOP AREA|DATABASE`
- `/STOP REGION`

Recommendation: Because the client must use the commit-then-send flow, the output from these commands cannot be tied to the input command. The OTMA prefixes are not replicated (the only field common to both the input and the output is the transaction-pipe name). Therefore, it is recommended that the client submit IMS commands using a transaction pipe that the client reserves for IMS command processing.

IMS Messages Introduced by OTMA

The IMS messages (DFSnnnn) that are introduced by OTMA are listed in Table 10.

Related Reading: For more information on these messages, see *IMS Version 9: Messages and Codes, Volume 2*.

Table 10. IMS Messages introduced by OTMA

DFS1268	**DFS1269E**	**DFS1280E**	**DFS1281E**	**DFS1282E**
DFS1283E	**DFS1284E**	**DFS1285E**	**DFS1286E**	**DFS1287E**
DFS1288E	**DFS1289E**	**DFS1290E**	**DFS1291E**	**DFS1292E**
DFS1293E	**DFS1294E**	**DFS1295E**	**DFS1296E**	**DFS1297E**
DFS2224	**DFS2360I**	**DFS2361I**	**DFS2362I**	**DFS2363I**
DFS2364I	**DFS2365I**	**DFS2368I**	**DFS2369I**	**DFS2370I**
DFS2371I	**DFS2372I**	**DFS2373I**	**DFS2374W**	**DFS2375W**
DFS2376W	**DFS2384E**	**DFS2385E**	**DFS2389I**	**DFS2390I**
DFS2391I	**DFS2392I**	**DFS2393I**	**DFS2394I**	**DFS2395I**
DFS2396I				

In addition, messages DFS3657 and DFS3659X are modified for OTMA to include the 16-byte descriptor name.

Chapter 4. OTMA Diagnostic Information

This chapter provides diagnostic information for OTMA.

In this chapter :

OTMA Sense Codes for NAK Messages

This section lists the sense codes that accompany OTMA negative acknowledgement (NAK) messages. Codes X'0000' through X'0FFF' and X'9000' through X'FFFF' are reserved for IBM use. Codes X'1000' through X'8FFF' are reserved for customer use.

0001

Explanation: IMS received a message from a client, but the message is not part of the OTMA sign-on protocol. The client is not yet ready for message processing.

Programmer Response: Make sure that the client has sent the client-bid request and successfully received an ACK for that client-bid request.

0002

Explanation: IMS received a message from a client, but the client cannot send or receive messages.

Programmer Response: Make sure that the client is not stopped.

0003

Explanation: Either the client-bid request did not correctly set the length of the state-data section, or the application-data section is present.

Programmer Response: Make sure that the length of the state-data section and the XCF length fields are correct. The application-data section should not be included in the client-bid request.

0004

Explanation: Reserved

0005

Explanation: IMS received a message for a multi-segment message that duplicated an existing segment.

Programmer Response: Make sure that the segment number is not duplicated.

0006

Explanation: IMS received a return code from XCF after attempting to receive the message. The return code is saved in the reason code field of the message-control information section of the message prefix.

Programmer Response: Refer to the appropriate z/OS programming manual for action.

0007

Explanation: The maximum number (255) of clients was reached. No new client structure is created.

Programmer Response: Make sure the client name is correct and that you have not exceeded the limit of 255 clients.

0008

Explanation: The security check by IMS rejected the client-bid request.

Programmer Response: Make sure the security data specified in the client-bid request is valid and correct. In addition, verify RACF definitions and the IMS security exit routine.

0009

Explanation: IMS received an invalid OTMA command.

Programmer Response: Verify the OTMA command. For a list of allowable commands, see "OTMA Message-Control Information" on page 69.

000A

Explanation: IMS received an OTMA data message. IMS only accepts transaction, command, response, or commit messages. A data message must be a

continuation of an IMS conversational transaction.

Programmer Response: Verify the data message being sent. The data flag in byte 1 of the OTMA control data could have been set incorrectly, or the conversation iteration message was sent to a terminated IMS conversational transaction.

000B

Explanation: IMS received an invalid message type. An OTMA message must be a transaction, command, response, data, or commit message.

Programmer Response: Make sure the Message Type in the message-control information section of the message prefix is set properly.

000C

Explanation: Reserved

000D

Explanation: IMS received an OTMA data message for continuation of an IMS conversation using a nonexistent transaction pipe.

Programmer Response: Make sure the transaction pipe exists and that the first iteration of the IMS conversation is successfully completed.

000E

Explanation: IMS was unable to create the transaction pipe to process the message.

The synchronized tpipe flag in the processing flag of the OTMA message-control-information prefix was set incorrectly on or off an existing tpipe. After a tpipe is created for an input or output OTMA message, the synchronized tpipe setting for the tpipe cannot be changed for the subsequent input or output OTMA message.

Programmer Response: Make sure IMS storage pools are properly allocated and available.

000F

Explanation: The transaction pipe for the message was stopped.

Programmer Response: Find out why the transaction pipe was stopped. Issue an IMS `/START` command to restart the transaction pipe.

0010

Explanation: The OTMA message had no state data.

Programmer Response: Verify that the state-data flag is set and that state data is present.

0011

Explanation: IMS received the OTMA commit message, but the request was not to terminate the conversation. IMS only allows commit-type messages for terminating IMS conversations (equivalent to the IMS `/EXIT` command).

Programmer Response: Make sure the commit message is used to terminate an IMS conversation.

0012

Explanation: The OTMA message prefix was too large. The maximum size of the OTMA prefix is 4 KB.

Programmer Response: Check the size of the OTMA message prefix.

0013

Explanation: The client-bid request did not set the size of the message hash table. This size is a required field, which is to be set by the client for client-bid requests.

Programmer Response: Set the message hash table size in the client-bid request.

0014

Explanation: The client sent a second client-bid request while the first client-bid request was still active.

Programmer Response: Make sure the client leaves and joins the XCF group before sending a new client-bid request to IMS.

0015

Explanation: IMS was unable to allocate storage for the message hash table.

Programmer Response: Check the specified size of the message hash table; it might be too large.

0016

Explanation: Client was not yet active and ready for message processing.

Programmer Response: Make sure the client-bid request was sent and successfully completed. Also check that the XCF state is active.

0017

Explanation: The OTMA message specified an invalid synchronization level in the state-data section of the message prefix. The synchronization-level field should be either None, Commit, or SYNCPT.

Programmer Response: Make sure the synchronization level is valid.

0018

Explanation: The OTMA message had an invalid transaction-pipe name. See “OTMA Naming Conventions” on page 14 for the naming rules for transaction pipes.

Programmer Response: Correct the transaction-pipe name.

0019

Explanation: The OTMA message had an invalid client name. See “OTMA Naming Conventions” on page 14 for the naming rules for clients.

Programmer Response: Correct the client name.

001A

Explanation: IMS detected an error and canceled the message before putting it on the IMS queue for processing. Usually, an IMS message (accompanying the NAK) describes the problem.

Programmer Response: See the reason code that accompanies the NAK code.

Reason Codes:

Hex:	Decimal:	Explanation:
X'15'	21	The message segment length or ZZ field cannot be changed by DFSNPRT0 exit.
X'16'	22	Invalid security option specified in the message prefix.
X'17'	23	Invalid command from an OTMA client. See DFS1285E.
X'18'	24	Transaction currently not available for use. See DFS3470E.
X'19'	25	SMB transaction/LTERM is stopped. See DFS065.
X'1A'	26	Invalid CPIC transaction. See DFS1286E.
X'1B'	27	Invalid remote destination (RCNT). See DFS1287E.
X'1C'	28	Invalid CNT name specified. See DFS1288E.
X'1D'	29	SMB not found. See DFS064.
X'1E'	30	Invalid security. See DFS1292E.
X'1F'	31	System error requested.
X'20'	32	System error message.
X'21'	33	User error message.
X'22'	34	Single-segment message. See DFS1290E.
X'23'	35	All messages discarded. See DFS249.
X'24'	36	Null segment sent. See DFS249. One cause for this error maybe the length specified on the MSGLEN key parameter of the IXCMSGO macro does not match the length of the OTMA data. No extra or null data can be padded at the end of the application data section.
X'25'	37	Queue overflow as unsuccessful insert.
X'26'	38	Commit mode 0 not allowed for IMS conversational or Fast Path transaction. See DFS1291E.
X'27'	39	IMS conversation is stopped, similar to an `/EXIT` command.
X'28'	40	DFSNPRT0 requested that a message be rerouted to a remote system, but failed. See DFS064.
X'29'	41	DFSNPRT0 requested that a message be rerouted to a remote system, but failed. See DFS070.
X'32'	50	The length specified in the application data section or the segment length for multi-segment input data exceeds the length specified on the MSGLEN key parameter of the XCF IXCMSGO macro.

001B

Explanation: IMS rejected the message, because IMS is shut down.

Programmer Response: Resend the message when IMS is restarted.

001C

Explanation: IMS rejected the message, because the synchronization flag was not set in the state-data section of the message prefix.

Programmer Response: Set the synchronization flag before sending the message to IMS.

001D

Explanation: IMS rejected the message, because the length of the user-data section of the message prefix exceeded the maximum allowable length (1024 bytes).

Programmer Response: Check the length of the user-data section of the message prefix.

001E

Explanation: IMS rejected the message, because the length of the server user data in the state-data section of the message prefix exceeded the maximum allowable length (256 bytes).

Programmer Response: Check the length of the server user data.

001F

Explanation: IMS rejected the message, because the recoverable sequence number in the message-control information section of the message prefix does not match the IMS sequence number for the synchronized transaction pipe.

If IMS receives 001F from a client, IMS stops the Tpipe.

Programmer Response: Check the recoverable sequence number in the message and ensure it is at least one greater than the current recoverable sequence number for the synchronized transaction pipe.

0020

Explanation: IMS rejected the message, because the message did not have any application data.

Programmer Response: Check the application-data section of the message prefix and ensure it has a transaction code or a valid IMS command.

0021

Explanation: IMS rejected the message, because the chain flag was not set in the message-control information section of the message prefix.

Programmer Response: Ensure the chain flag is set properly.

0022

Explanation: IMS was unable to find the transaction pipe associated with the Server token to process the conversational message.

Programmer Response: Ensure that the Server token is passed correctly to the server with the conversational state bit on.

0023

Explanation: The input recoverable sequence number is invalid.

Programmer Response: If one of the following conditions exists, IMS sends this NAK message in response to the input message:

- The Tpipe is synchronized, the transaction is defined as recoverable, and the input recoverable sequence number is 0.
- The Tpipe is synchronized, the transaction is defined as irrecoverable, and the input recoverable sequence number is not 0.
- The Tpipe is not synchronized, and the input recoverable sequence number is not 0.
- The Tpipe is synchronized for command input, the commit mode is 0, and the input recoverable sequence number is not 0.

0024

Explanation: A conversational program has not yet responded to the last input message. Multiple conversational messages were received by OTMA; subsequent messages were rejected because the first message is still being processed. IMS discards subsequent messages, and responds to the accepted input message.

Programmer Response: Modify the client program to wait for the conversational transaction output before sending the next input message.

0025

Explanation: IMS has detected a resynchronization protocol violation during the resynchronization process.

Programmer Response: If one of the following conditions exists, IMS sends this NAK message in response to the input message:

- The client's resynchronization logic does not follow the OTMA resynchronization protocol.
- The client's message prefix specifies the incorrect Tpipe name or member name.

0026

Explanation: During resynchronization, IMS tried to dequeue messages without success.

Programmer Response: Contact your IBM Support Center for assistance.

0027

Explanation: During resynchronization, IMS tried to reset recoverable sequence numbers without success.

Programmer Response: Contact your IBM Support Center for assistance.

002C

Explanation: The OTMA message specified an invalid commit mode with a synchronized Tpipe in the state-data section of the message prefix. You must use the Commit-then-Send commit mode.

Programmer Response: Make sure that you choose the Commit-then-Send commit mode.

002D

Explanation: An incompatibility between the synchronization level and the commit level was indicated in the message prefix. A send-then-commit level is required with a sync level of `SYNCPT`.

Programmer Response: Correct the incompatibility in the message prefix.

002E

Explanation: The OTMA message prefix indicated an incompatibility between the sync level field and the context ID field. the `context_id` field must only be supplied with a sync level of `SYNCPT`.

Programmer Response: Correct the incompatibility in the message prefix.

002F

Explanation: IMS was unable to express unprotected interest in the context contained in the context_id field of the OTMA message prefix.

Programmer Response: Contact your IBM Support Center for assistance.

OTMA Return Codes

If an error occurs in an IMS OTMA exit routine, the IMS application program can retrieve a status code for the IMS call. The X'67D0' records are written to the IMS log, and the return codes include the following:

Code: Explanation:

X'1C' An internal interface error occurred.

X'20' DFSYDRU0 overrides exceed the maximum limit.

X'24' DFSYDRU0 specifies an invalid destination.

X'28' DFSYDRU0 specifies an invalid return code.

X'2C' DFSYPRX0 returned an invalid XCF member name.

X'30' DFSYPRX0 required an XCF member name that was not returned.

X'34' DFSYPRX0 returned an invalid return code.

X'38' The destination is in a different client, and the XCF member name from DFSYDRU0 is invalid.

X'3C' DFSYDRU0 returned an invalid user data length.

Chapter 5. OTMA Message Prefix

This chapter describes the syntax of OTMA messages. OTMA messages are mapped by the DFSYMSG DSECT in IMS.ADFSMAC. The maximum length for a message prefix is 4096 bytes; this length does not include the application data.

In this chapter

Table 11 shows the segments of the OTMA message prefix and lists the locations of some of the key fields within those prefix segments. In this table, the term "Server" refers to IMS.

Table 11. OTMA Message Prefix segments and their Key Fields

Message Control Information	State Data	Security Data	User Data	Application Data
Tpipe name Message type Sequence numbers Processing flag Response Indicator Chaining Indicator	Destination override Map name Sync Flags Sync_level Commit Mode Tokens Server State	User ID Utoken Security Flags		

OTMA Message-Control Information

For every message, you must provide message-control information on the MSGCNTL parameter of the XCF IXCMSGO macro.

Format of OTMA Message-Control Information

Table 12 is a summary of the content of the message-control information section of the message prefix (column 1 from Table 11). The summary includes byte, length, content, hexadecimal value, the meaning, and includes usage comments. More information about the content follows the table in "Explanation of OTMA Message-Control Information Fields" on page 73.

Table 12. Message-Control Information Summary

Byte	Length	Content	Value	Meaning	Comments
0	1	**Architecture Level**	X'01'	OTMA release level.	Mandatory for all messages.
1	1	**Message Type**			Mandatory for all messages.
		Data	X'80'	Server output data (output from an IMS application program).	This data is not a transaction.

Table 12. Message-Control Information Summary (continued)

Byte	Length	Content	Value	Meaning	Comments
		Transaction	X'40'	Transaction or IMS command input to the server.	The actual transaction name is specified in the application-data section of the prefix.
		Response	X'20'	A response message.	
		Command	X'10'	An OTMA command (not an IMS command).	
		Commit Confirmation	X'08'	Commit complete.	Used by the server to notify the client of sync point completion. Only used for send-then-commit transactions. See "commit-confirmation flag" below.
2	1	**Response flag**			Response flags are mutually exclusive.
		ACK	X'80'	Positive acknowledgement.	
		NAK	X'40'	Negative acknowledgement.	A NAK can be accompanied by an additional sense code.
		Response Requested	X'20'	A response is requested for this message.	
		Extended Response Requested	X'10'	Requests architected transaction or command attributes to be returned to the client.	
3	1	**Commit-confirmation flag**			
		Committed	X'80'	Server committed successfully.	
		Aborted	X'40'	Server aborted commit.	
4	1	**Command Type**			
		Client-Bid	X'04'	Sent by a client to the server.	The response-requested flag and the appropriate state data fields (for example, `Member Name`) must also be set.
		Server Available	X'08'	Sent by a the server to a client.	The appropriate state data fields (for example, `Member Name`) must also be set.
		CBresynch	X'0C'	Sent by a client to the server to request a resynchronization.	This client-bid request with resynchronization to follow is optional, and causes the server to send an SRVresynch command to the client.
		Suspend Processing for All Tpipes	X'14'	The server sends this command to suspend all message activity with the client.	

Table 12. Message-Control Information Summary (continued)

Byte	Length	Content	Value	Meaning	Comments
		Resume Processing for All Tpipes	X'18'	The server sends this command to resume message processing with the client.	
		Suspend Input for Tpipe	X'1C'	The server sends this command when it is overloaded.	
		Resume Input for Tpipe	X'20'	The server sends this command when it is ready for client input (following a Suspend Input for Tpipe command).	
		Resume Output for Tpipe	X'24'	Sent by a client to the server to request queued tpipe output be resent.	
		SRVresynch	X'2C'	Sent by the server to a client who has sent a CBresynch.	This command identifies all synchronized Tpipes within the server.
		Resume Output for the Special Queue for Tpipe	X'28'	Sent by a client to the server to request messages from the special queue for Tpipe.	
		REQresynch	X'30'	Sent by the server to a client to specify the state of a synchronized Tpipe.	
		REPresynch	X'34'	Sent by a client to the server to indicate the type of resynchronization to be performed by the server.	
		TBresynch	X'38'	Sent by a client to the server to initiate resynchronization for a particular Tpipe.	
5	1	**Processing flag**			
		Message in Special Queue	X'08'	One or more messages in the special queue for Tpipe.	
		Synchronized Tpipe	X'40'	Input and output sequence numbers are maintained for the transaction pipe.	
		Asynchronous output	X'20'	The server is sending unsolicited queued data messages.	

Table 12. Message-Control Information Summary (continued)

Byte	Length	Content	Value	Meaning	Comments
		Error Message Follows	X'10'	An error data message follows.	Set by the server when sending a NAK.
6	8	**Tpipe Name**		OTMA identification and processing control token.	This name is used to override the LTERM name on the I/O PCB for an IMS application program.
14	1	**Chain flag**			This flag is mandatory for multi-segment messages.
		First-In-Chain	X'80'	The first segment of a multi-segment message.	A message of only one segment is indicated by setting both the first-in-chain and last-in-chain flags.
		Middle-In-Chain	X'40'	Part of a multi-segment message.	
		Last-In-Chain	X'20'	The last segment of a multi-segment message.	
		Discard Chain	X'10'	Discard the current chain of message segments.	
15	1	**Prefix flag**			Indicates which sections of the message prefix are attached to this message.
		State Data	X'80'	The state-data section is included with the message.	State data section is mandatory for each message.
		Security	X'40'	The security section is included with the message.	
		User Data	X'20'	The user-data section is included with the message.	This data is specified by an OTMA client.
		Application Data	X'10'	The application-data section is included with the message.	
16	4	**Send-sequence number**		The sequence number for the transaction pipe.	Incremented on every send for each transaction pipe.
20	2	**Sense Code**		Accompanies a NAK message.	
	4	**Aging Value**		Aging value for the input user ID.	The aging value specifies how often the cached user ID ACEE should be refreshed. The aging value flag in byte 5 of the State Data must also be set. This value does not apply to an ACK or NAK message.
22	2	**Reason Code**		Accompanies a NAK message.	

Table 12. Message-Control Information Summary (continued)

Byte	Length	Content	Value	Meaning	Comments
24	4	**Recoverable Sequence Number**		The recoverable sequence number for the transaction pipe.	Incremented on every send of a recoverable message using a synchronized transaction pipe. Required for resynchronization only.
28	2	**Segment Sequence Number**		Sequence number for segments of a multi-segment OTMA message.	
30	2	**Reserved**			

Explanation of OTMA Message-Control Information Fields

This section provides explanations for the fields in the message-control information section of the message prefix.

Architecture level

Specifies the OTMA architecture level. The client specifies an architecture level, and the server indicates in the response message which architecture level it is using. The architecture levels used by a client and a server must match.

With IMS Version 6, the only valid value is X'01'.

Mandatory for all messages.

Message type Specifies the message type. Every OTMA message must specify a value for the message type. The values are not mutually exclusive. For example, when the server sends an ACK message to a client-submitted transaction, both the transaction and response flags are set.

Data Specifies server output data sent to the client. If the client specifies synchronization level `Confirm` in the state-data section of the message prefix, the server also sets `Response Requested` for the response flag. If the client does not specify a synchronization level, the server uses the default, `Confirm`.

Transaction Specifies client input data to the server.

Whether the server replies with an ACK or NAK message depends only on whether `Response Requested` is also set for the response flag.

Response Specifies the message type as response message, and is set when the message response flag specifies `Response Requested`.

If this flag is set, the response flag specifies either ACK or NAK.

The send-sequence numbers must match for the original data message and the response message. Chained transaction input messages to the server must always request a response before the next transaction (for a particular transaction pipe) is sent.

Command Specifies an OTMA protocol command. OTMA commands must always specify Response Requested for the Response flag.

Commit Confirmation
Specifies that commit is complete. This is sent by the server when a sync point has completed, and is only applicable for send-then-commit transactions. The commit-confirmation flag is also set.

This message is not affected by the synchronization-level flag in the state-data section of the message prefix.

Response Flag
Specifies either that the message is a response message or that a response is requested.

Acknowledgements to transactions include attributes (for that transaction) in the application-data section of the message prefix only if the transaction specifies Extended Response Requested.

ACK Specifies a positive acknowledgement.

NAK Specifies a negative acknowledgement.

See the sense code field for more information on the reason for the NAK.

Response Requested
Specifies that a response is requested for this message. This can be set for message types of Data, Transaction, or Command.

When sending send-then-commit IMS command output, IMS does not request an ACK regardless of the synchronization level.

Extended Response Requested
Specifies that an extended response is requested for this message. Can be set by a client only for transactions (or for transactions that specify an IMS command instead of a transaction code).

If this flag is set for a transaction, IMS returns the architected attributes for that transaction in the application-data section of the ACK message.

If this flag is set for a command, IMS returns the architected attributes in the application-data section of the ACK message. This flag can be set for the IMS commands `/DISPLAY TRANSACTION` and `/DISPLAY TRANSACTION ALL`.

Commit-Confirmation Flag
Specifies the success of a commit request. Sent by the server to the client in a commit-confirmation message. These messages are only applicable for send-then-commit transactions, and are not affected by the synchronization-level flag in the state-data section of the message prefix.

Committed Specifies that the server committed successfully.

Aborted Specifies that the server aborted the commit.

Command Type

Specifies the OTMA protocol command type.

IMS MTO commands are specified in the application-data section of the message.

Client-Bid Specifies the first message a client sends to the OTMA server. This command must also set the response-requested flag and the security flag in the message-control information section of the message prefix. The appropriate state-data fields (for example, Member Name) must also be set.

The security-data prefix must specify a Utoken field so the OTMA server can validate the client's authority to act as an OTMA client.

Because the server can respond to the client-bid request, this message should not be sent until the client is ready to start accepting data messages.

