CICS Transaction Server for z/OS
Version 5 Release 4

Distributed Transaction Programming Guide

IBM

Note

Before using this information and the product it supports, read the information in "Notices" on page 151.

This edition applies to the IBM CICS® Transaction Server for z/OS® Version 5 Release 4 (product number 5655-Y04) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About this PDF

This PDF describes the technique (called distributed transaction processing, or DTP) of spreading the functions of a transaction over several transaction programs in a network. It also provides guidance in producing application programs that exchange data through distributed transaction processing (DTP) on Advanced Program-to-Program Communication (APPC), multiregion operation(MRO), and LUTYPE6.1 links.

For details of the terms and notation used in this book, see Conventions and terminology used in the CICS documentation in IBM Knowledge Center.

Date of this PDF

This PDF was created on January 20th 2020.

Chapter 1. Distributed transaction processing

The technique of distributing the functions of a transaction over several transaction programs within a network is called **distributed transaction processing (DTP)**.

This chapter contains the following topics:

Overview of DTP

When CICS arranges function shipping, distributed program link (DPL), asynchronous transaction processing, or transaction routing for you, it establishes a logical data link with a remote system.

A data exchange between the two systems then follows. This data exchange is controlled by CICS-supplied programs, using APPC, LUTYPE6.1, or MRO protocols. The CICS-supplied programs issue commands to allocate conversations, and send and receive data between the systems. Equivalent commands are available to application programs, to allow applications to converse. The technique of distributing the functions of a transaction over several transaction programs within a network is called **distributed transaction processing (DTP)**.

Of the five intercommunication facilities, DTP is the most flexible and the most powerful, but it is also the most complex. This chapter introduces you to the basic concepts.

Advantages over function shipping and transaction routing

Distributed transaction processing has advantages over function shipping and transaction routing.

Function shipping gives access to remote resources, and transaction routing lets a terminal communicate with remote transactions. These two facilities might appear sufficient for all your intercommunication needs, especially from a functional perspective. However, you must consider other design criteria, for example, machine loading, response time, continuity of service, and economic use of resources.

Consider the example of a supermarket chain. It has many branches that each stock a different range of goods, which are served by several distribution centers. Local stock records at the branches are updated online from point-of-sale terminals. Sales information must be sorted for the separate distribution centers, and transmitted to them to enable reordering and distribution.

An analyst might consider using function shipping to write each reorder record to a remote file as it arises. This method has simplicity, but must be rejected for several reasons:

- Data is transmitted to the remote systems irregularly in small packets. This means inefficient use of the links.
- The transactions associated with the point-of-sale devices are competing for sessions with the remote systems. This could mean unacceptable delays at point-of-sale.
- Failure of a link results in a catastrophic suspension of operations at a branch.
- Intensive intercommunication activity (for example, at peak periods) causes reduction in performance at the terminals.

Now consider the solution where each sales transaction writes its reorder records to a transient data queue. The data is quickly disposed of, leaving the transaction to carry on its conversation with the terminal.

Restocking requests are seldom urgent, so it might be possible to delay the sorting and sending of the data until an off-peak period. Alternatively, the transient data queue could be set to trigger the sender transaction when a predefined data level is reached. Either way, the sender transaction has the same job to do.

Again, it is tempting to use function shipping to transmit the reorder records. After the sort process, each record could be written to a remote file in the relevant remote system. However, this method is still not ideal. The sender transaction would have to wait after writing each record to make sure that it got the correct response. Apart from using the link inefficiently, waiting between records would make the whole process impossibly slow. You can use distributed transaction processing to solve this problem, and others.

The flexibility of DTP can, in some circumstances, be used to achieve improved performance over function shipping. Consider browsing a remote file to select a record that satisfies some criteria. If you use function shipping, CICS ships the GETNEXT request across the link, and lets the mirror perform the operation and ship the record back to the requester. This is a lot of activity (two flows on the network) and the data flow can be significant. If the browse is on a large file, the overhead can be unacceptably high.

One alternative is to write a DTP conversation that ships the selection criteria, and returns only the keys and relevant fields from the selected records. This reduces both the number of flows and the amount of data sent over the link, thus reducing the overhead incurred in the function-shipping case.

Why distributed transaction processing?

In a multisystem environment, data transfers between systems are necessary because users need access to remote resources.

In managing these resources, network resources are used. But performance suffers if the network is used excessively. There is therefore a performance gain if application design is oriented toward doing the processing associated with a resource in the resource-owning region.

DTP lets you process data at the point where it arises, instead of overworking network resources by assembling it at a central processing point.

There are, of course, other reasons for using DTP. DTP does the following:

- Allows some measure of parallel processing to shorten response times
- Provides a common interface to a transaction that is to be attached by several different transactions
- Enables communication with applications running on other systems, particularly on non-CICS systems
- Provides a buffer between a security-sensitive file or database and an application, so that no application need know the format of the file records
- Enables batching of less urgent data destined for a remote system.