Server Available

Specifies the first message the server sends to a client. It is sent when the server has connected to the XCF group before the client has connected. The client replies to the Server Available message with a client-bid request. The appropriate state data fields (for example, `Member Name`) must also be set.

If the client connects first, it is notified by XCF when the server connects, and begins processing with a client-bid request.

CBresynch Specifies a client-bid message with a request by the client for resynchronization. This command is optional and causes the server to send an SRVresynch message to the client. The CBresynch command is the first message that a client sends to the OTMA server when it attempts to resynchronize with IMS and existing synchronized Tpipes exist for the client. Other than the CBresynch message indicator in the message prefix, the information required for the message prefix should be identical to the client-bid command.

If IMS receives a client-bid request from the client and IMS is aware of existing synchronized Tpipes, IMS issues informational message DFS2394I to the MTO. IMS resets the recoverable send- or receive-sequence numbers to 0 (zero) for all the synchronized Tpipes.

Related Reading: For more information on resynchronization, see "Client/Server Resynchronization with OTMA" on page 28.

Suspend Processing for All Tpipes

Specifies that the server is suspending all message activity with the client. All subsequent data input

receives a NAK message from the server. Similarly, the client should send a NAK message for any subsequent server messages.

If a client wishes to suspend processing for a particular transaction pipe, it must submit a /STOP TPIPE command as an OTMA message.

Resume Processing for All Tpipes
Specifies that the server is resuming message activity with the client.

If a client wishes to resume processing for a particular transaction pipe that has been stopped, it must submit a /START TPIPE command as an OTMA message.

Suspend Input for Tpipe
Specifies that the server is overloaded and is temporarily suspending input for the transaction pipe. All subsequent client input receives NAK messages for the transaction pipe specified in the message-control information section of the message prefix. A response is not requested for this command.

This command is also sent by IMS when the master terminal operator enters a /STOP TPIPE command.

Resume Input for Tpipe
Specifies that the server is ready to resume client input following an earlier Suspend Input for Tpipe command. A response is not requested for this command.

This command is also sent by IMS when the IMS master terminal operator issues a /START TPIPE command.

Resume Output for Tpipe
Specifies one or multiple tpipe names to the OTMA server. All queued output on the tpipes will be resent again.

Resume Output for the Special Queue for Tpipe
Specifies that a client is requesting to retrieve messages from the special queue for Tpipe. There are command options to retrieve messages. See “Format of OTMA State Data for Resume Output for the Special Queue for Tpipe” on page 85 for more information about the available command options.

SRVresynch Specifies the server’s response to a client’s CBresynch command. This command specifies the states of synchronized transaction pipes within the server (the send- and receive-sequence numbers).

This command is sent as a single message (with single or multiple segments), and an ACK is requested.

REQresynch Specifies the send-sequence number and the

receive sequence for a particular Tpipe. REQresynch is sent from IMS to a client.

REPresynch Specifies the client's desired state information for a Tpipe. A client sends the REPresynch command to IMS in response to the REQresynch command received from IMS.

TBresynch Specifies that the client is ready to receive the REQresynch command from IMS.

Processing Flag

Specifies options by which a client or server can control message processing.

Synchronized Tpipe

Specifies that the transaction pipe is to be synchronized. Allows the client to resynchronize a transaction pipe if there is a failure. Only valid for commit-then-send transactions.

This flag causes input and output sequence numbers to be maintained for the transaction pipe. All transactions routed through the transaction pipe must specify this flag consistently (either on or off).

Asynchronous Output

Specifies that the server is sending unsolicited queued output to the client. This can occur when IMS inserts a message to an alternate PCB.

Certain IMS commands, when submitted as commit-then-send, can cause IMS to send the output to a client with this flag set. In this case, the OTMA prefixes contain no identifying information that the client can use to correlate the output to the originating command message. These command output data messages simply identify the transaction-pipe name. IMS can also send some unsolicited error messages with only the transaction-pipe name.

Error Message Follows

Specifies that an error message follows this message. This flag is set for NAK messages from the server. An additional error message is then sent to the client.

The asynchronous-output flag is not set in the error data message, because the output is not generated by an IMS application.

Message in the Special Queue

Specifies that one or more messages exist in the special queue for the Tpipe to be delivered. This flag is always on for an IMS output message that has been sent from the special queue for Tpipe. Therefore, this flag can be used to determine whether there is any message in the special queue for an IMS output message that has been sent from the regular queue for Tpipe.

To determine whether the IMS output message is sent from the regular queue or from the special queue, check the “From Special Queue” flag in the Server State of the State Data.

To retrieve one or more messages from the special queue, issue the “Resume Output for the Special Queue for Tpipe” protocol command.

Tpipe Name Specifies the transaction-pipe name. For IMS, this name is used to override the LTERM name on the I/O PCB. This field is applicable for all transaction, data, and commit-confirmation message types. It is also applicable for certain response and command message types.

Related Reading: See also the Destination Override field in “Explanation of OTMA State Data Fields” on page 85.

Chain Flag Specifies how many segments are in the message. This flag is applicable to transaction and data message types, and it is mandatory for multi-segment messages.

First-In-Chain Specifies the first segment in a chain of segments which comprise a multi-segment message. Subsequent segments of the message only need the message-control information section of the message prefix. Other applicable prefix segments (for example, those specified by the client on the transaction message) are sent only with the first segment (with the first-in-chain flag set).

If the OTMA message has only one segment, the last-in-chain flag should also be set.

Middle-In-Chain
Specifies a segment that is neither first nor last in a chain of segments that comprise a multi-segment message. These segments only need the message-control information section of the message prefix.

Restriction: Because the client and server tokens are in the state-data section of the message prefix, they cannot be used to correlate and combine segmented messages. The transaction-pipe name and send-sequence numbers can be used for this purpose; they are in the message-control information section of the message prefix for each segment.

Last-In-Chain Specifies the last segment of a multi-segment message.

Discard Chain
Specifies that the entire chain of a multi-segment message is to be discarded. The last-in-chain flag must also be set.

Prefix Flag Specifies the sections of the message prefix that are attached to the OTMA message. Every message must have the

message-control information and state-data sections, but any combination of other sections can be sent with an OTMA message.

State data Specifies that the message includes the state-data section of the message prefix. See “OTMA State Data” on page 80.

Security data Specifies that the message includes the security-data section of the message prefix. See “OTMA Security Data” on page 89.

User data Specifies that the message includes the user-data section of the message prefix. See “OTMA User Data” on page 90.

Application data
Specifies that the message includes the application-data section of the message prefix. See “OTMA Application Data” on page 91.

Send-Sequence Number
Specifies the sequence number for a transaction pipe. This sequence number is updated by the client and server when sending messages or transactions.

Recommendation: Increment the number separately for each transaction pipe.

This number can also be used to match an ACK or NAK message with the specific message being acknowledged.

Sense Code Specifies the sense code that accompanies a NAK message. See Chapter 4, “OTMA Diagnostic Information,” on page 63.

Reason Code Specifies the reason code that accompanies a NAK message. This code can further qualify a sense code.

Userid Aging Value
Specifies the aging value in seconds for the input user ID. This field is different from the aging value specified in the OTMA client-bid command for OTMA connection. The aging value in the client-bid command sets the default aging value for all the OTMA user IDs; however, the Userid Aging Value overrides the default aging value for a specific user ID.

Recoverable Sequence Number
Specifies the recoverable sequence number for a transaction pipe. Incremented on every send of a recoverable message using a synchronized transaction pipe. Both the client and the server increment their recoverable send-sequence numbers and maintain them separately from the send-sequence number. Required for resynchronization only.

Related Reading: For more information on resynchronization, see “Client/Server Resynchronization with OTMA” on page 28.

Segment Sequence Number
Specifies the sequence number for a segment of a multi-segment message. This number must be updated for each segment, because messages are not necessarily delivered sequentially by XCF.

This number must have a value of 1 if the message has only one segment.

OTMA State Data

The state data is mandatory for any message. It immediately follows the message-control information section in the message prefix. It contains transaction-related information.

The state-data section has different formats for transaction-related information and for commands. State data for commands can be followed by a security-data section.

Format of OTMA State Data for Transaction-Related Information

Table 13 shows the format of state data (column 2 from Table 11 on page 69) for transaction-related information. The summary includes byte, length, content, hexadecimal value, the meaning, and includes usage comments. The state data fields are defined in “Explanation of OTMA State Data Fields” on page 85.

Table 13. State Data Format for Transaction-Related Information

Byte	Length	Content	Value	Meaning	Comments
0	2	**Length**		Length of the state-data section.	Includes the length field itself.
2	1	**Server State**			
		From Special Queue	X'20'	Output from the special queue for Tpipe.	Set by the server when sending output.
		Conversational State	X'80'	For a conversational transaction.	Set by the server and client when sending conversational data.
		Response Mode	X'40'	For a response-mode transaction.	Set by the server when sending output.
3	1	**Synchronization flag**			
		Commit-then-send	X'40'	A commit-then-send transaction.	The server commits output before sending it.
		Send-then-commit	X'20'	A send-then-commit transaction.	The server sends output before committing it.
4	1	**Synchronization Level**			The default is Confirm.
		None	X'00'	No synchronization.	Server application does not request an ACK message when sending output to a client.
		Confirm	X'01'	Synchronize.	Server sends transaction output with the Response Requested flag set.
		Syncpt	X'02'	This message is part of a protected conversation.	The resources updated under this conversation use the two-phase commit protocol.
5	1	**Client Flags**			

Table 13. State Data Format for Transaction-Related Information (continued)

Byte	Length	Content	Value	Meaning	Comments
		Send Only Message	X'80'	This is a send only message; the response is placed on the special queue.	This flag is valid only if the client requested a special queue during open.
		Reroute Request	X'20'	Reroute Requested.	Setting this flag reroutes CM0 output to the destination that is specified in the **Destination Override** field.
		Set Aging Value	X'40'	To accept the aging value.	This flag instructs IMS to accept the aging value specified at byte X'14' of the message control information.
6	8	**Map Name**		Data format map used by the server to map application input or output.	Optional.
14	16	**Server Token**		Server name.	Must be returned by the client to the server on responses.
30	16	**Correlator Token**		A client token to correlate input with output.	Optional.
46	16	**Context ID**		RRS Context ID	Used with SYNCLVL=02 and protected conversations.
62	8	**Destination Override**		Override the destination name for server output.	Optional.
70	2	**Server user data length**		Length of the server data	The length does not include the length field itself.
72	*	**Server user data**		Any data needed by the server.	Variable length. Optional.

Format of OTMA State Data for Server-Available and Client-Bid Commands

Table 14 summarizes the format for state data for command messages. The summary includes byte, length, content, hexadecimal value, the meaning, and includes usage comments.

Table 14. Server-Available and Client-Bid Command Format

Byte	Length	Content	Value	Meaning	Comments
0	2	**Length**		Length of the state-data section.	Includes the length field itself.
2	16	**Member Name**		XCF member name of originating server.	
18	8	**Originator's Token**		XCF member token of the originator of the message.	
26	8	**Destination Token**		XCF member token of the destination of the message.	

Table 14. Server-Available and Client-Bid Command Format (continued)

Byte	Length	Content	Value	Meaning	Comments
Note: The following fields are present only for client-bid commands.					
34	8	**DRU Exit Name**		Destination Resolution exit routine name.	
42	2	**MaxBlocksize**		Maximum block size for XCF transmissions from server to client.	Optional.
44	1	**Client-Bid Flag**			
		Special Queue	X'80'	Specifies that a special queue for a Tpipe is needed.	
45	1	**Reserved**			
46	4	**Aging value**		ACEE aging value in seconds.	The minimum value for caching support is 300 seconds (5 minutes).
50	4	**Hash Table Size**		Hash table size is used for processing multi-segment messages.	Suggested value is X'00000065',

Format of OTMA State Data for SRVresynch Command

The SRVresynch command is sent by IMS to pass all its known synchronized Tpipe names to the client. If the command data can not fit into a single buffer, chained multi-segment buffers will be sent instead. Table 15 summarizes the format of state data for the SRVresynch command. The summary includes byte, length, content, hexadecimal value, the meaning, and includes usage comments. The state data fields are defined in “Explanation of OTMA State Data Fields” on page 85.

Table 15. SRVresynch Command Format

Byte	Length	Content	Value	Meaning	Comments
0	2	**Length**		Length of the state-data section.	Includes the length field itself.
2	8	**Tpipe Name**		The transaction pipe name.	The Tpipe name can be repeated as necessary.

Format of OTMA State Data for REQresynch Command

The REQresynch command is used by IMS to pass the send-sequence number and the receive sequence for a specific Tpipe to the client. Table 16 summarizes the format of state data for the REQresynch command. The summary includes byte, length, content, hexadecimal value, the meaning, and includes usage comments. The state data fields are defined in “Explanation of OTMA State Data Fields” on page 85.

Table 16. REQresynch Command Format

Byte	Length	Content	Value	Meaning	Comments
0	2	**Length**		Length of the state-data section.	Includes the length field itself.

Table 16. REQresynch Command Format (continued)

Byte	Length	Content	Value	Meaning	Comments
2	8	Tpipe Name		The transaction pipe name.	
10	4	**Send-sequence number**		IMS recoverable send-sequence number for the transaction pipe.	
14	4	**Receive-sequence number**		IMS recoverable receive-sequence number for the transaction pipe.	
18	1	**Tpipe Flag 1**		Reserved for future use.	
19	1	**Tpipe Flag 2**		Reserved for future use.	
20	6	**RESERVED**			

Format of OTMA State Data for REPresynch Command

The REPresynch command is sent by the client in reply to the REQresynch request from IMS. It contains the desired state information for a Tpipe. Table 17 summarizes the format of state data for the REPresynch command. The summary includes byte, length, content, hexadecimal value, the meaning, and includes usage comments. The state data fields are defined in “Explanation of OTMA State Data Fields” on page 85.

Table 17. REPresynch Command Format

Byte	Length	Content	Value	Meaning	Comments
0	2	**Length**		Length of the state-data section.	Includes the length field itself.
2	8	**Tpipe Name**		The transaction pipe name.	
10	4	**Send-sequence number**		Client recoverable send-sequence number for the transaction pipe.	
14	4	**Receive-sequence number**		Client recoverable receive-sequence number for the transaction pipe.	
18	1	**Tpipe Flag 1**			Mutually exclusive values.
		Continue	X'00'	The last message has been received and IMS continues processing for this synchronized transaction pipe.	
		Dequeue Last Output	X'04'	IMS can dequeue the last output message. The recoverable send-sequence number is updated.	

Table 17. REPresynch Command Format (continued)

Byte	Length	Content	Value	Meaning	Comments
		Reset Sequence Numbers	X'08'	IMS resets the recoverable send-sequence number and the recoverable receive-sequence number, as passed in this command.	
		Stop Tpipe	X'0C'	IMS stops this synchronized transaction pipe.	
		Stop Tpipe & wait for TBresynch	X'10'	IMS stops this synchronized transaction pipe and waits for TBresynch from the client.	
19	1	**Tpipe Flag 2**		Reserved.	
20	6	**Reserved**			

Format of OTMA State Data for TBresynch Command

The TBresynch command is sent by the client to IMS if the client decides it is ready to receive REQresynch from IMS. The TBresynch command can be issued in the following two situations:

- The client has received an ACK message after sending REPresynch with “stop and wait for TBresynch” to IMS.
- The client may request a TBresynch with IMS at any time after the initial nondeferred resynchronization has completed for this tpipe.

Table 18 summarizes the format of state data for the TBresynch command. The summary includes byte, length, content, hexadecimal value, the meaning, and includes usage comments. The state data fields are defined in “Explanation of OTMA State Data Fields” on page 85.

Table 18. TBresynch Command Format

Byte	Length	Content	Value	Meaning	Comments
0	2	**Length**		Length of the state-data section.	Includes the length field itself.
2	8	**Tpipe Name**		The transaction pipe name.	

Format of OTMA State Data for Resume Output for Tpipe

This command is sent by the client to force any queued output to be resent again. The number of Tpipes and Tpipe names are needed in the command.

If the special queue for the Tpipe exists and holds messages, those messages will also be sent to the client. Table 19 on page 85 summarizes the format of state data for Resume Output for Tpipe. The summary includes byte, length, content, hexadecimal value, the meaning, and includes usage comments.

Table 19. Resume Output for Tpipes Command Format

Byte	Length	Content	Value	Meaning	Comments
0	2	**Length**		Length of the state-data section.	
2	2	**Tpipe Count**		Number of Tpipe names in the command.	
4	8	**Tpipe Name**		The transaction tpipe name.	Different Tpipe names can be added as necessary.

Format of OTMA State Data for Resume Output for the Special Queue for Tpipe

An OTMA client sends the command to inform IMS to deliver one or all queued messages on the special queue for Tpipe. If this command is not issued, messages will be held in the special queue. However, the option specified in the command can be used to request how IMS holds and delivers messages. One of the four options in Table 20 can be specified in the State Data. If the client or XCF returns a NAK message to IMS, the current option is reset to `Non-Auto`, which is the default.

Table 20 summarizes the format of state data for Resume Output for the special queue Tpipe. The summary includes, as appropriate, byte, length, content, hexadecimal value, and the meaning. The state data fields are defined in "Explanation of OTMA State Data Fields."

Table 20. Resume Output for the Special Queue for Tpipes Command Format

Byte	Length	Content	Value	Meaning
0	2	**Length**		Length of the state-data section.
2	1	**Option**		
		Non-Auto	X'00'	Exhaust all the messages in the queue only when the command is issued. This is the default.
		One Only	X'01'	Deliver one message in the queue when the command is issued.
		Auto	X'02'	Exhaust all the messages in the queue. After that, automatically deliver messages when they are queued.
		Auto-One	X'04'	Deliver one message automatically when a message is available in the queue. The message could already be in the queue or it might be later. After the message is delivered, this option is reset to `Non-Auto`.

Explanation of OTMA State Data Fields

This section provides explanations for the content of the state data fields of the message prefix:

Length Specifies the total length of the state-data section of the message prefix, including the length field.

Server State Specifies the mode in which the transaction is running.

Conversational State
Specifies a conversational mode transaction. The server sets this state when processing a conversational-mode transaction. This state is also set by the client when sending subsequent IMS conversational data messages to IMS.

Response Mode
Specifies a response-mode transaction. Set by the server when processing a response-mode transaction.

This state has little significance for an OTMA server, because OTMA does not use sessions or terminals.

From Special Queue
Specifies that the output message was sent from the IMS special queue for the Tpipe. The server initially sets this flag when sending a commit-then-send output message. The client also needs to set this flag when sending the subsequent ACK or NAK to IMS.

Synchronization Flag
Specifies the commit mode of the transaction. This flag controls and synchronizes the flow of data between the client and server.

Commit-then-Send
Specifies a commit-then-send transaction. The server commits output before sending it; for example, IMS inserts the output to the IMS message queue.

Send-then-Commit
Specifies a send-then-commit transaction. The server sends output to the client before committing it.

Synchronization Level
Specifies the transaction synchronization level, the way in which the client and server transaction program (for example, IMS application program) interacts with program output messages.

The default is Confirm. IMS always requests a response when sending commit-then-send output to a client.

None
Specifies that no synchronization is requested. The server application program does not request an ACK message when it sends output to a client.

None is only valid for send-then-commit transactions.

Confirm
Specifies that synchronization is requested. The server sends transaction output with the response flag set to `Response Requested` in the message-control information section of the message prefix.

Confirm can be used for either commit-then-send or send-then-commit transactions.

Syncpt
Specifies that the programs participate in coordinated commit processing on resources updated during the conversation under the RRS/MVS recovery platform. A conversation with this level is also called a protected conversation.

Client Flags Specifies optional processing requested by the client.

Send Only Message
This is a send only message; the response is placed on the special queue. This flag is valid only if the client requested a special queue during open.

Reroute Request
Setting this flag reroutes CM0 output to the destination that is specified in the **Destination Override** field.

Map Name Specifies the formatting map used by the server to map output data streams (for example, 3270 data streams). Although OTMA does not provide MFS support, you can use the map name to define the output data stream. The name is an 8-byte MOD name that is placed in the I/O PCB. IMS replaces this field in the prefix with the map name in the I/O PCB when the message is inserted.

The map name is optional.

Server Token Specifies the server name. The Server Token must be returned by the client to the server on response messages (ACKs or NAKs). For conversational transactions, the Server Token must also be returned by the client on subsequent conversational input.

Correlator Token
Specifies a client token to correlate input with output. This token is optional and is not used by the server.

Recommendation: Clients should use this token to help manage their transactions.

Context ID Specifies the RRS/MVS token that is used with SYNCLVL=02 and protected conversations.

Destination Override
Specifies an LTERM name used to override the LTERM name in the IMS application program's I/O PCB. This override is used if the client does not want to override the LTERM name in the I/O PCB with the transaction-pipe name.

This optional override is not used if it begins with a blank.

Server User Data Length
Specifies the length of the server user data, if any. The maximum length of the server user data is 256 bytes. The server user data length is not included in the length calculation.

Server User Data
Specifies any data needed by the server. If included in a transaction message by the client, it is returned by the server in the output data messages.

Member Name
Specifies the XCF-member name of the originating server.

Originator's Token
Specifies the XCF-member token of the originator (either client or server) of the message.

Destination Token
Specifies the XCF-member token of the destination (either client or server) of the message.

DRU Exit Name
Specifies the name of the OTMA Destination Resolution exit routine.

MaxBlocksize Specifies the maximum block size for XCF conversations between the server and the client.

Client-Bid Flag
Specifies the options for the client-bid flag. There is currently one option available.

Special Queue
Specifies that a special queue for a Tpipe is needed. The special queue can hold commit-then-send output that is NAK'd, as well as alternate PCB output for an OTMA client. The output messages in the queue will not be delivered until the client requests that those messages be delivered. Use of the special queue is optional. Without the special queue, a regular queue for the Tpipe will be used to hold and deliver all the Commit-then-send output messages. The default is no special queue for a Tpipe.

OTMA resynchronization protocol currently does not support the special queue.

Aging Value Specifies the access control environment element (ACEE) aging value, in seconds. IMS creates an ACEE if the age of the current ACEE is greater than this value.

The minimum value for caching support is 300 seconds (5 minutes). If the aging value specified is less than the minimum, IMS always creates a non-cached ACEE.

Set Aging Value
Specifies that the user ID aging value is to be used to refresh the cached user ID ACEE. This flag is reset for the output message.

Userid Aging Value
Specifies the aging value in seconds for the input user ID. This field is different from the Aging Value specified in the OTMA client-bid command for OTMA connection. The Aging Value in the client-bid command sets the default aging value for all the OTMA user IDs; however, the Userid Aging Value overrides the default aging value for a specific user ID.

Hash Table Size
Defines the size of the IMS OTMA hash table for processing multi-segment messages input. The table is used to correctly chain all the input segments together. IMS will create the table based on the size specified. Suggested value is X'00000065'.