DTP's place in the CICS intercommunication facilities

Today, an increasing number of organizations are connecting their information systems together and distributing resources among them. To support this kind of processing, applications need to be designed and developed to access resources across multiple systems.

So CICS provides the following basic intercommunication facilities:

- *Function shipping*, which enables your application program to access resources in another CICS system.
- *Distributed program link*, which enables a program in one CICS system to issue a link command that invokes a program in another CICS system, waiting for a RETURN.

- *Asynchronous processing*, which enables a CICS transaction to initiate a transaction in another CICS system and pass data to it.
- *Transaction routing*, which enables a terminal connected to one CICS system to run a transaction in another CICS system.
- *Distributed transaction processing*, which enables a CICS transaction to communicate with a transaction running in another system. The transactions are designed and coded specifically to communicate with each other, and in doing so to use the intersystem link with maximum efficiency.

In addition, CICS provides the following methods of accessing CICS programs and transactions from non-CICS environments:

- The CICS bridge
- The external CICS interface (EXCI)
- Transactional EXCI
- Support for ONC Remote Procedure Calls
- The web interface.

This information discusses only distributed transaction processing. The other basic intercommunication facilities are described in Intercommunication methods.

What is DTP?

DTP is one of the ways in which CICS allows processing to be split between intercommunicating systems. Only DTP allows two or more communicating application programs to run simultaneously in different systems and to pass data back and forth between themselves—that is, to carry on a conversation.

Of the intercommunication facilities offered by CICS, DTP is the most flexible and powerful, but also the most complex. This section introduces you to the basic concepts involved in creating DTP applications. For a broad discussion of intercommunication concepts, see Getting started with intercommunication.

DTP allows two or more partner programs in different systems to interact with each other for some purpose. DTP enables a CICS transaction to communicate with one or more transactions running in different systems. A group of such connected transactions is called a **distributed process**.

The process can best be shown by discussing the operation of DTP between two CICS systems, CICSA and CICSB. The configuration is shown in Figure 1 on page 3.

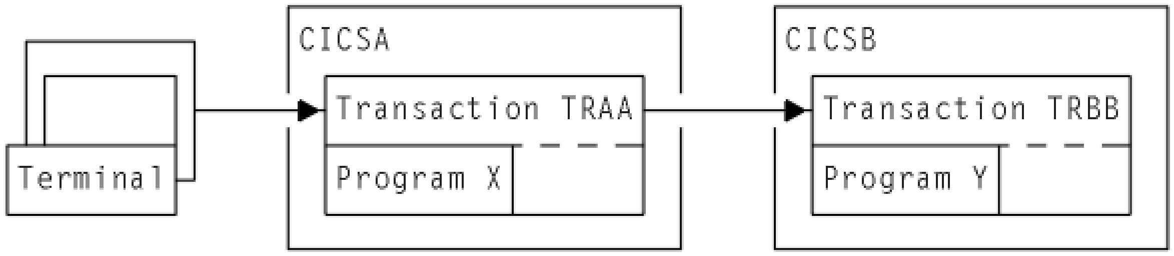

Figure 1. DTP between two CICS transactions

1. A transaction (TRAA) is initiated on CICSA, for example, by a terminal operator keying in a transaction ID and initial data.
2. To fulfill the request, the processing program X begins to execute on CICSA, probably reading initial data from files, perhaps updating other files and writing to print queues.
3. Without ending, program X asks CICSA to establish a communication session with another CICS system, CICSB. CICSA responds to the request.
4. Also without ending, program X sends a message across the communication session, asking CICSB to start a new transaction, TRBB. CICSB initiates transaction TRBB by invoking program Y.

5. Program X now sends and receives messages, including data, to and from program Y. Between sending and receiving messages, both program X and program Y continue normal processing completely independently. When the two programs communicate, their messages can consist of:

   - Agreements on how to proceed with communication or how to end it. For example, program X can tell program Y when it may transmit messages across the session. At any time, both programs must know the state of their communication, and thus, what actions are allowed. At any time, either system may have actual control of the communication.
   - Agreements to make permanent all changes made up to that point. This allows the two programs to **synchronize** changes. For example, a dispatch billing program on CICSA might want to commit delivery and charging for a stock item, but only when a warehouse program in CICSB confirms that it has successfully allocated the stock item and adjusted the inventory file accordingly.
   - Agreements between CICSA and CICSB to cancel, rather than make permanent, changes to data made since a given point. Such a cancelation (or rollback) might occur when customers change their minds, for example. Alternatively, it might occur because of uncertainty caused by failure of the application, the system, the communication path, or the data source.