OTMA Security Data

The security-data section is mandatory for every transaction or command, and is optional for OTMA protocol commands.

Format of OTMA Security Data

Table 21 is a summary of the content of the security-data section of the message prefix. The summary includes, as appropriate, byte, length, content, hexadecimal value, the meaning, and includes usage comments. More information about the content follows the table in “Explanation of OTMA Security Data Fields” on page 90.

Table 21. Content of Security Data Fields

Byte	Length	Content	Value	Meaning	Comments
0	2	Length		Length of the security-data section.	Includes the length field itself.
2	1	Security flag			
		No Security	N	No RACF checking is done.	It is assumed that the user ID and password are already verified.
		Check	C	RACF checks transactions and commands.	Transaction and command authorization RACCHECKs are performed (TCLASS and CCLASS).
		Full	F	RACF checks transactions, commands, and regions.	Transaction, IMS command, and MPP region authorization RACCHECKs are performed.
3	1	**Reserved**			
	1	**Utoken Length**		Length of Utoken plus the length of Utoken Type.	Length does not include length field itself.
	1	**Utoken Type**	X'00'	Type of data to follow.	
	*	**Utoken**		The user token.	Variable length, from 1 to 80 bytes.
	1	**User ID Length**		Length of the user ID plus the length of the User ID Type.	Length does not include length field itself.
	1	**User ID Type**	X'02'	Type of data to follow.	
	*	**User ID**		The user ID.	Variable length, from 1 to 8 bytes.
	u	**Profile Length**		Length of the profile plus the length of the Profile Type.	Length does not include length field itself.
	1	**Profile Type**	X'03'	Type of data to follow.	
	*	**Profile**		The SAF profile.	Variable length, from 1 to 8 bytes.

Explanation of OTMA Security Data Fields

The following provides additional detail on the content of the security-data section of the message prefix:

Length Specifies the length of the security-data section of the message prefix, including the length field.

Security Flag Specifies the type of security checking to be performed. It is assumed that the user ID and password are already verified.

- **No Security** Specifies that no security checking is to be done.
- **Check** Specifies that transaction and command security checking is to be performed.
- **Full** Specifies that transaction, command, and MPP region security checking is to be performed.

Reserved After the reserved field, the following three fields can be omitted or appear in any order. Each field has the following structure:

- Length field
- Field type
- Data field

The length field is not calculated in the length calculation. The actual length of the user ID or profile should not be less than the value specified for the length of each field.

Utoken Length
Specifies the length of the user token plus the length of the user token type.

Utoken Type Specifies that this field contains a user token.

Utoken Specifies the user token. The user ID and profile are used to create the user token. The user token is passed along to the IMS dependent region.

If the client has already called RACF, it should pass the Utoken with field type X'00' so that RACF is not called again.

User ID Length
Specifies the length of the User ID plus the User ID type.

User ID Type Specifies that this field contains a user ID.

User ID Specifies the actual user ID.

Profile Length
Specifies the length of the profile plus the length of the profile type.

Profile Type Specifies that this field contains a profile.

Profile Specifies the system authorization facility (SAF) profile. For RACF, this is the group name.

OTMA User Data

The user-data section is variable length and follows the security-data section of the message prefix. It can contain any data.

Format of OTMA User Data

Table 22 is a summary of the content of the user-data section of the message prefix. The summary includes, as appropriate, byte, length, content, hexadecimal value, the meaning, and includes usage comments. More information about the content follows the table in “Explanation of OTMA User Data Fields.”

Table 22. Content of User Data Fields

Byte	Length	Content	Value	Meaning	Comments
0	2	**Length**		Length of the user-data section.	The length includes the length field itself.
2	*	**User data**		The user data.	Optional; variable length.

Explanation of OTMA User Data Fields

The following provides additional detail on the content of the user-data section of the message prefix:

Length Specifies the length of the user-data section of the message prefix, including the length field. The maximum length of the user data is 1024 bytes.

User Data Specifies the optional user data. This data is managed by the client, and can be created and updated using the DFSYDRU0 exit routine. The server returns this section unchanged to the client as the first segment of any output messages.

OTMA Application Data

The application-data section is variable length and follows the user-data section of the message prefix. You include IMS commands and transactions in the application-data section. The data in this section is unchanged by the receiver (server or client), and is transmitted directly to the server application program or to the client application program.

Format of OTMA Application Data

Table 23 is a summary of the content of the application-data section of the message prefix. The summary includes, as appropriate, byte, length, content, hexadecimal value, the meaning, and includes usage comments. More information about the content follows the table in “Explanation of OTMA Application Data Fields” on page 92.

Table 23. Application Data

Byte	Length	Content	Value	Meaning	Comments
0	2	**Length**		Length of the application-data section.	The length includes the length field itself. The maximum length is 32KB (32767 bytes).
2	2	**ZZ**		Application data IMS ZZ fields.	
4	*	**Application data**		The application data.	Variable length. The maximum length of application data is 32KB-4.

Explanation of OTMA Application Data Fields

The following provides additional detail on the content of the application-data section of the message prefix:

Length Specifies the length of the application-data section of the message prefix, including the length field.

Application data
Specifies the optional application data.

Multiple send requests might be required for a server output segment. For a client's transaction, the transaction code is specified in the first 8 bytes of the data area following the LLZZ. For transactions specified with MULTSEG, the standard IMS LLZZ format is required for each segment. The transaction code is only required in the first segment.

Sample OTMA Messages

This section shows three sample OTMA messages. They are not necessarily related to each other, but are intended to show what OTMA messages look like when fully constructed, including the parts of the message prefix.

Figure 34 shows an OTMA client-bid message. The total length of the state-data section plus the security-data section of the message prefix is X'8C' bytes.

```
MESSAGE CONTROL INFORMATION:

   01102000 04004040 40404040 4040A0C0     |......        ff{|
   00000000 00000000 00000000 00000400     |................|

STATE DATA + SECURITY DATA:

   0036C3D3 C9C5D5E3 F1404040 40404040     |..CLIENT1       |
   40400100 00010003 00020100 00010003     |  ..............|
   0001C4C6 E2E8C4D9 E4F02000 00007FFF     |..DFSYDRU0....".|
   FFFF0000 00650056 C3525100 50018059     |........C...&...|
   15569555 55555555 55555555 B7B686B0     |..n.........%&f[|
   81A61515 1B1B1B1B 1B1B1B1B B7B686B0     |aw.........%&f[|
   81A61515 55555555 55555555 8C918CA4     |aw.........._j_u|
   15151515 55555555 55555555 09151515     |................|
   15151515 15151515 15151515              |................|
```

Figure 34. OTMA Client-Bid Message

Figure 35 on page 93 shows an OTMA transaction message. The total length of the state-data, security-data, and application-data sections of the message prefix is X'D6' bytes.

```
MESSAGE CONTROL INFORMATION:

   01402000 0000E3D7 C9D7C5F1 4040A0D0     |. ....TPIPE1  ff}|
   00000001 00000000 00000001 00010000     |................|

STATE DATA + SECURITY DATA + APPLICATION DATA:

   00480020 0100E3C5 E2E3D4C1 D7400000     |......TESTMAP ..|
   00000000 00000000 00000000 0000C9D4     |..............IM|
   E2F0F0F0 F0F10000 00000000 00004040     |S00001........  |
   40404040 40404040 40404040 40404040     |                |
   40404040 40400000 0056C652 51005001     |      ....F...&.|
   80465551 95555555 55555555 55555555     |....n...........|
   55555555 55555555 55555555 55555555     |................|
   55555555 55555555 55555555 555586A3     |..............ft|
   A781B0B7 A4155555 55555555 5555B1B7     |xa[%u.........&%|
   8CB6A5A5 A415B7BD B7A41515 15150038     |%&vvu.%"%u......|
   0000C1D7 D6D3F1F8 4040E2C1 E840C8C5     |..APOL18  SAY HE|
   D3D3D640 40404040 40404040 40404040     |LLO             |
   40404040 40404040 40404040 40404040     |                |
   40404040 4040                           |                |
```

Figure 35. OTMA Transaction Message

Figure 36 shows an OTMA response message. The total length of the state-data, security-data, and application-data sections of the message prefix is X'EE' bytes.

```
MESSAGE CONTROL INFORMATION:

   01A08000 0000E3D7 C9D7C5F1 404080D0     |.ff....TPIPE1  .}|
   00000001 00000000 00000001 00010000     |................|

STATE DATA + SECURITY DATA + APPLICATION DATA:

   00480020 0100E3C5 E2E3D4C1 D740AB7F     |......TESTMAP %"|
   28EB9FAD 9A024040 40404040 4040C9D4     |.. Y..        IM|
   E2F0F0F0 F0F10000 00000000 00004040     |S00001........  |
   40404040 40404040 40404040 40404040     |                |
   40404040 40400000 0056C652 51005001     |      ....F...&.|
   80465551 95555555 55555555 55555555     |....n...........|
   55555555 55555555 55555555 55555555     |................|
   55555555 55555555 55555555 555586A3     |..............ft|
   A781B0B7 A4155555 55555555 5555B1B7     |xa[%u.........&%|
   8CB6A5A5 A415B7BD B7A41515 15150050     |%&vvu.%"%u.....&|
   0300D6E4 E3E2C5C7 40D5D67E F0F0F0F0     |..OUTSEG NO=0000|
   F140E2D7 C5C3C9C6 C9C5C440 E2C5C7E2     |1 SPECIFIED SEGS|
   C9E9C57E F0F0F0F8 F06B40E2 C5C7D5D6     |IZE=00080, SEGNO|
   7EF0F0F0 F0F34040 40404040 40F67EC3     |=00003       6=C|
   D6D340F6 F04040F7 7EC3D6D3 40F7         |OL 60  7=COL 7..|
```

Figure 36. OTMA Response Message

Chapter 6. OTMA Architected Transaction Attributes

This chapter describes the syntax of architected command output.

When you issue an IMS /DISPLAY TRANSACTION command from OTMA, the output is in the form of an OTMA message, returned to the client in the application-data section of the message prefix. Table 24 shows the Attributes Segment for a given transaction. The description includes byte, length, content, hexadecimal value, the meaning, and includes usage comments where appropriate.

Table 24. Transaction Attributes Segment

Byte	Length	Content	Value	Meaning	Comments
0	2	**Length**		Length of the Transaction Attributes Segment (LL).	This length includes the length field itself.
2	2	**ZZ**			
4	8	**Transaction Code**		The 8-byte IMS transaction code.	
12	1	**Transaction type flag 1**			The flag 1 values are mutually exclusive.
		Valid	X'00'	A valid OTMA transaction.	
		CPIC	X'04'	A CPI-C (APPC) transaction.	
		FPX	X'08'	A Fast Path-exclusive transaction.	
		FPP	X'0C'	A Fast Path-potential transaction.	
		MSC	X'10'	An MSC remote transaction.	
		Invalid Syntax	X'FE'	Syntax error.	The data area contains the text of the error.
		Invalid	X'FF'	Transaction not found, or invalid.	
13	1	**Transaction type flag 2**			The flag 2 values are **not** mutually exclusive.
		Response	X'80'	An IMS response mode transaction.	
		Conversation	X'40'	An IMS conversational transaction.	
		Update	X'20'	The transaction has update capability.	
		Irrecoverable	X'10'	The transaction is defined as irrecoverable.	
		Multi-Segment	X'08'a	The transaction has multiple segments.	
		Uppercase	X'04'	Uppercase translation requested.	

Table 24. Transaction Attributes Segment (continued)

Byte	Length	Content	Value	Meaning	Comments
14	1	**Transaction Status**			The indicated values are **not** mutually exclusive.
		STOP	X'80'	The transaction input queueing is stopped.	One of the following IMS commands was issued for the transaction: • /STOP • /PURGE
		OLC	X'40'	The transaction input queueing is stopped.	One of the following IMS commands was issued for the transaction: • /MOD PREPARE • /MOD COMMIT
		NOSCH	X'20'	IMS scheduling is stopped.	One of the following IMS commands was issued for the transaction: • /PSTOP • /LOCK
15	1	**Reserved**			
16	2	**Length**		Length (LL) of error text, if any.	If there is error text, it replaces all subsequent sections of the message.
18	*	**Error Text**		Text of the error message.	Variable length. The error test section is applicable only if transaction type flag 1 is set to Invalid Syntax (X'FE')
16	8	**PSB Name**		IMS PSB name.	Only present if there is no error text.
24	1	**Class**		SMB message class for IMS scheduling.	
25	1	**Current Priority**		Current SMB priority.	
26	1	**Normal Priority**		Normal SMB priority.	
27	1	**Limit Priority**		SMB limit priority.	
28	2	**Enqueue Count**		Number of messages enqueued.	
30	2	**Dequeue Count**		Number of messages dequeued.	
32	2	**Enqueue Limit**		Enqueue limit count.	
34	2	**Processing Limit Count**		Processing limit count.	
36	2	**Output Max Segment Length**		The maximum output segment length.	
38	2	**Output Limit of Message Segments**		The output limit of message segments.	
40	2	**Parallel Limit**		The PARLIM value from TRANSACTION statement.	

Table 24. Transaction Attributes Segment (continued)

Byte	Length	Content	Value	Meaning	Comments
42	1	**Region Count**		The number of regions in which the transaction is currently scheduled.	

Chapter 7. OTMA Callable Interface

This chapter introduces the OTMA Callable Interface, and describes how to use it.

In this chapter :

Introduction to OTMA Callable Interface

The IMS OTMA Callable Interface (C/I), which became available with IMS Version 6, provides a high-level interface for access to IMS applications from other z/OS subsystems. The interface consists of API calls that are available to a C/C++ program. The API calls are used to join the IMS/OTMA XCF group, to connect to IMS, to allocate communication sessions, to send IMS transactions/commands, to receive output from IMS, to close communication sessions, and to leave the XCF group.

Figure 37 on page 100 provides an overview of the OTMA C/I. Shown from left to right, in a sample z/OS environment, are sample C and C++ API calls (for example, OTMA_OPEN). The API calls pass through an object stub, the SVC interface routine, the API (for example, DFSYOPEN), and finally through the XCF group to IMS OTMA.

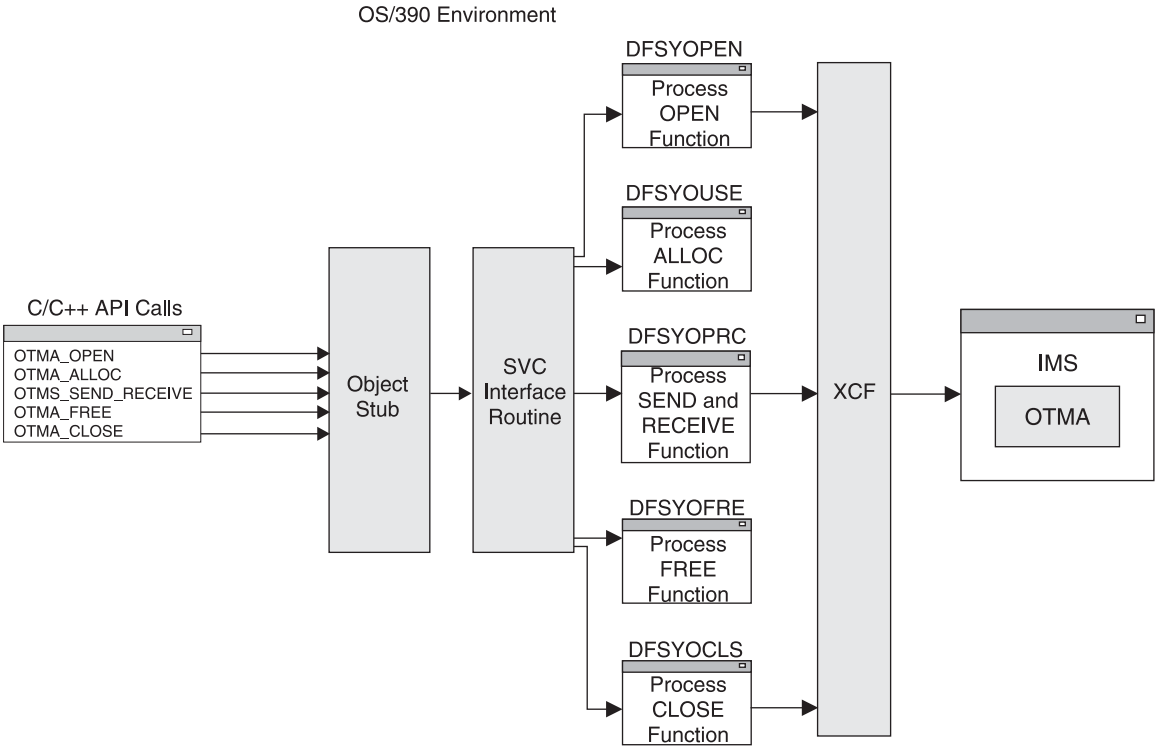

Figure 37. OTMA Callable Interface Overview

The program that invokes the API calls can be running from an authorized or unauthorized library in problem or supervisor state. DFSYC0, a C header file, is provided to define the API calls. DFSYCRET, a load module, contains the entry points for each API call and is linked with the application program. OTMA C/I uses BPE SVC Services to process the API call.

Benefits:A key benefit of OTMA C/I is that it is easy to use.

Other reasons to use OTMA C/I are that it:

- Extracts out the details of OTMA and XCF
- Submits IMS transactions and commands
- Enables programs running from other z/OS subsystems to connect to multiple IMSs
- Calls the APIs from an authorized or unauthorized library
- Connects to all IMS OTMA releases

Getting Started with OTMA C/I

This section describes the information that you need to begin using the OTMA C/I.

In this Section:

OTMA C/I Environment Requirements

To initialize and install OTMA C/I, OS/390® Release 3 or above and IMS Version 6 or above are required.

OTMA C/I Migration and Coexistence

OTMA C/I requires IMS Version 6 or above for initialization and installation. OTMA C/I can coexist with all OTMA releases.

RAS criteria, design elements, support plan, and hardware programming support remain consistent with IMS Version 6.

OTMA C/I Initialization

OTMA C/I provides a stand-alone program, DFSYSVI0, that must be run after the z/OS IPL to initialize the OTMA C/I. DFSYSVI0 invokes DFSYSVC0, one of the OTMA C/I modules. DFSYSVC0 loads and registers the SVC services by an authorized address space running on the same z/OS image as the application programs accessing it.

You must add an entry in the z/OS program properties table (PPT) for the OTMA Callable Interface initialization program. The steps for doing this are:

1. Edit the SCHED*xx* member of the SYS1.PARMLIB data set.
2. Add the following entry to the SCHED*xx* member:

```
PPT PGMNAME(DFSYSVI0)   /* PROGRAM NAME = DFSYSVI0            */
             CANCEL     /* PROGRAM CAN BE CANCELED            */
             KEY(7)     /* PROTECT KEY ASSIGNED IS 7          */
             SWAP       /* PROGRAM IS SWAPPABLE               */
             NOPRIV     /* PROGRAM IS NOT PRIVILEGED          */
             DSI        /* REQUIRES DATA SET INTEGRITY        */
             PASS       /* CANNOT BYPASS PASSWORD PROTECTION  */
             SYST       /* PROGRAM IS A SYSTEM TASK           */
             AFF(NONE)  /* NO CPU AFFINITY                    */
             NOPREF     /* NO PREFERRED STORAGE FRAMES        */
```

3. Take one of the following actions to make the SCHED*xx* changes effective:

   Re-IPL the z/OS system.

   or

   Issue the `MVS SET SCH=` command.

   Related Reading: For additional reading about updating the program properties table, see *z/OS MVS Initialization and Tuning Reference*.

A sample JCL procedure for running DFSYSVI0 is as follows:

```
//OTMAINIT PROC RGN=3000K,SOUT=A,
//              PARM1=
//*
//IEFPROC  EXEC PGM=DFSYSVI0,
//              REGION=&RGN
//*
//STEPLIB  DD   DISP=SHR,UNIT=SYSDA,
//              DSN=IMSVS.SDFSRESL
//*
//SYSPRINT DD   SYSOUT=&SOUT
//SYSUDUMP DD   SYSOUT=&SOUT
//*
```

OTMA C/I Security

To protect XCF groups from any non-authorized caller use IMSXCF.OTMACI, a RACF resource, defined in the RACF facility class for the OTMA C/I. When the RACF resource is defined, RACF RACHECK is invoked before OTMA C/I performs a XCF JOIN. This method protects the access to XCF, the XCF group, and the member. This RACF checking is performed only when a non-authorized caller is using OTMA C/I.

Additional security characteristics remain consistent with OTMA in IMS 6.1.

OTMA C/I Restrictions

These restrictions apply to the OTMA C/I:

- OTMA C/I must be initialized and installed in IMS 6.1 and above, but OTMA C/I can connect to all IMS OTMA releases.
- Application program languages other than C and C++ are not currently supported by OTMA C/I.
- All OTMA calls must be made in the same state (PSW key, supervisor or problem state, authorized or non-authorized) as the otma_open call. For example: If you were authorized when you did the otma_open call, you must be authorized for all subsequent calls.
- The resynchronization feature of IMS OTMA is not supported.
- IMS command /SECURE OTMA PROFILE, is not currently supported.

OTMA C/I Hints and Tips

This section describes several usage tips for the OTMA C/I.

- C/I must be installed in an OS/390 or z/OS environment before it can be invoked. If C/I is not installed and invoked, an F92 abend occurs when otma_create or otma_open is issued. If C/I is not properly installed, a DFS3911E error message occurs.
- otma_open, otma_openx, otma_send_receive, otma_send_receivex, otma_send_async, and otma_receive_async each have an ECB parameter. This ECB is posted by the function or by an SRB routine that the function precipitates. The caller must check the ECB and wait for it to be posted before inspecting the return code and output data. Be sure to initialize the ECB with 0 before passing to the C/I call.
- Each otma_alloc call creates an independent session for the subsequent otma_send_receive call. One of the otma_alloc calls can be used to specify the name of IMS transaction or IMS command to be sent to IMS. The maximum length of the transaction name is 8 characters. If no transaction name or command is specified in the otma_alloc call, the transaction name, followed by one or more blanks, or command needs be specified in the beginning of the send buffer of the otma_send_receive call. After the otma_send_receive call, otma_free is required, except for the IMS conversation transaction. See the invocation sample C for sending a conversation transaction.
- C/I builds the standard LLZZ part of IMS application data format. You do not need to worry about the LLZZ at all.
- To send a multi-segment message to IMS, the send segment list of the otma_send_receive call must identify the length of each input segment. The first element in the segment list specifies the number of the segment. The first element is then followed by the length of segment 1, the length of segment 2, and so on.
- When a multi-segment output message is received, an output segment list is provided for the otma_send_receive call. The first element in the output segment

list contains the number of the output segment. The first element is then followed by the length of output segment 1, the length of output segment 2, and so on.

- Sample programs (DFSYCSMP) are shipped with IMS. See also “OTMA C/I Sample Program #1: Synchronous Processing” on page 127 and “OTMA C/I Sample Program #2: Asynchronous Processing” on page 138.
- C/I can be used to send a protected transaction to IMS by passing a context token to the otma_send_receive call.
- Because some of the C/I calls require the calling program to wait, implementing the time-out routine in the calling program is highly recommended to avoid long running transactions in IMS and the internal C/I hang.
- To run the C/I application efficiently, limit the number of otma_open and otma_close calls in the application. Also, for all otma_open and otma_create calls, try to use the same member name rather than generating a different member name for each call.
- If the size of the output receive buffer specified in the otma_send_receive call is too small, the actual data returned is limited by the size of the receive buffer. The output can be rejected if a special option, SyncLevel1, is specified in the otma_alloc call. However, if the size of the output receive buffer is too small for the otma_receive_async call, C/I always rejects the output.
- C/I can support various program-to-program switches in IMS. See “OTMA Program-to-Program Switch Processing” on page 55 for more information.
- C/I could return a bad return code to inform the caller about an abnormal condition. Logging or saving the bad return code for debugging purpose is recommended.
- The otma_send_receive call sends an OTMA send-then-commit message with synclevel=none to IMS. The caller can set a synclevel=confirm for otma_send_receive.
- When an input RRS context token is given in the otma_send_receive call, the synclevel is then changed to SYNCPT to support the protected transaction.
- For complex program-to-program switches in IMS, a send-then-commit input message could result in a commit-then-send output message instead of the expected send-then-commit output message. C/I works in this special scenario. See “OTMA Program-to-Program Switch Processing” on page 55 for more information on program-to-program switches.
- The otma_send_async call sends an OTMA commit-then-send message to IMS.
- The otma_receive_async call receives an OTMA commit-then-send output message from IMS.
- C/I does not support either the OTMA resync protocol or the OTMA security PROFILE option.

OTMA C/I APIs

This section gives a detailed description of all of the OTMA callable interface application programming interfaces (APIs).

Using the Header File DFSYC0.H:

The header file included in the API calling program declares each API invocation and variables used for the invocation.

For a C/C++ program using OTMA Callable Interface, the C/C++ header file, DFSYC0.H, needs to be included in the C/C++ program.

Load Module DFSYCRET:

The object stub, DFSYCRET, receives all the API invocations and issues a SVC call to perform the requested function. The object stub needs to be available during the link-editing of the API invoking program. DFSYCRET can be found in SDFSRESL or ADFSLOAD data sets.

The functions implemented by the API are:

CALL	FUNCTION
otma_create	Creates storage structures to support communications but does not establish a connection with IMS.
otma_open	Establishes a connection with IMS. Issue an `otma_create` prior to establishing an `otma_open` call.
otma_openx	Provides the same function as `otma_open` API, with an added parameter to specify OTMA Destination Resolution User (DRU) exit name routine and special options.
otma_alloc	Creates an independent transaction session.
otma_send_receive	Sends to IMS and passes parameters for receive functions.
otma_send_receivex	Provides the same function as `otma_send_receive` API, with an added parameter to pass OTMA user data.
otma_send_async	Sends input (transaction or IMS command) only to IMS.
otma_receive_async	Receives unsolicited or queued output from IMS.
otma_free	Releases the independent transaction session.
otma_close	Ends the connection with IMS.

In this Section:

Using otma_create

Description

The `otma_create` API is to allocate storages for XCF and IMS communications. After the call, an anchor will be returned. The anchor must be used for the subsequent calls. Invoking `otma_create` is not required. During the `otma_open`,

OTMA C/I will allocate storages for communication, if it detects that `otma_create` has not been called. If `otma_create` is invoked first, the same input parameters need to be used again for the subsequent `otma_open` call.

Invocation

Called by the client in TCB mode.

Input

***ecb** Pointer to the next event control block.

***group_name** Pointer to the string containing the XCF group name. (char[8])

***member_name**
Pointer to the string containing the XCF member name for this member. (char[16])

***partner_name**
Pointer to the string containing the XCF member name for IMS. (char[16])

***sessions** Number of parallel sessions that are intended to be supported with IMS. Long integer from 001 to 999.

***tpipe_prefix** First 1 to 4 characters of the TPIPE names. (char[4])

For more information on OTMA Tpipe naming conventions, see “OTMA Naming Conventions” on page 14.

Attention: For the input fields **group_name**, **member_name**, and **partner_name**, all XCF names that are pointed to must be left justified, filled with blanks, and consist of legal upper case EBCDIC characters. If any of those naming rules are violated, underlying XCF errors will be reported.

Output

***anchor** Pointer to the anchor word.

***retrsn** Pointer to the return code structure.

C-Language Function Prototype

```
otma_create(
    otma_anchor_t          *anchor,        [out]
    otma_retrsn_t          *retrsn,        [out]
    ecb_t                  *ecb,           [in]
    otma_grp_name_t        *group_name,    [in]
    otma_clt_name          *member_name,   [in]
    otma_srv_name          *partner_name,  [in]
    signed long int        *sessions,      [in]
    unsigned char          *tpipe_name);   [in]
```

Return Values (rc value)

The rc and reason are valid after ECB has been posted. For the complete description of each error, see Table 25 on page 119.

0 The call was completed successfully.

8 User error.

12 Storage obtain failure.

Using otma_open

Description

The caller must call `otma_open` to connect when IMS is available. The caller must wait on the ECB, that is posted when the connection is completed or when the attempt has failed. When IMS is not up or OTMA is not started the attempt will fail.

The caller can cancel the attempt to connect with IMS by issuing an `otma_close` call at any time. The ECB will be posted accordingly.

If IMS fails after this connection is established, any call to a function interface will receive a return code to indicate that IMS is no longer listening for messages. If IMS resumes before a close is performed, the connection will be reestablished without any action from the client. The `otma_close` and `otma_open` interfaces may be called again to reestablish communications with IMS. All existing conversations will have been terminated. This implementation does not use OTMA Resynchronization Protocol.

An extended version of the `otma_open` API, which is called `otma_openx`, provides extended functionality. See “Using otma_openx” on page 107 for more details about the `otma_openx` API.

Invocation

Called by the client in TCB mode.

Input

***anchor** Pointer to the anchor word. If `otma_create` is not used to set up the anchor environment, the anchor word must be initialized with a zero.

***group_name** Pointer to the string containing the XCF group name. (char[8])

***member_name**
Pointer to the string containing the XCF member name for this member. (char[16])

***partner_name**
Pointer to the string containing the XCF member name for IMS. (char[16])

***sessions** Number of parallel sessions that are intended to be supported with IMS. Long integer from 001 to 999.

***tpipe_prefix** First 1 to 4 characters of the TPIPE names. (char[4]).

For more information on OTMA tpipe naming conventions, see “OTMA Naming Conventions” on page 14

Attention: For the input fields **group_name**, **member_name**, and **partner_name**, all XCF names that are pointed to must be left justified, filled with blanks, and consist of legal upper case EBCDIC characters. If any of those naming rules are violated, underlying XCF errors will be reported.

Output

***anchor** Pointer to the anchor word to receive the address of global storage.

***retrsn** Pointer to the return code structure.

***ecb** Pointer to the event control block to be posted when the open completes.

C-Language Function Prototype

```
otma_open(
          otma_anchor_t   *anchor           [in/out]
          otma_retrsn_t   *retrsn,          [out]
          ecb_t           *ecb,             [out]
          otma_grp_name_t *group_name,      [in]
          otma_clt_name_t *member_name,     [in]
          otma_srv_name_t *partner_name,    [in]
          signed long int *sessions,        [in]
          unsigned char   *tpipe_name);     [in]
```

Post Codes

The caller of the OPEN routine must check the ECB that was provided to OPEN. If this ECB is not already posted, the caller must wait for this ECB (for the OPEN protocol to complete).

0 XCF OPEN completes successfully.

4 IMS is not ready. Try again later.

8 Your XCF group and member are already active.

12 A system error occurred.

Return Values (rc value)

The rc and reason are valid after ECB has been posted.

0 XCF JOIN was successful, client-bid was sent, and acknowledgment received. For the complete description of each error, see Table 25 on page 119.

4 IMS is not ready. Try again later.

8 Your XCF group and member are already active.

12 A system error occurred.

Using otma_openx

Description

The `otma_openx` API has the same functionality as the `otma_open` API, with the following extended features:

- The ability to specify an OTMA DRU exit routine
- Added capability for future enhancements to the API

Invocation

Same as for the `otma_open` API.

Input

Same as for the `otma_open` API, with the following additional parameters:

***ims_dru_name**
Pointer to the string containing the user-defined OTMA Destination Resolution User Exit Routine.

***special_options**
Pointer to an area codifying non-standard options. Currently, there are no special options supported. Specify a NULL for this parameter.

Output
Same as for the otma_open API.

C-Language Function Prototype
```
otma_openx(
          otma_anchor_t     *anchor,           [out]
          otma_retrsn_t     *retrsn,           [out]
          ecb_t             *ecb,              [out]
          otma_grp_name_t   *group_name,       [in]
          otma_clt_name_t   *member_name,      [in]
          otma_srv_name_t   *partner_name,     [in]
          signed long int   *sessions,         [in]
          tpipe_prfx_t      *tpipe_prefix,     [in]
          otma_dru_name_t   *ims_dru_name,     [in]
          otma_profile4_t   *special_options); [in]
```

Post Codes
Same as for the otma_open API.

Return Values (rc value)
Same as for the otma_open API.

Using otma_alloc

Description
The otma_alloc API is called to create an independent session to exchange messages.

Invocation
Called by the client in TCB mode.

Input

***anchor** Pointer to anchor word that was set up by otma_open.

***username** Pointer to string holding the RACF username for transaction/commands.

For calls from authorized programs, the input username is trusted and passed to IMS. For calls from unauthorized programs, OTMA C/I invokes a RACF call with the current ACEE context to obtain the username. The input username, if any, will be ignored. A NULL can be specified for callers from unauthorized programs.

***transaction** Name of IMS transaction or command to be sent to IMS.

If the IMS command entered is longer than eight characters, the first eight characters of the command can be provided in this parameter. The rest of the characters of the command need to be provided in the beginning of the send buffer of the subsequent otma_send_receive API.

If this parameter is left blank, then the IMS transaction name or command must be specified (left aligned) in the beginning of the send buffer of the subsequent otma_send_receive API.

***prfname** Pointer to a string holding the RACF group name for transactions/commands.

***special_options**
Pointer to the processing options for the subsequent otma_send_receive or otma_send_receivex API call. The supported processing options include:

Bit 0 SyncOnReturn - with this option, IMS is asked to process the message without the RRS context token; in this case, the user ID is obtained when RRS CTXRDTA is invoked.

Bit 1 SyncLevel1 - with this option, OTMA send_then_commit sync level 1 is used instead of sync level 0, which is the default for OTMA C/I. Refer to the DFSYCO header file for additional information.

Output

***retrsn** Pointer to return code structure.

***session_handle**
Pointer to session handle that uniquely identifies the session for the subsequent otma_send_receive.

C-Language Function Prototype

```
otma_alloc(
          otma_anchor_t  *anchor,           [in]
          otma_retrsn_t  *retrsn,           [out]
          sess_handle_t  *session_handle,   [out]
          otma_profile_t *special_options,  [in]
          tran_name_t    *transaction,      [in]
          racf_uid_t     *username,         [in]
         racf_prf_t     *prfname);          [in]
```

Return Values (rc value)

The rc and reason are valid after ECB has been posted. For the complete description of each error, see Table 25 on page 119.

0 Success.

4 Session limit reached.

8 Null anchor.

Using otma_send_receive

Description

The otma_send_receive API is invoked to initiate a message exchange with IMS. The caller gives buffer definitions for both send and receive. Both send buffer and receive buffer information is provided. By providing receive information at the same time as send there are no unexpected messages from IMS, greatly simplifying the protocol. When the reply arrives from IMS the ECB will be posted. All the work of buffer management is handled in the message exit routine.

An extended version of the otma_send_receive API, which is called otma_send_receivex, provides extended functionality. See “Using otma_send_receivex” on page 112 for more details about the otma_send_receivex API.

Invocation

Called by the client in TCB mode.

Input

***anchor** Pointer to anchor word that was set up by `otma_open`.

***session_handle**
Pointer to session handle for `TPIPE` returned by `otma_alloc`.

***lterm** Pointer to `lterm` name field. On input is passed to IMS. Will be updated on output to `lterm` field returned by IMS. Can be blank in both cases.

***modname** Pointer to `MODname` name field. On input is passed to IMS. Will be updated on output to `MODname` field returned by IMS. May be blank in both cases.

If the input modname is DFSM01, DFSMO2, or DFSM05, it will be treated as blanks.

***send_buffer** Pointer to the data to be sent to IMS. When a NULL is specified for the transaction parameter, the client code must provide the transaction name or command, and a blank, to the data in this buffer when sending to IMS.

***send_length** Length of send data.

***send_segment_list**
An array of lengths of message segments to be sent to IMS. First element is count of following segment lengths. Optional: If a single segment is to be sent, either the first element or the address of the array can be zero.

***receive_buffer**
Pointer to buffer to receive reply message from IMS.

***receive_length**
Length of buffer available to receive message.

***receive_segment_list**
An array to hold the number of segments sent by IMS. First element must be set as the number of elements in the array. Optional: If a single segment is to be received, either the first element or the address of the array can be zero. In which case all segments will be received contiguously without indication of segmentation boundaries.

***context_id** Null or Distributed Sync Point Context ID from RRS.

- For an authorized caller, OTMA C/I passes the Context ID directly to IMS and does not validate the Context ID data.
- For an unauthorized caller, OTMA C/I invokes the CTXSWCH call to disassociate the token and to validate if the token is current for a task. When OTMA C/I receives a response from IMS, it switches the context back onto the task before returning control to the caller.

Output

***retrsn** Pointer to return code structure.

***ecb Event** Control block to be posted when the message exchange is complete.

***received_length**
Field to receive length of data received to `receive_buffer`. Should be equal to the sum of the segment lengths.

***receive_segment_list**
An array of lengths of message segments received from IMS. First element is count of following segment lengths and must be set by client to indicate maximum length of array. It will be modified by receive.

***error_message**
Address of the pointer to the error message field. It is provided by the user to receive error or informational messages from IMS. If the post code returns a 20, then this field will contain data.

C-Language Function Prototype

```
otma_send_receive(
             otma_anchor_t  *anchor,                [in]
             otma_retrsn_t  *retrsn,                [out]
             ecb_t          *ecb,                   [out]

             sess_handle_t  *session_handle,        [in]
             lterm_name_t   *lterm,                 [in/out]
             mod_name_t     *modname,               [in/out]

             char           *send_buffer,           [in]
             data_leng_t    *send_length,           [in]
             ioseg_list_t   *send_seg_list,         [in]

             char           *receive buffer,        [in]
             data_leng_t    *receive_length,        [in]
             data_leng_t    *received_length,       [out]
             ioseg_list_t   *receive_segment_list,  [in/out]
             context_t      *context_id,            [in]

             char           *error_message);        [out]
```

Post Codes

0 Normal completion.

8 No anchor/bad session handle/segment too large.

12 Send failed.

16 Receive has been cancelled.

20 Error from IMS.

Return Values (rc value)

The rc and reason are valid after ECB has been posted. For the complete description of each error, see Table 25 on page 119.

0 Normal completion.

8 No anchor/bad session handle/segment too large.

12 Send failed.

16 Receive has been cancelled.

20 Error from IMS.

Using otma_send_receivex

Description

The otma_send_receivex API has the same functionality as the otma_send_receive API, but adds the extended ability to pass OTMA user data.

Invocation

Same as for the otma_send_receive API.

Input

Same as for the otma_send_receive API, with the following additional parameter:

***otma_user_data**

Pointer to the OTMA user data. The OTMA user data field can contain any user data that is used to identify the user input, or to correlate input with output. If a value is specified in this field, the data is sent to IMS. IMS user exits DFSYIOE0 and DFSYDRU0 can read or change the data. The data is returned to the user if the otma_receive_async API with otma_user_data is issued.

If there is no OTMA user data, specify a NULL for this field.

Output

Same as for the otma_send_receive API.

C-Language Function Prototype

```
otma_send_receivex(
            otma_anchor_t  *anchor,                [in]
            otma_retrsn_t  *retrsn,                [out]
            ecb_t          *ecb,                   [out]

            sess_handle_t  *session_handle,        [in]
            lterm_name_t   *lterm,                 [in/out]
            mod_name_t     *modname,               [in/out]

            char           *send_buffer,           [in]
            data_leng_t    *send_length,           [in]
            data_leng_t    *send_segment_list,     [in]

            char           *receive buffer,        [in]
            data_leng_t    *receive_length,        [in]
            data_leng_t    *received_length,       [out]
            data_leng_t    *receive_segment_list,  [in/out]
            context_t      *context_id,            [in]

            char           *error_message,         [out]
            otma_user_t    *otma_userdata);        [in/out]
```

Post Codes

Same as for the otma_send_receive API.

Return Values (rc value)

Same as for the otma_send_receive API.

Using otma_send_async

Description

The otma_send_async API is invoked to send a transaction or command to IMS.

Restriction: This API cannot be used to submit an IMS fast path transaction, a protected transaction (the transactions with RRS context IDs), or an IMS conversational transaction. For these three types of transactions, use the `otma_send_receive` API instead.

Invocation

Called by the client in TCB mode.

Input

***anchor** Pointer to anchor word that was set up by `otma_open`.

***lterm** Pointer to `lterm` name field. If there is no input `lterm`, specify a NULL.

***modname** Pointer to `MODname` name field. If there is no input `MODname`, specify a NULL.

***otma_user_data**
Pointer to the OTMA user data. This 1022-byte field is optional. The OTMA user data field can contain any user data that is used to identify the user input, or to correlate input with output. If a value is specified in this field, the data is sent to IMS. IMS user exits DFSYIOE0 and DFSYDRU0 can read or change the data. The data is returned to the user if the `otma_receive_async` API with `otma_user_data` is issued.

If there is no OTMA user data, specify a NULL for this field.

***prfname** Pointer to string holding the RACF group name for transactions/commands. This parameter is optional. If there is no input RACF group name, specify a NULL.

***send_buffer** Pointer to the data to be sent to IMS. When a NULL is specified for the transaction parameter, the client code must provide the transaction name or command, and a blank, to the data in this buffer when sending to IMS.

***send_length** Length of send data.

***send_segment_list**
An array of lengths of message segments to be sent to IMS. This parameter is required for multi-segment input messages. If specified, the first element needs to contain the count of total input segments. This field is optional for single segment input. If a single segment is to be sent, either the first element or the address of the array can be zero.

***special_options**
Pointer to an area codifying non-standard options. Currently, no special options are supported. Specify a NULL for this parameter.

***tpipe_name** Pointer to OTMA tpipe name field. This name must be different from the tpipe name specified for the `otma_create` and `otma_open` APIs.

***transaction** Name of IMS transaction or command to be sent to IMS.

If the IMS command entered is longer than eight characters, the first eight characters of the command can be provided in this parameter. The rest of the characters of the command need to be provided in the beginning of the send buffer.

If NULL or blanks are specified in this parameter, OTMA C/I expects the user to include the IMS transaction name or command in the beginning of the send buffer.

***username** Pointer to a string holding the RACF username for transaction/commands.

For calls from authorized programs, the input username is trusted and passed to IMS. For calls from unauthorized programs, OTMA C/I invokes a RACF call with the current ACEE context to obtain the username. The input username, if any, will be ignored. A NULL can be specified for callers from unauthorized programs.

Output

***ecb Event** Event control block to be posted when IMS receives or rejects the input.

***error_message**
Address of the pointer to the error message field. It is provided by the user to receive error or informational messages from IMS. If the post code returns a 20, then this field will contain data.

***retrsn** Pointer to return/reason code structure. If IMS OTMA rejects the input, the NAK code and its associated reason code are available in OTMA C/I reason codes 2 and 3. See “OTMA Return Codes” on page 67 for an explanation of the NAK code.

C-Language Function Prototype

```
otma_send_async(
             otma_anchor_t  *anchor,                [in]
             otma_retrsn_t  *retrsn,                [out]
             ecb_t          *ecb,                   [out]

             tpipe_name_t   *tpipe_name,            [in]
             tran_name_t    *transaction,           [in]
             racf_uid_t     *username,              [in]
             racf_prf_t     *prfname,               [in]
             lterm_name_t   *lterm,                 [in]
             mod_name_t     *modname,               [in]
                 otma_user_t    *otma_userdata,         [in]

             char           *send_buffer,           [in]
             data_leng_t    *send_length,           [in]
             data_leng_t    *send_segment_list[],   [in]
             char           *error_message,         [out]
             void           *special_options);      [in]
```

Post Codes

0 Normal completion.

8 Invalid input.

12 Input failed.

16 Input cancelled (IMS is down or OTMA is stopped).

20 Error or information message from IMS.

Return Values (rc value)

The rc and reason are valid after ECB has been posted. For the complete description of each error, see Table 25 on page 119.

0 Normal completion.

8 No anchor/bad input.

12 Send failed.

16 Input cancelled (IMS is down or OTMA is stopped).

20 Error or information message from IMS.

Using otma_receive_async

Description

The `otma_receive_async` API is invoked to receive an IMS output message or an unsolicited message. The caller provides the buffer definitions to receive the IMS message. When the IMS message arrives, the ECB is posted.

Invocation

Called by the client in TCB mode.

Input

***anchor** Pointer to anchor word that was set up by `otma_open`.

***tpipe_name** Pointer to OTMA tpipe name field. This name must be different from the tpipe name specified for the `otma_create` and `otma_open` APIs.

receive_length
Length of buffer available to receive message.

Output

***ecb Event** Event control block to be posted when IMS receives the reply.

***error_message**
Address of the pointer to the error message field. It is provided by the user to receive error or informational messages from IMS. If the post code returns a 20, then this field will contain data.

***lterm** Pointer to `lterm` name field. Can be updated with `lterm` value that is returned by IMS.

***modname** Pointer to `MODname` name field. Can be updated with `MODname` value that is returned by IMS.

***otma_user_data**
Pointer to the OTMA user data. This 1022-byte field is optional. If the field is specified and IMS returns the OTMA user data, the data is passed back to the caller.

The OTMA user data received is either provided in the `otma_send_async` API or created by the IMS DRU exit DFSYDRU0.

***receive_buffer**
Pointer to buffer to receive reply message from IMS.

***received_length**
Field to receive length of data received to receive_buffer. Should be equal to the sum of the segment lengths.

***receive_segment_list**
An array of lengths of message segments received from IMS. The client must set the first element to indicate the maximum number of message segments that can be received. After all the segments are

received, the first array element indicates the actual number of segments received, and the rest of the array elements indicate the length of each segment received.

***retrsn** Pointer to return/reason code structure.

***special_options**
Pointer to an area codifying non-standard options. Currently, no special are options supported. Specify a NULL for this parameter.

C-Language Function Prototype

```
otma_receive_async(
            otma_anchor_t  *anchor,                 [in]
            otma_retrsn_t  *retrsn,                 [out]
            ecb_t          *ecb,                    [out]

            tpipe_name_t   *tpipe_name,             [in]
            lterm_name_t   *lterm,                  [out]
            mod_name_t     *modname,                [out]
                otma_user_t    *otma_userdata,          [out]

            char           *receive_buffer,         [out]
            data_leng_t    *receive_length,         [in]
            data_leng_t    *received_length,        [out]
            data_leng_t    *receive_segment_list[], [in/out]

            void           *special_options);       [in]
```

Post Codes

0 Normal completion.

12 Receive failed.

Return Values (rc value)

The rc and reason are valid after ECB has been posted. For the complete description of each error, see Table 25 on page 119.

0 Normal completion.

8 No anchor/bad session handle/segment too large.

12 Send failed.

Using otma_free

Description

The `otma_free` API is called to free an independent session created by `otma_alloc`.

Invocation

Called by the client in TCB mode.

Input

***anchor** Pointer to anchor word returned by `otma_open`.

***session_handle**
Pointer to session handle returned by `otma_alloc`.

Output

***retrsn** Pointer to return code structure.

***session_handle**
Pointer to session handle will be nulled by `otma_free`.

C-Language Function Prototype

```
otma_free(
   otma_anchor_t  *anchor,          [in]
   otma_retrsn_t  *retrsn,          [out]
   sess_handle_t  *session_handle); [in/out]
```

Return Values (rc value)

The rc and reason are valid after ECB has been posted. For the complete description of each error, see Table 25 on page 119.

0 Success.

4 Not allocated session.

8 Incorrect anchor.

Using otma_close

Description

The `otma_close` API is called to free storages for communication and to leave XCF group. This function may be called when communications are in flight or an open is processing. In these cases all relevant ECBs will be posted with a canceled post code.

Invocation

Called by the client in TCB mode.

Input

***anchor**
Pointer to anchor word returned by `otma_open`.

Output

***anchor**
Pointer to anchor word returned by `otma_open`.

***retrsn**
Pointer to return code.

C-Language Function Prototype

```
otma_close(
     otma_anchor_t  *anchor,  [in/out]
     otma_retrsn_t   *retrsn); [out]
```

Return Values (rc value)

The rc and reason are valid after ECB has been posted. For the complete description of each error, see Table 25 on page 119.

0 Success.

4 Null anchor.

8 Cannot leave XCF group.

Codes and Messages Used by OTMA C/I

This section describes the messages and codes that OTMA C/I uses.

In this Section:

OTMA Post Codes

The asynchronous nature of OTMA message delivery and communication state change requires that the program that uses the OTMA C/I use Event Signaling methods. The functions `otma_open` and `otma_send_receive` are asynchronous, the other APIs are not.

The `otma_open`, `otma_openx`, `otma_send_receive`, `otma_send_receivex`, `otma_send_async`, and `otma_receive_async` APIs each have an ECB parameter. This ECB is posted by the function or by an SRB that the function precipitates. The caller of `otma_open`, `otma_openx`, `otma_send_receive`, `otma_send_receivex`, `otma_send_async`, and `otma_receive_async` functions must check this ECB and wait for it to be posted before releasing or inspecting any of the output fields mentioned in the API, except for the return code in the return code structure. The return code from all six functions indicates failure to initiate communications with a non-zero value. The ECB has the same value in these cases.

The general meanings of the POST codes are as follows:

0 Data transfer or state change expected is completed.

4 Transient problem detected. Try again later.

8 User error.

12 System error.

16 Client or XCF has aborted the function.

20 Error from IMS.

OTMA Return Codes

The return code structure consists of a return code that indicates the status of a request.

0 Function completed normally.

4 Transient problem detected. Try again later.

8 User error.

12 System error.

16 Function cancelled.

20 Error from IMS.

The return code is part of a data structure used by all the API calls and is the post code found in any posted ECB. All numerical values are in decimals.

Four reason code fields are available.

Reason 1
Indicates what area in the OTMA client API reported the error. The value depends on the function that was called and is listed in Table 25. All numerical values are in decimals.

Reason 2, Reason 3
Provides additional information about the problem. All numerical values are in decimals.

Reason 4
Indicates the API that was issued.

A vector of reason codes describes the details of any failed or partial result. The meanings are specific to the function and are described in Table 25.

Table 25 describes OTMA C/I return codes and reason codes by function. The description of each function includes the return code (in decimal), four possible reason codes (in decimal), and a general description.

Table 25. OTMA C/I Return Codes and Reason Codes by Function

Function	Return Code (Decimal)	Reason 1 (Decimal)	Reason 2 (Decimal)	Reason 3 (Decimal)	Reason 4 (Decimal)	Description
CREATE	0				1	Normal completion.
	8	4			1	The user is not allowed to use OTMA C/I because the user is not permitted to use the RACF IMSXCF.OTMACI resource.
	8	20			1	OTMA C/I was used in OS/390 R2 or below. OTMA C/I should be used on OS/390 R3 or above.
	8	28			1	The session number that is specified is greater than the maximum number allowed (999).
	12	12	BPESVC return code		1	The OTMA C/I service call was rejected by the BPESVC service. See “BPE Codes” in *IMS Version 9: Messages and Codes, Volume 1* for more information on BPESVC return codes.
	12	800			1	Storage obtain failure.
OPEN and OPENX	0				2	Normal completion.
	0	1xx	xcf ixcjoin rc	xcf ixcjoin rsn	2	Member state changed. For more information, see *OS/390 MVS Programming: Sysplex Services Reference*.
	4	80	xcf ixcjoin rc	xcf ixcjoin rsn	2	IMS OTMA is not started, an invalid XCF group name was specified, or an invalid XCF member name was specified. Try again later. Server member is not active. For more information, see *OS/390 MVS Programming: Sysplex Services Reference.*

Table 25. OTMA C/I Return Codes and Reason Codes by Function (continued)

Function	Return Code (Decimal)	Reason 1 (Decimal)	Reason 2 (Decimal)	Reason 3 (Decimal)	Reason 4 (Decimal)	Description
	8	0	xcf ixcjoin rc	xcf ixcjoin rsn	2	Your member name is already active. For more information, see *OS/390 MVS Programming: Sysplex Services Reference.*
	8	4			2	The user is not allowed to use OTMA C/I because the user is not permitted to use the RACF IMSXCF.OTMACI resource.
	8	8			2	The user is not allowed to specify the input anchor. The input anchor should be returned from the `otma_create` API or a new anchor should be initialized with 0. If the input anchor is correct, ensure that the input group name, the input member name, and the input partner name are the same as the names specified in the `otma_create` API.
	8	12			2	Client bid is refused.
	8	16			2	Client bid is refused due to security failure.
	8	20			2	OTMA C/I was used in OS/390 R2 or below. OTMA C/I should be used on OS/390 R3 or above.
	8	28			2	The session number that is specified is greater than the maximum number allowed (999).
	8	112	xcf ixcjoin rc	xcf ixcjoin rsn	2	Member is in an unknown state. For more information, see *OS/390 MVS Programming: Sysplex Services Reference*.
	12	10x	xcf ixcjoin rc	xcf ixcjoin rsn	2	Join sysplex failed. For more information, see *OS/390 MVS Programming: Sysplex Services Reference*.
	12	11x	xcf ixcjoin rc	xcf ixcjoin rsn	2	Join local failed. For more information, see *OS/390 MVS Programming: Sysplex Services Reference*.
	12	12	BPESVC return code		2	The OTMA C/I service call was rejected by the BPESVC service. See “BPE Codes” in *IMS Version 9: Messages and Codes, Volume 1* for more information on BPESVC return codes.
	12	400	xcf ixcquery rc	xcf ixcquery rsn	2	Query sysplex failed. For more information, see *OS/390 MVS Programming: Sysplex Services Reference*.
	12	800	0	0	3	Getmain failure.
ALLOC	0	0	0	0	5	Normal.

Table 25. OTMA C/I Return Codes and Reason Codes by Function (continued)

Function	Return Code (Decimal)	Reason 1 (Decimal)	Reason 2 (Decimal)	Reason 3 (Decimal)	Reason 4 (Decimal)	Description
	4	12	8	0	5	The OTMA C/I cannot dynamically allocate a session because either: • The number of existing sessions is already the maximum allowed, or • C/I cannot obtain any free storage from subpool 230 for the session usage.
	8	4	0	0	5	Null anchor.
	8	16			5	Incorrect input anchor.
	8	20			5	A transaction name or command was entered. However, it was not left-justified.
	8	24	Caller's new state	Caller's old state	5	Caller changed the program state or key -- see reason codes 3 and 4. Program state and key should remain the same for all API calls.
	12	12	BPESVC return code		5	The OTMA C/I service call was rejected by the BPESVC service. See "BPE Codes" in *IMS Version 9: Messages and Codes, Volume 1* for more information on BPESVC return codes.
SEND RECEIVE and SEND RECEIVEX	0	0			7	Normal.
	0	4			7	Conversational.
	0	44	Maximum number of multi-segments specified in the receive segment list	Actual number of multi-segments sent from IMS	7	The first array element of the receive segment list specifies the maximum number of multi-segments that can be received from IMS. If IMS sends more multi-segments than the number specified in the receive segment list, the last element in the receive segment list will include the size of the rest of the multi-segments. All of the multi-segments will be stored in the receive buffer.
	0	48	Length of the receive buffer specified.	Length of the receive buffer needed to contain all of the output.	7	The size of the buffer is too short. The actual data returned is limited by the size of the receive buffer.
	8	4			7	No anchor.
	8	8			7	Bad session handle.
	8	12	Session state		7	Invalid session state.
	8	16			7	Incorrect input anchor.

Table 25. OTMA C/I Return Codes and Reason Codes by Function (continued)

Function	Return Code (Decimal)	Reason 1 (Decimal)	Reason 2 (Decimal)	Reason 3 (Decimal)	Reason 4 (Decimal)	Description
	8	24	Caller's new state	Caller's old state	7	Caller changed the program state or key -- see reason codes 3 and 4. Program state and key should remain the same for all API calls.
	8	32	Segment number	Maximum Size	7	Segment is too large.
	8	40			7	Buffer!=segments.
	8	44	The maximum number of multi-segments specified in the receive segment list.	Actual number of multi-segments sent from IMS.	7	The client specified SyncLevel1 in the special option parameter of the otma_alloc API. C/I rejected the IMS output because the size of the receive segment list is too small.
	8	48	Length of receive buffer specified.	Length of receive buffer needed to include all the output.	7	The size of the receive buffer is too short. Check the receive_buffer and receive_length parameter, and correct the application program to use the buffer size returned in reason code 3. This error occurs only when the client specifies SyncLevel1 in the special option parameter of the otma_alloc API.
	8	52			7	Either input Iterm or modname is NULL. The Iterm and modname fields are updated on output so they cannot be set to NULL on input. Leave these fields blank if they are not to be used.
	8	56	rrs ctxswc rc		7	Switch off failed. For more information, see *OS/390 MVS Programming: Resource Recovery*.
	8	60			7	Input Native token.
	8	64			7	Non-current token.
	8	68			7	The input send length is 0 or less than 0.
	8	80			7	Input rejected. An attempt was made to send a new input message for a non-conversational transaction on an existing session handle. You must free the previous session handle by issuing an OTMA_FREE and then issue an OTMA_ALLOC for the new transaction.
	12	8	Send return code.	Send return code.	7	Send failed.

Table 25. OTMA C/I Return Codes and Reason Codes by Function (continued)

Function	Return Code (Decimal)	Reason 1 (Decimal)	Reason 2 (Decimal)	Reason 3 (Decimal)	Reason 4 (Decimal)	Description
	12	12	BPESVC return code		7	The OTMA C/I service call was rejected by the BPESVC service. See “BPE Codes” in *IMS Version 9: Messages and Codes, Volume 1* for more information on BPESVC return codes.
	12	16	rrs ctxswc rc		7	Switch on failed. For more information, see *OS/390 MVS Programming: Resource Recovery*.
	16					Receive has been cancelled by IMS. Either IMS is down, or OTMA is stopped.
	20	NAK code	NAK reason			Either IMS rejected the input message, or a backout was performed.
SEND ASYNC	0	0			10	Normal.
	8	4			10	No anchor.
	8	16			10	Incorrect input anchor.
	8	20			10	Invalid Trancode.
	8	24	Caller’s new state	Caller’s old state	10	Caller changed the program state or key -- see reason codes 3 and 4. Program state and key should remain the same for all API calls.
	8	32	Segment number	Maximum size	10	Segment is too large.
	8	40			10	Buffer!=segments.
	8	56			10	Invalid send buffer length.
	8	60			10	Missing tpipe name.
	8	64			10	Invalid tpipe name.
	8	68			10	Invalid tpipe name. First four characters of name share the same tpipe prefix in otma_open or otma_openx.
	8	72			10	Invalid error message parameter specified.
	12	8	Send return code	Send return code	10	Send failed.
	12	12	BPESVC return code		10	The OTMA C/I service call was rejected by the BPESVC service. See “BPE Codes” in *IMS Version 9: Messages and Codes, Volume 1* for more information on BPESVC return codes.
	16					Send has been cancelled by IMS. Either IMS is down, or OTMA is stopped.
	20	NAK code	NAK reason			Either IMS rejected the input message, or a backout was performed.
RECEIVE ASYNC	0	0			11	Normal.

Table 25. OTMA C/I Return Codes and Reason Codes by Function (continued)

Function	Return Code (Decimal)	Reason 1 (Decimal)	Reason 2 (Decimal)	Reason 3 (Decimal)	Reason 4 (Decimal)	Description
	0	44	Maximum number of multi-segments specified in the receive segment list	Actual number of multi-segments sent from IMS	11	The first array element of the receive segment list specifies the maximum number of multi-segments that can be received from IMS. If IMS sends more multi-segments than the number specified in the receive segment list, the last element in the receive segment list will include the size of the rest of the multi-segments. All of the multi-segments will be stored in the receive buffer.
	4	80			11	The IMS OTMA function is not started.
	8	16			11	Incorrect input anchor.
	8	20			11	Invalid Trancode.
	8	24	Caller's new state	Caller's old state	11	Caller changed the program state or key -- see reason codes 3 and 4. Program state and key should remain the same for all API calls.
	8	48	Length of receive buffer specified.	Length of receive buffer needed to include all the output.	11	The size of the receive buffer is too short. Check the receive_buffer and receive_length parameters, and correct the application program to use the buffer size returned in reason code 3.
	8	56			11	Invalid send buffer length.
	8	60			11	Missing tpipe name.
	8	64			11	Invalid tpipe name.
	8	68			11	Invalid tpipe name. First four characters of name share the same tpipe prefix in otma_open or otma_openx.
	12	12	BPESVC return code		11	The OTMA C/I service call was rejected by the BPESVC service. See "BPE Codes" in *IMS Version 9: Messages and Codes, Volume 1* for more information on BPESVC return codes.
	16					Receive has been cancelled by IMS. Either IMS is down, or OTMA is stopped.
FREE	0	0			14	Normal.
	4	0			14	Not allocated.
	4	4			14	Quitting.
	8	4			14	Null anchor.
	8	8			14	Obsolete handle.
	8	16			14	Incorrect input anchor.
	8	24	Caller's new state	Caller's old state	14	Caller changed the program state or key -- see reason codes 3 and 4. Program state and key should remain the same for all API calls.

Table 25. OTMA C/I Return Codes and Reason Codes by Function (continued)

Function	Return Code (Decimal)	Reason 1 (Decimal)	Reason 2 (Decimal)	Reason 3 (Decimal)	Reason 4 (Decimal)	Description
	12	12	BPESVC return code		14	The OTMA C/I service call was rejected by the BPESVC service. See “BPE Codes” in *IMS Version 9: Messages and Codes, Volume 1* for more information on BPESVC return codes.
CLOSE	0	0	xcf ixcleav rc	xcf ixcleav rsn	15	Normal completion. For more information, see *OS/390 MVS Programming: Sysplex Services Reference*.
	8	4	0	0	15	Null anchor.
	8	16			15	Incorrect input anchor.
	12	8	xcf ixcleav rc	xcf ixcleav rsn	15	Non-zero return code from IXCLEAV. For more information, see *OS/390 MVS Programming: Sysplex Services Reference*.
	12	12			15	One of the OTMA C/I service routines abended. The abend could be caused by incorrect input parameters. If not, save the dump and contact your IBM Support Center for assistance.

OTMA Error Messages

DFS3908E ABEND *code* IN OTMA SVC INIT MODULE DFSYSVI0, PSW=*psw1psw2*

Explanation: An abend occurred while module DFSYSVI0 was in control. Module DFSYSVI0 is the module that initializes the OTMA Callable Services SVC service, and is typically run as a stand-alone job prior to running applications that use the OTMA Callable Services. DFSYSVI0 processing is protected by an internal ESTAE, which attempts to retry from the abend and clean up any global resources (such as common storage) that DFSYSVI0 obtained. Message DFS3908E is issued to alert the operator that an abend occurred.

In the message text:

code The abend code. For system abends, the format of *code* is S*xxx*, where *xxx* is the 3-digit abend code in hexadecimal. For user abends, the format of *code* is U*dddd*, where *dddd* is the 4-digit abend code in decimal.

psw1 The first word of the PSW at abend.

psw2 The second word of the PSW at abend.

System Action: The DFSYSVI0 ESTAE collects diagnostic data about the abend, and then resumes execution in a cleanup routine within DFSYSVI0. This routine attempts to release any global resources that DFSYSVI0 obtained as a part of its processing. DFSYSVI0 then issues message DFS3911E and returns to its caller. Typically, unless the abend occurred at the very end of DFSYSVI0 processing, the OTMA SVC routine is not initialized.

The first time that DFSYSVI0 abends, its ESTAE takes an SDUMP of the address space, and causes a record to be written to the SYS1.LOGREC data set to document the abend. If DFSYSVI0 abends a second time or more (within one execution), its ESTAE does not take another SDUMP. However, it does write another record to SYS1.LOGREC.

System Programmer Response: Save any dump, SYSLOG, and SYS1.LOGREC information and contact the IBM Support Center.

Module: DFSYSVI0

DFS3911E ERROR INITIALIZING OTMA SVC - *details*

Explanation: An error occurred in module DFSYSVI0. Module DFSYSVI0 is the module that initializes the OTMA Callable Services SVC service, and is typically run as a stand-alone job prior to running applications that use the OTMA Callable Services. When DFSYSVI0 cannot complete the OTMA callable services initialization, it issues message DFS3911E to indicate why initialization failed.

In the message text:

details A short summary of the reason why the OTMA Callable Services SVC initialization failed. *details* corresponds with the return code issued by the DFSYSVI0 module, and may be one of the following:

NOT EXECUTING IN PSW KEY 7

DFSYSVI0 was not given control in PSW key 7. DFSYSVI0 must run as an authorized program in PSW key 7. This is accomplished by adding DFSYSVI0 to the program properties table. Refer to the *IMS Version 9 Open Transaction Manager Access Guide and Reference* for instructions on how to add DFSYSVI0 to the program properties table.

ESTAE CREATE FAILED, RC=*rc*

DFSYSVI0 attempted to establish a z/OS recovery routine (ESTAE), but the create ESTAE call failed. *rc* is the return code from the z/OS ESTAE macro.

BPESVC INIT FAILED, RC=*rc*

DFSYSVI0 could not initialize the BPE SVC service. *rc* is the return code from the BPESVC initialization call.

BLDL FOR DFSYSVC0 FAILED, RC=*rc*

A z/OS BLDL call for module DFSYSVC0 failed. Ensure that DFSYSVC0 is included in the library from which you are running DFSYSVI0. *rc* is the return code from the z/OS BLDL macro call.

GET FOR STORAGE FAILED, RC=*rc*

DFSYSVI0 could not get storage required for the OTMA Callable Services SVC module. *rc* is the return code from the z/OS STORAGE macro call.

LOAD FOR DFSYSVC0 FAILED, RC=*rc*

A z/OS LOAD call for module DFSYSVC0 failed. *rc* is the return code from the z/OS LOAD macro call.

BPESVC REGISTRATION FAILED, RC=*rc*

Registration of the OTMA Callable Services SVC routine with BPESVC (BPE SVC services) failed. *rc* is the return code from the BPESVC REGISTER macro call.

ABEND OCCURRED DURING INITIALIZATION

An abend occurred during DFSYSVI0 processing. This message should be preceded by a DFS3908E message indicating the abend code and PSW, and by an SDUMP of the DFSYSVI0 job's address space.

System Action: Module DFSYSVI0 terminates. The OTMA Callable Services SVC is not initialized (or, if it was previously initialized, is not replaced).

System Programmer Response: For environmental errors (such as DFSYSVC0 not being in the same library as the one from which you are running DFSYSVI0), correct the error and re-run DFSYSVI0. For NOT EXECUTING IN PSW KEY 7 error, ensure that the library where DFSYSVC0 resides is APF authorized. For other problems contact the IBM Support Center.

Module: DFSYSVI0

OTMA C/I Sample Programs

The two sample C programs that are shown in this section are for display purposes only. The code for the two sample programs is available to licensed customers of IMS Version 6 and above in PTF number UQ23685.

Warranty and Distribution for OTMA C/I Sample Programs

The code is provided "AS IS." IBM makes no warranties, express or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose, regarding the function or performance of this code. IBM shall not be liable for any damages arising out of your use of the sample code, even if they have been advised of the possibility of such damages.

The sample code can be freely distributed, copied, altered, and incorporated into other software, provided that it bears the following Copyright notices and DISCLAIMER OF WARRANTIES intact.

```
(c) Copyright IBM Corp.
2000 All Rights Reserved. Licensed Materials - Property of IBM
DISCLAIMER OF WARRANTIES.
```

```
The following "enclosed" code is sample code created by IBM Corporation.
This sample code is not part of any standard or IBM product and is provided
to you solely for the purpose of assisting you in the development
of your applications.
```

OTMA C/I Sample Program #1: Synchronous Processing

The program shown below illustrates how OTMA C/I can be used for synchronous (one in-one out) processing. In this sample program, the `otma_send_receive` API is used to send and receive IMS data.

```
#pragma langlvl(extended)
/*********************************************************************/
/*                                                                   */
/* Callable Interface sample program using synchronous APIs          */
/*                                                                   */
/* Parameters:                                                       */
/*         Server Name                                               */
/*         Client Name                                               */
/*         User Name                                                 */
/*         Iterations                                                */
/*         Transaction                                               */
/*         User Group                                                */
/*         OTMA Data                                                 */
/*                                                                   */
/* Note:  The send buffer is sent as a file with a ddname of         */
/*        SENDBUFn in the invoking JCL.                              */
/*                                                                   */
/* Example:  //SENDBUF0 DD *,DLM=$$                                  */
/*           SEND OTMA TO SKS1                                       */
/*           $$                                                      */
/*                                                                   */
/* Note: COMPAR1 is the DDNAME of an input file used to compare      */
/*       actual output with expected output. '?' is used to delimit  */
/*       the compare string and '|' is used to ignore a char compare */
/*                                                                   */
/* Example:  //COMPAR0  DD *,DLM=$$                                  */
/*           SEND OTMA TO SKS1?                                      */
/*           $$                                                      */
/*                                                                   */
/*********************************************************************/

/*********************************************************************/
/*   Entry...                                                        */
/*                                                                   */
/*     This test program is callable from JCL                        */
/*                                                                   */
/* //NA1OTMA  JOB CLASS=A,MSGLEVEL=(1,1),MSGCLASS=H,REGION=2M       */
/* //*************************************************************** */
/* //* PARM=server_member_name tpipe_name client_member_name        */
/* //*      iterations command groupid OTMA_Data                     */
/* //MINISAMP EXEC PGM=NA1OTMA,                                      */
/* // PARM='TRAP(OFF)/IMS61CR1 IMSTESR G214992 1 /DISP groupid      */
/* //      OTMAData'                                                 */
/* //STEPLIB  DD DISP=SHR,DSN=OTMA.TEST.LOAD                         */
/* //SYSUDUMP DD SYSOUT=*                                            */
/* //STDOUT   DD SYSOUT=*                                            */
/* //STDERR   DD SYSOUT=*                                            */
/* //CEEDUMP  DD SYSOUT=*                                            */
/* //COMPAR1  DD *,DLM=$$                                            */
/* EXPECTED OUTPUT GOES HERE                                         */
/* $$                                                                */
/* //SENDBUF0 DD *,DLM=$$                                          */
/* SEND DATA GOES HERE                                               */
/* $$                                                                */
/*                                                                   */
/* Note: TRAP(OFF)/ Passes LE run-time option TRAP(OFF) which turns */
```

```
/*       off LE condition handling. To get a LE dump on abend set   */
/*       TRAP ON and provide a CEEDUMP DDNAME.                      */
/*                                                                  */
/* Note: COMPAR1 is the DDNAME of an input file used to compare     */
/*       actual output with expected output. '?' is used to delimit */
/*       the compare string and '|' is used to ignore a char compare*/
/*                                                                  */
/********************************************************************/

/********************************************************************/
/* An example for using the OTMA Client API in C lang.              */
/*   This program is broken into the following parts:               */
/*      Declarations for special support                            */
/*      Process invocation parameters                               */
/*      Setup for C signal handling                                 */
/*      Do XCF open processing and analysis                         */
/*      Do session allocate processing                              */
/*      Execute a command or transaction per invocation parm        */
/*      Do session free processing                                  */
/*      Do close                                                    */
/*      End                                                         */
/********************************************************************/

/********************************************************************/
/* API's for non-authorized OTMA caller                             */
/********************************************************************/
#include "dfsyc0.h"             /* Non-authorized OTMA API's        */
#include <stdlib.h>             /* Standard C Header file           */
#include <stddef.h>             /* Standard C Header file           */
#include <stdio.h>              /* Standard C Header file           */

/********************************************************************/
/* Internal functions                                               */
/********************************************************************/
int  memc(char *comp_buf, char *rec_buf1 );

/* macro to move string to blank filled left justified char field    */
#define splat(t,s) \
  {\
  memset((char*)&(t),' ',sizeof(t));\
  strncpy((char*)&(t), s ,strlen(s));}

/* standard math routines                                           */
#define   min(a,b)        ((a)<(b)?(a):(b))
#define   max(a,b)        ((a)>(b)?(a):(b))

main(int argc,char *argv[])
{

   /*         Following fields used by all Functions               */

 otma_anchor_t    anchor;         /*  Handle returned by create      */
                                  /*  and used by all others.        */
 otma_retrsn_t    retrsn;         /*  Return code returned by all.   */
 long int         retsave;        /*  Return code save area          */

   /*     Following fields used by several Functions      */

 sess_handle_t     sess_handle;   /* Handle returned by allocate     */
                                  /* used by send_receive and free.  */
 otma_grp_name_t   grp_name;      /* API XCF Group Member Name.      */
 otma_clt_name_t   clt_name;      /* API XCF Client Member Name.     */
 otma_srv_name_t   srv_name;      /* API XCF Server Member Name.     */
                                  /*   (IMS's XCF member name).      */
 racf_uid_t        userid;        /* Our z/OS logon ID.              */
 racf_prf_t        groupid;       /* RACF Group ID                   */
 otma_user_t       otma_data;     /* Otma Data                       */
```

```
 lterm_name_t  lterm;                  /* Lterm name                */
 mod_name_t    modname;                  /* ModName                */

 unsigned char error_message_text[120];/* IMS error msg field       */
                                       /* A place to receive any IMS      */
                                       /* DFS error messages.             */
 unsigned char *error_message = (unsigned char*)&error_message_text;
                                       /* a pointer to which is parameter */
                                       /* on send_receive.                */

 char         *tran;                   /* Transaction Name / IMS Command  */
 tran_name_t  tran_name;               /* Transaction Name / IMS Command  */

#define BUFFER_LEN  4096               /* set our buffer sizes           */
#define NUM_BUFFER  60
#define COM_BUFFER  80
#define GROUP_NAME  "HARRY"            /* Set XCF group name to join     */

 char compare_buf[NUM_BUFFER + 1]; /* Compare buffer                */
 int long      buffer_length = 0;
 int long      rec_buffer_len = BUFFER_LEN;
 char          rec_buf[BUFFER_LEN];
 long int      rec_data_len = 0;
 char          send_buf[BUFFER_LEN];
 char          temp_buf[NUM_BUFFER];

 context_t     context = {0x00000000000000000000000000000000};
                       /* This test is not distributed sync point.   */
                       /* Too complicated for here.                  */
                       /* Normally this is obtained from RRS         */

/*********************************************************************/
/*  The callable interface makes use of z/OS Event Control Blocks.   */
/*  Any language which call the interface must deal with this.       */
/*********************************************************************/

 unsigned long  *(ecb_list[2]);        /* z/OS pause stuff           */
 unsigned long **pecb_list;

 ecb_t         ecbOPEN   = 0L;     /* ecb to be posted by OTMA API   */
 ecb_t         ecbIO     = 0L;     /* ecb to be posted by OTMA API   */
 ecb_t         signal    = 0L;     /* ecb to be posted by C runtime  */

 ecb_t         temp_ecb  = 0L;     /* used by compare and swap       */
 ecb_t         reset_ecb = 0L;     /* used by compare and swap       */

/*********************************************************************/
/*    Local variables                                                */
/*********************************************************************/

 int           iterations;
 int           loop_count;
 int           compare_result;
 long int      retcode;

 signed long   sessions;        /* number of sessions to support     */
 tpipe_prfx_t  tpipe_prefix;       /* first part of tpipe NAME       */

 FILE * stream;
 int num;       /* number of characters read from stream */


/*********************************************************************/
/* To support test functions - names of parms                        */
/* Print the parms out for documentation                             */
/*********************************************************************/
```

```
 char * argdefs[8]={ "pgm name",          /* 1                    */
                     "server name",       /* 2                    */
                     "client name",       /* 3                    */
                     "userid     ",       /* 4                    */
                     "iterations ",       /* 5                    */
                     "transaction",       /* 6                    */
                     "group id   ",       /* 7                    */
                     "otma data  ",       /* 8                    */
                     };

/*********************************************************************/
/*  Declare an array of compare file ddnames to                      */
/*  compare actual output received with expected output.             */
/*********************************************************************/

 char * infiledd[4]={"DD:COMPAR0",        /* 1                   */
                     "DD:COMPAR1" ,       /* 2                   */
                     "DD:COMPAR2" ,       /* 3                   */
                     "DD:COMPAR3" ,       /* 4                   */
                     };

/*********************************************************************/
/*  Declare an array of send file ddnames to                         */
/*  send application data to OTMA.                                   */
/*********************************************************************/

 char * sndfiledd[4]= {"DD:SENDBUF0",        /* 1                   */
                       "DD:SENDBUF1" ,       /* 2                   */
                       "DD:SENDBUF2" ,       /* 3                   */
                       "DD:SENDBUF3" ,       /* 4                   */
                      };

/* ---------------------------------------------------------------- */
/* Anounce the startup of the test program.                         */
/* ---------------------------------------------------------------- */
   printf("Otmci01   Starting, version %s %s\n" ,__DATE__,__TIME__ );

/* ---------------------------------------------------------------- */
/* z/OS Pause Init - do this first, in case it fails bail out.      */
/* This sets up a C environment for signaling from the API.         */
/* ---------------------------------------------------------------- */

  ecb_list[0] = (unsigned long *) &(signal); /* post by C signal    */
  ecb_list[1] = (unsigned long *)            /* post by OTMA        */
                ((unsigned long)&(ecbOPEN) |
                 (unsigned long)0x80000000);/* end of list         */
  pecb_list = &ecb_list[0];                 /* pointer to list     */
                                            /* define callable I/F */

/*********************************************************************/
/* Begin Test Case...                                                */
/* Anounce the startup of the test program.                          */
/*********************************************************************/
   printf("OTMCI01 Run Date: %s Run Time: %s\n" ,__DATE__,__TIME__ );

/*********************************************************************/
/* Process parms/command line arguments.                             */
/*********************************************************************/

  /* First, print the parameters. */
  printf("Invocation parameters = \n");
   for (i=1 ; i<(min(8,argc));i++)
   {
     printf("%d   %s = ", i, argdefs[i]);
     printf("%s.\n", argv[i]);
   }
```

```
 if (argc>1)  splat( srv_name, argv[1])      /* XCF memname of IMS */
 else         splat( srv_name, "IMS61CR1");  /* hard coded default */
 if (argc>2)  splat( clt_name, argv[2])      /* Client name        */
 else         splat( clt_name, "XCFTEST"  ); /* hard coded default */
 if (argc>3)  splat( userid  , argv[3])      /* ID to use          */
 else         splat( userid  , "XCFTEST"  ); /* hard coded default */
if (argc>4)   iterations = atoi(argv[4]);    /* loop count         */
 else         iterations = 1;                /* hard coded default */
if (argc>5)   tran = argv[5];                /* Transaction/IMS CMD*/
 else         tran = "";                     /* hard coded default */
 if (argc>6)  splat( groupid, argv[6])       /* Group ID to use    */
 else         splat( groupid, "        " );  /* hard coded default */
 if (argc>7)  splat( otma_data, argv[7])     /* OTMA Data          */
 else         splat( otma_data, "" );        /* hard coded default */

 /* ------------------------------------------------------------*/
 /* Open the file with the ddname SENDBUF0 supplied in the     */
 /* JCL which invoked this C driver.  Then read the file into  */
 /* temp_buf.                                                  */
 /* ------------------------------------------------------------*/

 if (( stream = fopen("DD:SENDBUF0","rb")) != NULL )
 {
   num = fread( temp_buf, sizeof( char ), NUM_BUFFER, stream );
  printf("BUFF SIZE = %d.\n", num);
   if (num == NUM_BUFFER) {
     printf( "Number of characters read = %i\n", num );
     fclose( stream );
   }
   else {
     if ( ferror(stream) )
       printf( "Error reading DDNAME sendbuf0/n");
     else if ( feof(stream)) {
       printf( "EOF found\n" );
       printf( "Number of characters read %d\n", num );
       printf( "temp_buf = %.*s\n", num, temp_buf);
       fclose( stream );
     }
   }
 }
 else
   printf( "ERROR opening DDNAME sendbuf0/n" );

 /*  Initialize API parameters and buffers.                    */
 splat( grp_name,GROUP_NAME );              /* XCF Group Name     */
 splat( tpipe_prefix,"TPAS" );              /* Tpipe Prefix Name  */
 splat( tran_name,tran );                   /* do scan here       */
 strncat(send_buf, temp_buf,num);   /* Copy temp_buf into send_buf */
 buffer_length = strlen(send_buf);  /* Set send buffer length     */

/*************************************************************/
/* Example of setting up parms to Open the XCF Link          */
/*************************************************************/

  retrsn.ret    = -1;
  retrsn.rsn[0] = -1;
  retrsn.rsn[1] = -1;
  retrsn.rsn[2] = -1;
  retrsn.rsn[3] = -1;

  sessions      = 10;    /* OTMA supports multiple parallel    */
                         /* sessions (TPIPES) How many do you want?*/

/*************************************************************/
/*BEGIN:                                                     */
/* We have a CREATE function to set up storage and           */
```

```
/* an OPEN function to start the protocol.                          */
/* If you don't need to customize the environment you can start    */
/* with the OPEN function, the CREATE will be done by OPEN.        */
/*******************************************************************/

  printf("-\n");
  otma_create(&anchor,      /* (out) ptr to addr to receive ancho*/
         &retrsn,           /* (out) return code                */
         (ecb_t *) &ecbOPEN,/* not posted by create but stored   */

         &grp_name,         /* (in) ptr to valid groupname       */
         &clt_name,         /* (in) Our member name              */
         &srv_name,         /* (in) Our server name              */

         &sessions,         /* (in) number of sessions to support*/
         &tpipe_prefix      /* (in) first part of tpipe name     */
         );

   printf("OTMA_CREATE issued. ret = %d rsn = %.8x,%.8x,%.8x,%.8x\n"
          "  anchor is at %.8x.\n",
          retrsn.ret,
          retrsn.rsn[0],
          retrsn.rsn[1],
          retrsn.rsn[2],
          retrsn.rsn[3],
          anchor);


 printf("-\n");

/*******************************************************************/
/*  Connect to IMS                                                 */
/*******************************************************************/

 otma_open(&anchor,        /* out ptr to addr to receive anchor  */
         &retrsn,          /* out return code                    */
         (ecb_t *)&ecbOPEN, /* out posted by open if failure     */
                           /*     else posted by exit pgm        */
         &grp_name,        /* in ptr to valid XCF groupname      */
         &clt_name,        /* in Our member name                 */
         &srv_name,        /* in Our server name                 */

         &sessions,        /* in number of sessions to support  */
         &tpipe_prefix     /* in first part of tpipe name       */
         );

   printf("OTMA_OPEN issued. ret = %.8x rsn = %.8x,%.8x,%.8x,%.8x\n"
          " Waiting for ecb at %.8x.=%.8x.\n",
          retrsn.ret,
          retrsn.rsn[1],
          retrsn.rsn[2],
          retrsn.rsn[3],
          ecb_list[1],
          *ecb_list[1]
          );

  printf("-\n");

/* ---------------------------------------------------------------- */
/* Here we wait for Open to signal complete                         */
/* ---------------------------------------------------------------- */
    DFSYCWAT(ecb_list[1]);   /* WAIT on ecb                         */

     printf("OPEN_OTMA done. ret = %.8x rsn = %.8x,%.8x,%.8x,%.8x \n"
            "\nEcb at %.8x.= %.8x.\n",
            retrsn.ret,
            retrsn.rsn[0],
```

```
          retrsn.rsn[1],
          retrsn.rsn[2],
          retrsn.rsn[3],
          ecb_list[1], *ecb_list[1]
          );

   printf("Local Area Anchor at %8.8X = %8.8X\n",
               &anchor, anchor);

   printf("-\n");

/* ------------------------------------------------------------*/
/* The post code from open indicates success or failure       */
/* ------------------------------------------------------------*/
   if (0!=(0x00ffffff & ecbOPEN))
     {
       printf("OPEN_OTMA ecb is posted failure.\n");
       return(retrsn.rsn[0]);
     }

/* ------------------------------------------------------------*/
/* Set userid to blanks if userid = bobdavis                  */
/* ------------------------------------------------------------*/

 printf(" Trans  = %.8s,\n  ", tran_name );
 printf(" Userid = %.8s,\n  ", userid );
 printf("Groupid = %.8s,\n  ", groupid );

/*************************************************************/
/* Like CREATE the ALLOC function just creates control blocks */
/* and stores data in them. Other functions may be invented   */
/* to modify these structures before the command-of-execution,*/
/* SEND_RECEIVE is issued.                                    */
/*************************************************************/

 otma_alloc(
            &anchor,               /* in   ptr to global word   */
            &retrsn,               /* out  rc,reason(1-4)       */

            &sess_handle,          /* out  session id           */
            NULL,                  /* in   default overrides    */

            &tran_name,            /* in   IMS tp name or cmd   */
            &userid,               /* in   RACFid  or blanks    */
            &groupid               /* in   RACF group id or blnk*/
  );

  printf("OTMA_ALLOC done. ret = %.8x rsn = %.8x,%.8x,%.8x,%.8x\n",
         retrsn.ret,
         retrsn.rsn[0],
         retrsn.rsn[1],
         retrsn.rsn[2],
         retrsn.rsn[3]
         );

/*************************************************************/
/*   Even if ALLOC fails we go on here just to prove the      */
/*   API will reject the call.                                */
/*************************************************************/

/*************************************************************/
/* This is the call that sends the data and prepares to       */
/* receive the answer from IMS.                               */
/*                                                            */
/* This test program can iterate with multiple calls here.    */
/*************************************************************/
```

```
 /* ___Send message wait for reply_____________________        */
for (loop_count = 0 ; loop_count<iterations ; loop_count++)
{
 /* ___Change the environment to wait for ecbIO                */
 ecbIO = 0;                             /* clear ecb for reuse  */
 ecb_list[1] = (unsigned long *)          /* posted by OTMA     */
                 ((unsigned long)&(ecbIO) |
                  (unsigned long)0x80000000);  /* end of list */

  if (loop_count != 0)
   {

    /* ------------------------------------------------------------*/
    /* If looping more than once open the next file to send       */
    /* and read it into the send_buf.                              */
    /* ------------------------------------------------------------*/

    if (( stream = fopen(sndfiledd[loop_count],"rb")) != NULL )
     {
      num = fread( temp_buf, sizeof( char ), NUM_BUFFER, stream );
      printf("BUFF SIZE = %d.\n", num);
      if (num == NUM_BUFFER) {
        fclose( stream );
      }
      else {
       if ( ferror(stream) )
        printf( "Error opening file  %s\n",sndfiledd[loop_count]);
        else if ( feof(stream)) {
          printf( "EOF found\n" );
          printf( "Number of characters read %d\n", num );
          printf( "temp_buf = %.*s\n", num, temp_buf);
          fclose( stream );
        }
      }
     }
    else
     printf( "Error opening file %s\n", sndfiledd[loop_count]);
    /*  Initialize send and receiving buffers.       */
    memset(rec_buf ,0, sizeof(rec_buf));
    memset(send_buf ,0, sizeof(send_buf));
    strcat(send_buf, temp_buf );
    strcat(send_buf, " " );
    buffer_length = strlen(send_buf);
    printf("%s\n",send_buf);
    printf ("buffer length = %d\n", buffer_length);
   }  /* end if loop_count != 0    */

  /* Print otma_send_receive parms and start of API  */
  memset(error_message_text ,0, sizeof(error_message_text));
  printf("Send buf at %.8x.\n", &send_buf);
  printf("Send buf = %s.\n", send_buf);
  printf("Receive buf at %.8x.\n", &rec_buf);
  printf("Lterm = %.8s.\n", lterm );
  printf("Modname = %.8s.\n", modname );

  printf("-\n");
  otma_send_receivex(
                  &anchor,          /* (in)  anchor block         */
                  &retrsn,          /* (out) return status        */
                  &ecbIO,           /* (in)  ecb address          */

                  &sess_handle,     /* (in)  session handle       */
                  &lterm,           /* (in/out)  logical terminal */
                  &modname,         /* (in/out)  module name      */

 (unsigned char *) &send_buf,       /* (in)  send buffer          */
                  &buffer_length,   /* (in)  size of send buffer  */
```

```
                    0,                /* (in)  send_segment_list    */

  (unsigned char *) &rec_buf,         /* (in)  receive buffer       */
                    &rec_buffer_len,  /* (in)  size of buffer       */
                    &rec_data_len,    /* (out) received data length */
                    0,                /* (in/out) receive seg list  */

                    &context,         /* (in)  context id           */
                    &error_message,   /* (out) ims message          */
                    &otma_data);      /* (in)  Otma Data            */

     printf("OTMA_SEND done. ret = %.8x rsn = %.8x,%.8x,%.8x,%.8x\n",
             retrsn.ret,
             retrsn.rsn[0],
             retrsn.rsn[1],
             retrsn.rsn[2],
             retrsn.rsn[3]);

/* ---------------------------------------------------------------- */
/* Here we wait for receive to signal complete                      */
/* An application can go do other thing while IMS is processing and */
/* while the XCF scheduled SRBs are returning data to the caller's  */
/* buffers. DO NOT DEALLOCATE THE BUFERS WHILE THIS IS GOING ON!    */
/* None of the output areas of the SEND_RECIEVE can be freed until  */
/* the ECB is posted complete.                                      */
/* ---------------------------------------------------------------- */

    DFSYCWAT(ecb_list[1]);       /* WAIT on ecb                   */

    retsave = retrsn.ret;        /* Save Receive return code      */

     printf("OTMA_RECEIVE done. ret = %.8x rsn = %.8x,%.8x,%.8x,%.8x\n"
             "\nEcb at %.8x.= %.8x.\n",
             retrsn.ret,
             retrsn.rsn[0],
             retrsn.rsn[1],
             retrsn.rsn[2],
             retrsn.rsn[3],
             ecb_list[1],
             *ecb_list[1]
             );

     if (retrsn.ret != 0)
     {

      /* ___Error path Free allocated session _____________________ */
      printf("-error path retrsn.ret=%d\n",retrsn.ret);
      printf("-\n");
      printf( "Error message = %s\n", error_message );
      otma_free(
                     & anchor,        /* (out)  ptr to global word   */
                     & retrsn,        /* (out)  rc,reason (1-4)      */
                     & sess_handle    /* (in)   unique path id       */
               );

      printf("OTMA_FREE done. ret = %.8x rsn = %.8x,%.8x,%.8x,%.8x \n",
             retrsn.ret,
             retrsn.rsn[0],
             retrsn.rsn[1],
             retrsn.rsn[2],
             retrsn.rsn[3]
             );

      /* ___Sever IMS connection _________________________________       */
      printf("-\n");
      otma_close(
                     & anchor,        /* (in,out) tr to otma anchor */
```

```
                    & retrsn         /* (out) rc,reason (1-4)      */
             );

      printf("OTMA_CLOSE done. ret = %.8x rsn = %.8x,%.8x,%.8x,%.8x\n",
             retrsn.ret,
             retrsn.rsn[0],
             retrsn.rsn[1],
             retrsn.rsn[2],
             retrsn.rsn[3]
             );

      return (retsave);      /* EXIT with receive API return code */
     }

    /* ------------------------------------------------------------*/
    /*  If SEND_RECEIVE worked ..                                  */
    /* ------------------------------------------------------------*/


    /* ------------------------------------------------------------*/
    /*  Open the compare file containing the expected output      */
    /*  of the receive buffer.  Compare the expected output       */
    /*  with the actual output and return the result.             */
    /* ------------------------------------------------------------*/

   rec_buf[0] = ' ';         /* Remove possible NL ie x'15' */
   printf( "infiledd = %s\n", infiledd[loop_count] );

   if (( stream = fopen(infiledd[loop_count],"rb")) != NULL )
   {
     num = fread( compare_buf, sizeof( char ), COM_BUFFER, stream );
     if (num == COM_BUFFER) {  /* fread success */
       printf( "compare_buf = %s\n", compare_buf );
       printf( "    rec_buf = %s\n", rec_buf );
       fclose( stream );
       compare_result = memc( compare_buf, rec_buf );
       printf( "compare_result = %i\n",compare_result);
       if (compare_result != 0)
         return(compare_result);        /* Exit if NO COMPARE    */
     }
     else {  /* fread() failed */
       if ( ferror(stream) )         /* possibility 1 */
         printf( "Error reading file %s\n", infiledd[loop_count]);
       else if ( feof(stream)) {     /* possibility 2 */
         printf( "EOF found\n" );
         printf( "Number of characters read %d\n", num );
         printf( "compare_buf = %.*s\n", num, compare_buf);
       }
     }
   }
   else
     printf( "Error opening file %s\n", infiledd[loop_count]);
 }      /* end of loop */

   /*****************************************************************/
   /* Once a message is sent to IMS and the answer received it      */
   /* is usual to release the TPIPE for use by other transactions.  */
   /* For conversational trans an application would keep using      */
   /* the handle to continue a conversational transaction with IMS. */
   /* The Transaction name is specified in the ALLOC and it is      */
   /* intended that a FREE be done at the end of each transaction   */
   /* and a new ALLOC be done for the next one. This is not         */
   /* expensive.                                                    */
   /*****************************************************************/

     printf("-\n");
     otma_free(
```

```
                    & anchor,        /* (out)  ptr to global word   */
                    & retrsn,        /* (out)  rc,reason (1-4)      */
                    & sess_handle    /* (in)   unique path id       */
             );

     printf("OTMA_FREE done. ret = %.8x rsn = %.8x,%.8x,%.8x,%.8x \n",
            retrsn.ret,
            retrsn.rsn[0],
            retrsn.rsn[1],
            retrsn.rsn[2],
            retrsn.rsn[3]
            );

    printf("-\n");

   /*                                                               */
   /* Finally, CLOSE severs the connection with IMS and frees the   */
   /* Storage used by the OTMA API.                                 */
   /*  This will be done at job-step termination but its untidy.    */
   /*                                                               */

    otma_close(
                    & anchor,        /* (in,out) ptr to otma anchor */
                    & retrsn         /* (out)  rc,reason (1-4)      */
              );
    printf("OTMA_CLOSE done. ret = %.8x rsn = %.8x,%.8x,%.8x,%.8x \n",
            retrsn.ret,
            retrsn.rsn[0],
            retrsn.rsn[1],
            retrsn.rsn[2],
            retrsn.rsn[3]
            );

    return (compare_result);          /* Retern return code         */
} /* end of main */

/*====================================================================*/
/*      Subroutine to compare expected results(compare_buf)           */
/*      with actual results(err_msg) the "|" is used to signify       */
/*      an ignore compare and "?" is used to mark the end of string.  */
/* Note: Compare starts using an index i=1 ie. the 2nd character      */
/*       because the 1st character was blanked out. ( NL x'15' )      */
/*====================================================================*/


int memc(char *comp_buf, char *rec_buf1)
{

 int j;
 int i;

  j = 0;

  for (i=1;
      ( (j==0) && (comp_buf[i] != '?') );
       i++ )
    {
      if( comp_buf[i] != '|' )                  /* Ignore compare  */
      {
        if( comp_buf[i] != rec_buf1[i])         /* compare ok ?    */
        {
          j++;                                  /* No              */
          printf( "MISCOMPARE !!!   \n" );
          printf( "comp_buf[%d] = %c\n", i, comp_buf[i] );
          printf( "rec_buf1[%d] = %c\n", i, rec_buf1[i] );
        }
        else
```

```
          ;
      }
      else
        ;                                        /* Else null        */
    }

 return (j);
}
```

OTMA C/I Sample Program #2: Asynchronous Processing

The following program illustrates how OTMA C/I can be used for asynchronous (unsolicited) processing. In this sample program, one `otma_send_asynch` and one `otma_receive_asynch` call are performed in a loop.

Recommendation: If you will be using synchronous (one in-one out) processing exclusively, use the `otma_send_receive` API. The `otma_send_receive` API provides the most efficient means of synchronous processing.

```
#pragma langlvl(extended)

/*********************************************************************/
/*
/* Callable Interface sample program using asynchronous APIs         */
/*
/* Parameters:                                                       */
/*         Server Name                                                                */
/*         Client Name                                               */
/*         Transaction
/*         User Name
/*         User Group
/*         Lterm
/*         Mod Name
/*         OTMA Data
/*         Iterations
/*                                                                   */
/* Note:  The send buffer is sent as a file with a ddname of         */
/*        SENDBUFn in the invoking JCL.                              */
/*                                                                   */
/* Example:  //SENDBUF0 DD *,DLM=$$                                  */
/*           SEND OTMA TO SKS1                                       */
/*           $$                                                      */
/*                                                                   */
/* Note: COMPAR1 is the DDNAME of an input file used to compare      */
/*       actual output with expected output. '?' is used to delimit  */
/*       the compare string and '|' is used to ignore a char compare */
/*                                                                   */
/* Example:  //COMPAR0  DD *,DLM=$$                                  */
/*           SEND OTMA TO SKS1?                                      */
/*           $$                                                      */
/*                                                                   */
/* Note: TPIPBUFn is the DDNAME of an input file used to specify     */
/*       the tpipe name to be used for each iteration.               */
/*                                                                   */
/* Example:  //TPIPEBUF0  DD *,DLM=$$                                */
/*           TPIPE001                                                */
/*           $$                                                      */
/*                                                                   */
/*********************************************************************/

/*********************************************************************/
/*   Entry...                                                       */
/*                                                                  */
/*     This test program is callable from JCL                       */
/*                                                                  */
/* //NA1OTMA  JOB CLASS=A,MSGLEVEL=(1,1),MSGCLASS=H,REGION=2M       */
```

```
/* //************************************************************ */
/* //* PARM=server_member_name client_member_name transaction       */
/* //*      user_name group_name lterm_name ModName OTMA_Data       */
/* //*      iterations                                              */
/* //************************************************************ */
/* //MINISAMP EXEC PGM=NA1OTMA,                                     */
/* // PARM='TRAP(OFF)/IMS61CR1 IMSTESR G214992 /DISP user01 groupid */
/* //       Lterm ModName OTMAData 1'                                                */
/* //STEPLIB  DD DISP=SHR,DSN=OTMA.TEST.LOAD                        */
/* //SYSUDUMP DD SYSOUT=*                                           */
/* //STDOUT   DD SYSOUT=*                                           */
/* //STDERR   DD SYSOUT=*                                           */
/* //CEEDUMP  DD SYSOUT=*                                           */
/* //COMPAR1  DD *,DLM=$$                                           */
/* EXPECTED OUTPUT GOES HERE                                       */
/* $$                                                              */
/* //SENDBUF0 DD *,DLM=$$                                          */
/* SEND DATA GOES HERE                                             */
/* $$                                                              */
/* //TPIPBUF0 DD *,DLM=$$                                          */
/* TPIPE NAME GOES HERE                                            */
/* $$                                                              */
/*                                                                 */
/* Note: TRAP(OFF)/ Passes LE run-time option TRAP(OFF) which turns */
/*       off LE condition handling. To get a LE dump on abend set   */
/*       TRAP ON and provide a CEEDUMP DDNAME.                      */
/*                                                                 */
/* Note: COMPAR1 is the DDNAME of an input file used to compare     */
/*       actual output with expected output. '?' is used to delimit */
/*       the compare string and '|' is used to ignore a char compare */
/*                                                                 */
/*******************************************************************/

/*******************************************************************/
/* An example for using the OTMA Client API in C lang.              */
/*   This program is broken into the following parts:               */
/*      Declarations for special support                            */
/*      Process invocation parameters                               */
/*      Setup for C signal handling                                 */
/*      Do XCF open processing and analysis                         */
/*      Execute an API to send data per invocation parm             */
/*      Execute an API to receive data per invocation parm          */
/*      Do close                                                    */
/*      End                                                         */
/*******************************************************************/

/*******************************************************************/
/* Header Definitions.                                             */
/*******************************************************************/
#include "dfsyc0.h"             /* Non-authorized OTMA API's       */
#include <stdlib.h>             /* Standard C Header file          */
#include <stddef.h>             /* Standard C Header file          */
#include <stdio.h>              /* Standard C Header file          */

/*******************************************************************/
/* Internal functions                                              */
/*******************************************************************/
/* memory comparison macro.     */
int  memc(char *comp_buf, char *rec_buf1 );

/* macro to move string to blank filled left justified char field   */
#define splat(t,s) \
  {\
  memset((char*)&(t),' ',sizeof(t));\
  strncpy((char*)&(t), s ,strlen(s));}

/* standard math routines                                          */
```

```
#define   min(a,b)        ((a)<(b)?(a):(b))
#define   max(a,b)        ((a)>(b)?(a):(b))

/*********************************************************************/
/*                                                                   */
/*     This OTMA C/I Program                                         */
/*                                                                   */
/* Note: TRAP(OFF)/ Passes LE run-time option TRAP(OFF) which turns  */
/*       off LE condition handling. To get a LE dump on abend set    */
/*       TRAP ON and provide a CEEDUMP DDNAME.                       */
/*                                                                   */
/* Note: COMPAR1 is the DDNAME of an input file used to compare      */
/*       actual output with expected output. '?' is used to delimit  */
/*       the compare string and '|' is used to ignore a char compare  */
/*                                                                   */
/*********************************************************************/
main(int argc,char *argv[])
{

/*********************************************************************/
/*  Fields used by OTMA C/I APIs.                                    */
/*********************************************************************/

   /* The following fields used by all the OTMA C/I API's.           */

 otma_anchor_t    anchor;         /* Handle returned by create       */
                                  /* and used by all others.         */
 otma_retrsn_t    retrsn;         /* Return code returned by all.    */

   /* The following fields are used by the otma_create and                */
   /* otma_open API's.                                                    */

 otma_grp_name_t   grp_name;      /* API XCF Group Member Name.      */
 otma_clt_name_t   clt_name;      /* API XCF Client Member Name.     */
 otma_srv_name_t   srv_name;      /* API XCF Server Member Name.     */
                                  /*   (IMS's XCF member name).      */
 signed long       sessions;      /* number of sessions to support   */
 tpipe_prfx_t      tpipe_prefix;  /* first part of tpipe NAME        */
   /* The following fields are used by otma_send_async API.          */

 tpipe_name_t      tpipe;         /* User Tpipe Name.                */
 tran_name_t       trans;         /* IMS Trancode or CMD.            */
 racf_uid_t        user_name;     /* RACF UserID.                    */
 racf_prf_t        user_prf;      /* RACF Groupname.                 */
 lterm_name_t      lterm;         /* Input Lterm.                    */
 mod_name_t        modname;       /* Input Modname.                  */
 otma_user_t       otma_data;     /* OTMA Userdata.                  */
 char             send_buf[BUFFER_LEN];
int long          buffer_length = 0;     /* Send Buffer length.        */
unsigned char error_message_text[120]; /*  IMS error msg field -     */
                                    /* A place to receive any IMS      */
                                  /* DFS error messages.             */
 unsigned char *error_message = (unsigned char*)&error_message_text;
                                  /* a pointer to which is parameter */
                                  /* on send_receive.                */
 otma_profile2_t   send_options;  /* Send Special Options.           */

   /* The following fields are used by otma_receive_async API.       */

 lterm_name_t      rec_lterm;     /* Output Lterm.                   */
 mod_name_t        rec_modname;   /* Output Modname.                 */
 otma_user_t       rec_otma_data; /* OTMA Userdata.                  */
 char          rec_buf[BUFFER_LEN];
 int long      rec_buffer_len = BUFFER_LEN;
 long int      rec_data_len = 0;
 otma_profile3_t   rec_options;   /* Receive Special Options.        */
```

```
/*********************************************************************/
/*  The callable interface makes use of z/OS Event Control Blocks.   */
/*  Any language which call the interface must deal with this.       */
/*********************************************************************/

 unsigned long  *(ecb_list[2]);        /* z/OS pause ecb list        */
 unsigned long **pecb_list;

 ecb_t         ecbOPEN   = 0L;     /* ecb to be posted by OTMA API   */
 ecb_t         ecbIO     = 0L;     /* ecb to be posted by OTMA API   */
 ecb_t         signal    = 0L;     /* ecb to be posted by C runtime  */

 ecb_list[0] = (unsigned long *) &(signal); /* post by C signal      */
 ecb_list[1] = (unsigned long *)            /* post by OTMA          */
               ((unsigned long)&(ecbOPEN) |
               (unsigned long)0x80000000);  /* end of list           */
 pecb_list   = &ecb_list[0];                /* pointer to list       */
                                            /* define callable I/F   */

/*********************************************************************/
/*  Local Variables                                                  */
/*********************************************************************/

 long int         retsave;        /* Return code save area           */
int           iterations;     /* Number of iterations to use     */
int           loop_count;     /* Number of iterations used       */
int            compare_result; /* Return Code result of the      */
                               /* comparison for buffers.         */

/*********************************************************************/
/*  Local Constants                                                  */
/*********************************************************************/

 #define BUFFER_LEN  4096          /* Set our buffer sizes          */
 #define NUM_BUFFER  80            /* Set the number of buffers     */
 #define GROUP_NAME  "HARRY"       /* Set XCF group name to join    */
 char    temp_buf[NUM_BUFFER];     /* Swapping buffer               */
 char    compare_buf[NUM_BUFFER + 1]; /* Compare buffer             */
 FILE * stream;
 int num;       /* number of characters read from stream */

/*********************************************************************/
/* To support test functions - names of parms in order to pring   */
/* the parms out for documentation.                                  */
/*********************************************************************/

char * argdefs[10]={"Program Name",  /* 1                  */
                    "Server Name",   /* 2                   */
                    "Client Name",   /* 3                   */
                    "Transaction",   /* 4                   */
                    "User Name  ",   /* 5                   */
                    "User Group ",   /* 6                   */
                    "Lterm      ",   /* 7                   */
                    "Mod Name   ",   /* 8                   */
                    "OTMA Data  ",   /* 9                   */
                    "Iterations ",   /* 10                  */
                    };

/*********************************************************************/
/*  Declare an array of compare file ddnames to                     */
/*  compare actual output received with expected output.           */
/*********************************************************************/

 char * infiledd[4]={"DD:COMPAR0",        /* 1                */
                       "DD:COMPAR1" ,        /* 2                */
                    "DD:COMPAR2" ,       /* 3                 */
                       "DD:COMPAR3" ,         /* 4                  */
```

```
                    };

/*********************************************************************/
/*  Declare an array of send file ddnames to                         */
/*  send application data to OTMA.                                   */
/*********************************************************************/

 char * sndfiledd[4]= {"DD:SENDBUF0",       /* 1                    */
                       "DD:SENDBUF1" ,      /* 2                    */
                       "DD:SENDBUF2" ,      /* 3                    */
                          "DD:SENDBUF3" ,       /* 4                    */
                      };


/*********************************************************************/
/*  Declare an array of tpipe names ddnames for the                  */
/*  otma_send_async API.                                             */
/*********************************************************************/

 char * tpipefiledd[4]= {"DD:TPIPBUF0",    /* 1                    */
                         "DD:TPIPBUF1" ,   /* 2                    */
                         "DD:TPIPBUF2" ,   /* 3                    */
                         "DD:TPIPBUF3" ,   /* 4                    */
                      };

/*********************************************************************/
/* Begin Test Case...                                                */
/* Anounce the startup of the test program.                          */
/*********************************************************************/
   printf("OTMCI02 Run Date: %s Run Time: %s\n" ,__DATE__,__TIME__ );

/*********************************************************************/
/* Process parms/command line arguments.                             */
/*
/* Note:  If not a parameter is not used, then "NONE" is used in   */
/*        its place.                                                                                                  */
/*
/*********************************************************************/

  /* First, print the parameters. */
  printf("Invocation parameters = \n");
   for (i=1 ; i<(min(11,argc));i++)
   {
     printf("%d   %s = ", i, argdefs[i]);
     printf("%s.\n", argv[i]);
   }

   printf("\n");

   if (argc>1 && strcmp(argv[1],"NONE") != 0)
     splat( srv_name, argv[1])             /* Server Name.       */
   else
     splat( srv_name, "IMS61CR1");         /* Hard coded default */
   if (argc>2 && strcmp(argv[2],"NONE") != 0)
     splat( clt_name, argv[2])             /* Client name        */
   else
     splat( clt_name, "XCFTEST"  );        /* Hard coded default */
   if (argc>3 && strcmp(argv[3],"NONE") != 0)
     splat( trans, argv[3])                /* IMS Tran/Cmd to use*/
   else
     splat( trans, "");                    /* Hard coded default */
   if (argc>4 && strcmp(argv[4],"NONE") != 0)
     splat( user_name, argv[4])            /* RACF Username      */
   else
     splat( user_name, "");                /* Hard coded default */
   if (argc>5 && strcmp(argv[5],"NONE") != 0)
     splat( user_prf, argv[5])             /* RACF Group ID      */
```

```
else
  splat( user_prf, "" );                /* Hard coded default */
if (argc>6 && strcmp(argv[6],"NONE") != 0)
  splat( lterm , argv[6])               /* Lterm to use       */
else
  splat( lterm , "" );                  /* Hard coded default */
if (argc>7 && strcmp(argv[7],"NONE") != 0)
  splat( modname , argv[7])             /* ModName to use     */
else
  splat( modname , "" );                /* Hard coded default */
if (argc>8 && strcmp(argv[8],"NONE") != 0)
  splat( otma_data, argv[8])            /* OTMAData to use    */
else
  splat( otma_data, "" );               /* Hard coded default */
if (argc>9 && strcmp(argv[9],"NONE") != 0)
  iterations = atoi(argv[9]);           /* Loop count         */
else
  iterations = 1;                       /* Hard coded default */

/* ------------------------------------------------------------*/
/* Open the file with the ddname SENDBUF0 supplied in the     */
/* JCL which invoked this C driver.  Then read the file into  */
/* temp_buf.                                                  */
/* ------------------------------------------------------------*/

if (( stream = fopen("DD:SENDBUF0","rb")) != NULL )
{
  num = fread( temp_buf, sizeof( char ), NUM_BUFFER, stream );
  if (num == NUM_BUFFER) {
    printf( "Number of characters read = %i\n", num );
    fclose( stream );
  }
  else {
    if ( ferror(stream) )
      printf( "Error reading DDNAME sendbuf0/n");
    else if ( feof(stream)) {
      printf( "EOF found\n" );
      printf( "Number of characters read %d\n", num );
      printf( "temp_buf = %.*s\n", num, temp_buf);
      fclose( stream );
    }
  }
}
else
  printf( "ERROR opening DDNAME sendbuf0/n" );

/*------------------------------------------------------------*/
/* Initialize parameters for the otma_create and otma_open   */
/* APIs.                                                      */
/*------------------------------------------------------------*/

splat( grp_name,GROUP_NAME );               /* XCF Group Name     */
splat( tpipe_prefix,"TPAS" );               /* XCF Group Name     */
strcat(send_buf, temp_buf );       /* Copy temp_buf into send_buf */
strcat(send_buf, " " );            /* add a blank for strlen     */
buffer_length = strlen(send_buf);

/*************************************************************/
/* Example of setting up parms to Open the XCF Link          */
/*************************************************************/

 retrsn.ret    = -1;
 retrsn.rsn[0] = -1;
 retrsn.rsn[1] = -1;
 retrsn.rsn[2] = -1;
 retrsn.rsn[3] = -1;
 r             =  0;
```

```
  sessions      = 10;    /* OTMA supports multiple parallel        */
                         /* sessions (TPIPES) How many do you want?*/

/*****************************************************************/
/*BEGIN:                                                         */
/* We have a CREATE function to set up storage and               */
/* an OPEN function to start the protocol.                       */
/* If you don't need to customize the environment you can start  */
/* with the OPEN function, the CREATE will be done by OPEN.      */
/*****************************************************************/

  otma_create(&anchor,      /* (out) ptr to addr to receive ancho*/
        &retrsn,            /* (out) return code                 */
        (ecb_t *) &ecbOPEN,/* not posted by create but stored   */

        &grp_name,          /* (in) ptr to valid groupname       */
        &clt_name,          /* (in) Our member name              */
        &srv_name,          /* (in) Our server name              */

        &sessions,          /* (in) number of sessions to support*/
        &tpipe_prefix       /* (in) first part of tpipe name     */
        );

   printf("OTMA_CREATE issued. ret = %d rsn = %.8x,%.8x,%.8x,%.8x\n"
          "  anchor is at %.8x.\n",
          retrsn.ret,
          retrsn.rsn[0],
          retrsn.rsn[1],
          retrsn.rsn[2],
          retrsn.rsn[3],
          anchor);


 printf("-\n");


/*****************************************************************/
/*  Time to try to connect to IMS                                */
/*****************************************************************/

 /* ___start XCF connection_________________________ */

  otma_open(&anchor,        /* out ptr to addr to receive anchor */
        &retrsn,            /* out return code                   */
        (ecb_t *)&ecbOPEN, /* out posted by open if failure     */
                            /*     else posted by exit pgm       */
        &grp_name,          /* in ptr to valid XCF groupname     */
        &clt_name,          /* in Our member name                */
        &srv_name,          /* in Our server name                */

        &sessions,          /* in number of sessions to support  */
        &tpipe_prefix       /* in first part of tpipe name       */
        );

   printf("OTMA_OPEN issued. ret = %.8x rsn = %.8x,%.8x,%.8x,%.8x\n"
          " Waiting for ecb at %.8x.=%.8x.\n",
          retrsn.ret,
          retrsn.rsn[1],
          retrsn.rsn[2],
          retrsn.rsn[3],
          ecb_list[1],
          *ecb_list[1]
          );

  printf("-\n");

/* ------------------------------------------------------------- */
```

```
/* Here we wait for Open to signal complete                        */
/* ---------------------------------------------------------------- */
    DFSYCWAT(ecb_list[1]);   /* WAIT on ecb                */

     printf("OTMA_OPEN done. ret = %.8x rsn = %.8x,%.8x,%.8x,%.8x \n"
            "\nEcb at %.8x.= %.8x.\n",
            retrsn.ret,
            retrsn.rsn[0],
            retrsn.rsn[1],
            retrsn.rsn[2],
            retrsn.rsn[3],
            ecb_list[1], *ecb_list[1]
            );

     printf("Local Area Anchor at %8.8X = %8.8X\n",
                 &anchor, anchor);


 /* ----------------------------------------------------------*/
 /* The post code from open indicates success or failure      */
 /* ----------------------------------------------------------*/
    if (0!=(0x00ffffff & ecbOPEN))
      {
        printf("OPEN_OTMA ecb is posted failure.\n");
        return(retrsn.rsn[0]);
      }

 /************************************************************/
 /* This is the loop that sends and receives data.           */
 /*                                                          */
 /* This test program can iterate with multiple calls here.  */
 /************************************************************/

for (loop_count = 0 ; loop_count<iterations ; loop_count++)
{

 /* Change the environment to wait for ecbIO               */
 ecbIO = 0;                            /* clear ecb for reuse  */
 ecb_list[1] = (unsigned long *)       /* posted by OTMA       */
               ((unsigned long)&(ecbIO) |
               (unsigned long)0x80000000);  /* end of list     */

  if (loop_count != 0)
   {

    /* ----------------------------------------------------------*/
    /* If looping more than once open the next file to send      */
    /* and read it into the send_buf.                            */
    /* ----------------------------------------------------------*/

    if (( stream = fopen(sndfiledd[loop_count],"rb")) != NULL )
     {
      num = fread( temp_buf, sizeof( char ), NUM_BUFFER, stream );
      if (num == NUM_BUFFER) {
        fclose( stream );
      }
      else {
       if ( ferror(stream) )
        printf( "Error opening file  %s\n",sndfiledd[loop_count]);
        else if ( feof(stream)) {
          printf( "EOF found\n" );
          printf( "Number of characters read %d\n", num );
          printf( "temp_buf = %.*s\n", temp_buf);
          fclose( stream );
        }
      }
     }
```

```
  else
   printf( "Error opening file %s\n", sndfiledd[loop_count]);


  /*  Put data in to Send Buffer.  */
  memset(error_message_text ,0, sizeof(error_message_text));
  memset(send_buf ,0, sizeof(send_buf));
  strcat(send_buf, temp_buf );
  strcat(send_buf, " " );
  buffer_length = strlen(send_buf);

 }  /* end if loop_count != 0    */

  /* ------------------------------------------------------------*/
  /* If looping more than once open the next tpipe to use       */
  /* and read it into the tpipe.                                 */
  /* ------------------------------------------------------------*/

  if (( stream = fopen(tpipefiledd[loop_count],"rb")) != NULL )
   {
    num = fread( temp_buf, sizeof( char ), NUM_BUFFER, stream );
    if (num == NUM_BUFFER) {
      fclose( stream );
    }
    else {
     if ( ferror(stream) )
      printf( "Error opening file  %s\n",sndfiledd[loop_count]);
      else if ( feof(stream)) {
        printf( "EOF found\n" );
        printf( "Number of characters read %d\n", num );
        printf( "temp_buf = %.*s\n", temp_buf);
        fclose( stream );
      }
    }
   }
  else
   printf( "Error opening file %s\n", sndfiledd[loop_count]);

  memcpy(tpipe, temp_buf, 8);


/* Print announcement of send API.    */
printf("-\n-\n- Iteration #%d  Send API ---------------\n-\n",
        loop_count+1);
printf("Tpipe Name = %.8s.\n", tpipe);
printf("Transaction = %.8s.\n", trans);
printf("RACF UserID = %.8s.\n", user_name);
printf("RACF Group = %.8s.\n", user_prf);
printf("Lterm = %.8s.\n", lterm );
printf("Modname = %.8s.\n", modname );
printf("OTMA Data = %.50s.\n", otma_data );
printf("Send buf = %s.\n", send_buf);
printf("Send buf at %.8x.\n", &send_buf);
printf ("Buffer length = %d.\n", buffer_length);
printf ("Waiting for ecb at %.8x.=%.8x.\n", ecb_list[1],
        *ecb_list[1]);

  otma_send_async(
                &anchor,          /* (in)  anchor block         */
                &retrsn,          /* (out) return status        */
                &ecbIO,           /* (out) ecb address          */

                &tpipe,           /* (in)  user tpipe name      */
                &trans,           /* (in)  IMS trancode or cmd  */
                &user_name,       /* (in)  RACF userid          */
                &user_prf,        /* (in)  RACF group name      */
                &lterm,           /* (in)  logical terminal     */
```

```
                    &modname,         /* (in)  module name          */
                    &otma_data,       /* (in)  OTMA user data       */

 (unsigned char *) &send_buf,         /* (in)  send buffer          */
                    &buffer_length,   /* (in)  size of send buffer  */
                    0,                /* (in)  send_segment_list    */
                    &error_message,   /* (out) IMS Error msg.       */
                   &send_options);    /* (in)  send special options */

 DFSYCWAT(ecb_list[1]);              /* WAIT on ecb                */

 /* Print results of send API.    */
printf("OTMA_SEND_ASYNC done. ret = %.8x rsn = %.8x,%.8x,%.8x,%.8x\n"
       "Ecb at %.8x.=%.8x.\n",
           retrsn.ret,
           retrsn.rsn[0],
           retrsn.rsn[1],
           retrsn.rsn[2],
           retrsn.rsn[3],
           ecb_list[1],
           *ecb_list[1]
         );

  retsave = retrsn.ret;    /*  Save otma_send_async Return Code.  */


  /* Error Processing for OTMA_SEND_ASYNC API.          */
   if (retrsn.ret != 0)
   {

    /* ___Error path Free allocated session _____________________ */
    printf("-Error send_async API retrsn.ret=%d\n",retrsn.ret);
    printf( "Error message = %s\n", error_message );

if (( stream = fopen(infiledd[loop_count],"rb")) != NULL )
{
  num = fread( compare_buf, sizeof( char ), NUM_BUFFER, stream );
  if (num == NUM_BUFFER) {  /* fread success */
    printf( "Compare_buf = %.80s.\n", compare_buf );
    printf( "Error_buf   = %.80s.\n", error_message );
    fclose( stream );
    compare_result = memc( compare_buf, error_message );
    printf( "compare_result = %i\n",compare_result);
if (compare_result != 0)
  return(compare_result);        /* Exit if NO COMPARE    */
  }
  else {  /* fread() failed */
    if ( ferror(stream) )         /* possibility 1 */
      printf( "Error reading file %s\n", infiledd[loop_count]);
    else if ( feof(stream)) {     /* possibility 2 */
      printf( "EOF found\n" );
      printf( "Number of characters read %d\n", num );
      printf( "Receive compare_buf = %.*s\n", num, compare_buf);
    }
  }
}
else
  printf( "Error opening file %s\n", infiledd[loop_count]);

    printf("-\n");

    /* ___Sever IMS connection _________________________________ */
    printf("-\n");
    otma_close(
                    & anchor,        /* (in,out) tr to otma anchor */
                    & retrsn         /* (out) rc,reason (1-4)      */
            );
```

```
      printf("OTMA_CLOSE done. ret = %.8x rsn = %.8x,%.8x,%.8x,%.8x\n",
             retrsn.ret,
             retrsn.rsn[0],
             retrsn.rsn[1],
             retrsn.rsn[2],
             retrsn.rsn[3]
             );

      return (retsave);      /* EXIT with receive API return code */
     }

  /* Initialize otma_receive_async parameters.            */
   splat( rec_lterm , "" );
   splat( rec_modname , "" );
   splat( rec_otma_data , "" );
   ecbIO = 0;                              /* clear ecb for reuse      */
   ecb_list[1] = (unsigned long *)         /* posted by OTMA           */
   ((unsigned long)&(ecbIO) |
                    (unsigned long)0x80000000);  /* end of list     */

    /* Print announcement of receive API.    */
    printf("-\n-\n- Iteration #%d  Receive API ---------------\n-\n",
           loop_count+1);
    printf("Tpipe Name = %.8s.\n", tpipe);
    printf("Waiting for ecb at %.8x=%.8x.\n", ecb_list[1],
           *ecb_list[1]);

     otma_receive_async(
                     &anchor,          /* (in)  anchor block         */
                     &retrsn,          /* (out) return status        */
                     &ecbIO,           /* (out) ecb address          */

                     &tpipe,           /* (in)  user tpipe name      */
                     &rec_lterm,       /* (in)  logical terminal     */
                     &rec_modname,     /* (in)  module name          */
                     &rec_otma_data,   /* (in)  OTMA user data       */

    (unsigned char *) &rec_buf,        /* (out) Receive buffer       */
                     &rec_buffer_len,  /* (in)  size of rec buffer   */
                     &rec_data_len,    /* (in)  send_segment_list    */
                     0,                /* (in/out) rec multiple seg  */
                     &rec_options);    /* (in)  rec special options  */

 DFSYCWAT(ecb_list[1]);                /* WAIT on ecb                */
    /* Print results of receive API.    */
   printf("OTMA_REC_ASYNC done. ret = %.8x rsn = %.8x,%.8x,%.8x,%.8x\n"
           "Ecb at %.8x.=%.8x.\n",
             retrsn.ret,
             retrsn.rsn[0],
             retrsn.rsn[1],
             retrsn.rsn[2],
             retrsn.rsn[3],
             ecb_list[1],
             *ecb_list[1]);
    printf("Lterm = %.8s.\n", rec_lterm );
    printf("Modname = %.8s.\n", rec_modname );
    printf("OTMA Data = %.50s.\n", rec_otma_data );
    printf("Receive buf = %.80s.\n", rec_buf);
    printf("Receive buf at %.8x.\n", &rec_buf);
    printf("Data length = %d.\n", rec_data_len);
    printf("Buffer length = %d.\n", rec_buffer_len);


    retsave = retrsn.ret;    /* Save otma_receive_async Return Code.  */

    /* Error Processing for OTMA_RECEIVE_ASYNC API.          */
```

```
    if (retrsn.ret != 0)
    {

     /* ___Error path Free allocated session _____________________ */
     printf("-error path retrsn.ret=%d\n",retrsn.ret);
     printf("-\n");

     /* ___Sever IMS connection _________________________ */
     printf("-\n");
     otma_close(
                   & anchor,        /* (in,out) tr to otma anchor */
                   & retrsn         /* (out) rc,reason (1-4)      */
             );

     printf("OTMA_CLOSE done. ret = %.8x rsn = %.8x,%.8x,%.8x,%.8x\n",
            retrsn.ret,
            retrsn.rsn[0],
            retrsn.rsn[1],
            retrsn.rsn[2],
            retrsn.rsn[3]
            );

     return (retsave);      /* EXIT with receive API return code */
    }


  /* ------------------------------------------------------------*/
  /*  Open the compare file containing the expected output      */
  /*  of the receive buffer.  Compare the expected output       */
  /*  with the actual output and return the result.             */
  /* ------------------------------------------------------------*/

   printf("-\n-\n- Iteration #%d  Data Validation -----------\n-\n",
          loop_count+1);

 if (( stream = fopen(infiledd[loop_count],"rb")) != NULL )
 {
   num = fread( compare_buf, sizeof( char ), NUM_BUFFER, stream );
   if (num == NUM_BUFFER) {  /* fread success */
     printf( "compare_buf = %.80s.\n", compare_buf );
     printf( "    rec_buf = %.80s.\n", rec_buf );
     fclose( stream );
     compare_result = memc( compare_buf, rec_buf );
     printf( "compare_result = %i\n",compare_result);
 if (compare_result != 0)
   return(compare_result);        /* Exit if NO COMPARE    */
   }
   else {  /* fread() failed */
     if ( ferror(stream) )         /* possibility 1 */
       printf( "Error reading file %s\n", infiledd[loop_count]);
     else if ( feof(stream)) {     /* possibility 2 */
       printf( "EOF found\n" );
       printf( "Number of characters read %d\n", num );
       printf( "Receive compare_buf = %.*s\n", num, compare_buf);
     }
   }
 }
 else
   printf( "Error opening file %s\n", infiledd[loop_count]);

  memset(rec_buf ,' ', sizeof(rec_buf));

 printf( "End of loop \n" );
}      /* end of loop */


   printf("-\n");
```

```
   /*****************************************************************/
   /* Finally, CLOSE severs the connection with IMS and frees the   */
   /* Storage used by the OTMA API.                                 */
   /* This will be done at job-step termination but its untidy.     */
   /*****************************************************************/

    otma_close(
                     & anchor,        /* (in,out) ptr to otma anchor */
                     & retrsn         /* (out)  rc,reason (1-4)      */
              );
    printf("OTMA_CLOSE done. ret = %.8x rsn = %.8x,%.8x,%.8x,%.8x \n",
            retrsn.ret,
            retrsn.rsn[0],
            retrsn.rsn[1],
            retrsn.rsn[2],
            retrsn.rsn[3]
            );

    return (compare_result);      /* We're done */
} /* end of main */

/*====================================================================*/
/*      Subroutine to compare expected results(compare_buf)           */
/*      with actual results(err_msg) the "|" is used to signify       */
/*      an ignore compare and "?" is used to mark the end of string. */
/* Note: Compare starts using an index i=1 ie. the 2nd character      */
/*       because the 1st character was blanked out. ( NL x'15' )      */
/*====================================================================*/


int memc(char *comp_buf, char *rec_buf1)
{

 int j;
 int i;

  j = 0;

  for (i=1;
      ( (j==0) && (comp_buf[i] != '?') );
       i++ )
    {
      if( comp_buf[i] != '|' )                    /* Ignore compare  */
      {
        if( comp_buf[i] != rec_buf1[i])           /* compare ok ?    */
        {
          j++;                                    /* No              */
          printf( "MISCOMPARE !!!   \n" );
          printf( "comp_buf[%d] = %c\n", i, comp_buf[i] );
          printf( "rec_buf1[%d] = %c\n", i, rec_buf1[i] );
        }
        else
          ;
      }
      else
        ;                                        /* Else null        */
    }

 return (j);
}
```

Appendix. Frequently Asked Questions about OTMA

This appendix provides answers to some of the more frequently asked questions about OTMA.

Q: What software do I need to use OTMA?

A: You need the following software:

- **IMS**: minimally IMS/ESA Version 5, or subsequent versions, releases, and modification levels.
- **An operating system**: minimally MVS/ESA Version 5, or subsequent versions, releases and modification levels.
- **An OTMA client**: use a client provided by IBM, by another vendor, or write your own client.

Q: Do I need to modify my IMS applications?

A: No, but if you have applications that use `SETO` calls, you might have to modify them. The `SETO` call is relatively new and applies to APPC/IMS and SPOOL/API processing.

For each OTMA-originated transaction, the `SETO` call returns a status code. You can bracket the `SETO` call with an `INQY` call if necessary; see “Using DL/I Calls in an OTMA Environment” on page 54.

Q: What is a Tpipe?

A: A transaction pipe (Tpipe) is a logical connection between a client and the server. It is analogous to an IMS logical terminal (LTERM).

A client associates its transactions with a transaction pipe. The server sends output to the client using the transaction pipe and lets the client deliver the output to its final destination (for example, a user at a terminal or a printer).

Q: How can a client know whether or not a send-then-commit transaction ran successfully?

A: OTMA sends message DFS2082 to the client if the IMS transaction ends unsuccessfully and does not perform an insert to the IOPCB.

Q: Does specifying a transaction pipe as synchronized make the communication flow half-duplex?

A: No. Transaction pipes are always full-duplex.

Whether the communication flow is actually half-duplex depends on the client. For a synchronized transaction pipe, IMS processes all output messages in the order received. No new messages are sent for the transaction until IMS has received an ACK message for the previous message. A NAK message causes IMS to halt all output processing for that transaction.

While this output processing is taking place, the client could be sending new input transaction messages to IMS on that synchronized transaction pipe. If the client coordinates the sending of transactions with the receipt of IMS output, the client can effect half-duplex processing.

Q: Are transactions using synchronized transaction pipes recoverable?

A: Yes. Input messages are not recoverable for send-then-commit transactions, and requesting an ACK message has no effect on whether a transaction is recoverable. You should request ACK messages for proper synchronization of the synchronized transaction pipe.

Also, when a transaction reaches IMS, its recoverability depends on how it is defined to IMS.

Q: Is the ACEE refresh age controlled on a client basis or a user basis? How can the client or user refresh the ACEE?

A: The access control environment element (ACEE) is refreshed for each client based on the aging value set in the state-data section of the message prefix. This value applies to the client, not to specific users.

To refresh the ACEE, a client should send a command message with the aging value set to 0 (zero) to request that IMS call RACF to refresh the ACEE. A client should not leave and rejoin the XCF group to cause the ACEE refresh.

Q: Why would I use a synchronized Tpipe versus a nonsynchronized Tpipe?

A: Use a synchronized Tpipe to ensure that client transactions are processed only once in the case of a client or IMS failure. Therefore, synchronized Tpipes ensure better transaction recoverability. However, in order to guarantee transaction recovery, you are required to implement resynchronization logic with synchronized Tpipes.

Use a nonsynchronized Tpipe when the recoverability of a transaction is less of a concern. For nonsynchronized Tpipes, the client does not require the resynchronization logic.

Q: What is the major difference between the commit-then-send processing option and the send-then-commit processing option?

A: The commit-then-send processing option commits the transaction output as part of sync-point processing, and then delivers the output to the client later.

The send-then-commit processing option delivers the transaction output first, receives an acknowledgment from the client, and then completes the sync-point processing.

Q: What happened to the commit mode 0 and commit mode 1 processing options?

A: Commit mode 0 is now called “commit-then-send”, and commit mode 1 is called “send-then-commit”. Because the terms “commit-then-send” and “send-then-commit” are more intuitive when referring to these processing options, the terms “commit mode 0” and “commit mode 1” are no longer used.

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.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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 Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

This book is intended to help you use IMS Open Transaction Manager Access (OTMA) and can be used to write an OTMA client. This book documents General-use Programming Interface and Associated Guidance Information provided by IMS.

General-use programming interfaces allow the customer to write programs that obtain the services of IMS.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

BookManager
CICS
DB2
IBM
IMS
IMS/ESA
Java
MQSeries
MVS
MVS/ESA
OS/390
RACF
VTAM
z/OS

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Bibliography

This bibliography lists all of the information in the IMS Version 9 library.

- *IMS Queue Control Facility for z/OS: User's Guide and Reference*, SC26-9685
- *IMS Message Requeuer Program Description/Operations Manual*, SH21-1089
- *z/OS MVS Programming: Authorized Assembler Services Guide*, GC28-1763
- *z/OS MVS Programming: Authorized Assembler Services Reference, Volume 1*, SA22-7609
- *z/OS MVS Programming: Authorized Assembler Services Reference, Volume 2*, SA22-7610
- *z/OS MVS Programming: Authorized Assembler Services Reference, Volume 3*, SA22-7611
- *z/OS MVS Programming: Authorized Assembler Services Reference, Volume 4*, SA22-7612
- *z/OS MVS Programming: Resource Recovery*, GC28-1739
- *z/OS MVS Programming: Sysplex Services Reference*, GC28-1772
- *MVS/ESA System Commands*, GC28-1442
- *z/OS MVS Initialization and Tuning Reference*, SA22-7592

IMS Version 9 Library

ZES1-2330	ADB	*IMS Version 9: Administration Guide: Database Manager*
ZES1-2331	AS	*IMS Version 9: Administration Guide: System*
ZES1-2332	ATM	*IMS Version 9: Administration Guide: Transaction Manager*
ZES1-2333	APDB	*IMS Version 9: Application Programming: Database Manager*
ZES1-2334	APDG	*IMS Version 9: Application Programming: Design Guide*
ZES1-2335	APCICS	*IMS Version 9: Application Programming: EXEC DLI Commands for CICS and IMS*
ZES1-2336	APTM	*IMS Version 9: Application Programming: Transaction Manager*
ZES1-2337	BPE	*IMS Version 9: Base Primitive Environment Guide and Reference*
ZES1-2338	CR	*IMS Version 9: Command Reference*
ZES1-2339	CQS	*IMS Version 9: Common Queue Server Guide and Reference*
ZES1-2340	CSL	*IMS Version 9: Common Service Layer Guide and Reference*
ZES1-2341	CG	*IMS Version 9: Customization Guide*
ZES1-2342	DBRC	*IMS Version 9: DBRC Guide and Reference*
ZES1-2343	DGR	*IMS Version 9: Diagnosis Guide and Reference*
ZES1-2344	FAST	*IMS Version 9: Failure Analysis Structure Tables (FAST) for Dump Analysis*
ZES1-2346	OLR	*IMS Version 9: HALDB Online Reorganization Guide*
ZES1–2380	CT	*IMS Version 9: IMS Connect Guide and Reference*
ZES1-2347	JGR	*IMS Version 9: IMS Java Guide and Reference*
ZES1-2348	IIV	*IMS Version 9: Installation Volume 1: Installation Verification*
ZES1-2349	ISDT	*IMS Version 9: Installation Volume 2: System Definition and Tailoring*
ZES1-2350	INTRO	*IMS Version 9: An Introduction to IMS*
ZES1-2351	MIG	*IMS Version 9: Master Index and Glossary*
ZES1-2352	MC1	*IMS Version 9: Messages and Codes, Volume 1*
ZES1-2353	MC2	*IMS Version 9: Messages and Codes, Volume 2*
ZES1-2354	OTMA	*IMS Version 9: Open Transaction Manager Access Guide and Reference*
ZES1-2355	OG	*IMS Version 9: Operations Guide*
GC17-7831	RPG	*IMS Version 9: Release Planning Guide*
ZES1-2358	URDBTM	*IMS Version 9: Utilities Reference: Database and Transaction Manager*
ZES1-2359	URS	*IMS Version 9: Utilities Reference: System*

Supplementary Publications

GC17-7825	LPS	*IMS Version 9: Licensed Program Specifications*
ZES1-2357	SOC	*IMS Version 9: Summary of Operator Commands*

Publication Collections

LK3T-7213	CD	IMS Version 9 Softcopy Library
LK3T-7144	CD	IMS Favorites
LBOF-7789	Hardcopy and CD	Licensed Bill of Forms (LBOF): IMS Version 9 Hardcopy and Softcopy Library
SBOF-7790	Hardcopy	Unlicensed Bill of Forms (SBOF): IMS Version 9 Unlicensed Hardcopy Library
SK2T-6700	CD	OS/390 Collection
SK3T-4270	CD	z/OS Software Products Collection
SK3T-4271	DVD	z/OS and Software Products DVD Collection

Accessibility Titles Cited in this Book

SA22-7787		z/OS V1R1.0 TSO Primer
SA22-7794		z/OS V1R1.0 TSO/E User's Guide
SC34-4822		z/OS V1R1.0 ISPF User's Guide, Volume 1

Index

Special characters

Numerics

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

X

Z

Program Number: 5655-J38

IBM Confidential
Printed in USA

ZES1-2354-01

Spine information:

IMS

Open Transaction Manager Access Guide and Reference

Version 9