Although the two programs X and Y exist as independent units, it is clear that they are designed to work as one. Of course, DTP is not limited to pairs of programs. You can chain many programs together to distribute processing more widely. This is discussed later in the book.

In the overview of the process, the location of program Y has not been specified. Program X is a CICS program, but program Y need not be, because CICS can establish sessions with non-CICS, LUTYPE6.1, MRO, or APPC partners. This is discussed in "Designing distributed processes" on page 8.

Conversations

Although several programs can be involved in a single distributed process, information transfer within the process is always between self-contained *communication pairs*. The exchange of information between a pair of programs is called a *conversation*.

During a conversation, both programs are active; they send data to and receive data from each other. The conversation is two-sided but at any moment, each partner in the conversation has more or less control than the other. According to its level of control (known as its *conversation state*), a program has more or less choice in the commands that it can issue.

Conversation states

Thirteen conversation states have been defined for CICS DTP. The set of states possible for a particular conversation depends on the protocol and synchronization level used.

The concepts of protocol and synchronization level are explained in "Selecting the protocol" on page 10 and "Maintaining data integrity" on page 7 respectively. Table 1 on page 4 shows which conversation states are defined for which protocols and synchronization levels.

Table 1. The conversation states defined for different protocols. Yes and no indicate whether the state is defined.

State number	State name	APPC sync level 0	APPC sync level 1	APPC sync level 2	MRO	LUTYPE6.1 normal mode	LUTYPE6.1 migration mode
1	Allocated	Yes	Yes	Yes	Yes	Yes	Yes
2	Send	Yes	Yes	Yes	Yes	Yes	Yes
3	Pendreceive	Yes	Yes	Yes	No	Yes	Yes
4	Pendfree	Yes	Yes	Yes	Yes	Yes	Yes
5	Receive	Yes	Yes	Yes	Yes	Yes	Yes
6	Confreceive	No	Yes	Yes	No	No	Yes

Table 1. The conversation states defined for different protocols. Yes and no indicate whether the state is defined. *(continued)*

State number	State name	APPC sync level 0	APPC sync level 1	APPC sync level 2	MRO	LUTYPE6.1 normal mode	LUTYPE6.1 migration mode
7	Confsend	No	Yes	Yes	No	No	Yes
8	Conffree	No	Yes	Yes	No	No	Yes
9	Syncreceive	No	No	Yes	Yes	Yes	Yes
10	Syncsend	No	No	Yes	No	Yes	Yes
11	Syncfree	No	No	Yes	Yes	Yes	Yes
12	Free	Yes	Yes	Yes	Yes	Yes	Yes
13	Rollback	No	No	Yes	Yes	No	Yes

By using a special CICS command (EXTRACT ATTRIBUTES STATE), or the STATE option on a conversation command, a program can obtain a value that indicates its own conversation state. CICS places such a value in a variable named by the program; the variable is sometimes referred to as a **state variable**. Knowing the current conversation state, the program then knows which commands are allowed. If, for example, a conversation is in **send state**, the transaction can send data to the partner. (The transaction can take other actions instead, as indicated in the relevant state table.)

When a transaction issues a DTP command, this can cause the conversation state to change. For example, a transaction can deliberately switch the conversation from **send state** to **receive state** by issuing a command that invites the partner to send data. When a conversation changes from one state to another, it is said to undergo a **state transition**.

Not only does the conversation state determine what commands are allowed, but the state on one side of the conversation reflects the state on the other side. For example, if one side is in **send state**, the other side is in either **receive state**, **confreceive state**, or **syncreceive state**.

Sessions

A conversation takes place across a CICS resource called a **session**. One transaction (known as the **front-end transaction**) asks CICS to allocate a session, and then uses this session to request that the remote transaction (known as the **back-end transaction**) be initiated. Then the two transactions, which can be thought of as partners in the conversation, can “talk to” each other.

A session is a logical data path between two logical units. It is a shared resource and is allocated to a transaction in response to a request from the transaction. Resource definition determines the number of sessions available for allocation. While a conversation is active, it has sole use of the session allocated to it.

A transaction starts a conversation by requesting the use of a session to a remote system. When it obtains the session, the transaction can issue commands that cause an *attach* request to be sent to the other system to activate the transaction that is to be the conversation partner. A transaction can issue an attach request to more than one other transaction.

Distributed processes

A transaction can initiate other transactions, and hence, conversations. In a complex process, a distinct hierarchy emerges, usually with the terminal-initiated transaction at the top.

Figure 2 on page 6 shows a possible configuration. In this example, transaction TRAA, in system CICSA, is initiated from a terminal. Transaction TRAA attaches transaction TRBB to run in system CICSB. Transaction TRBB in turn attaches transaction TRCC in system CICSC and transaction TRDD in system

CICSD. Both transactions TRCC and TRDD attach the same transaction SUBR in system CICSE, thus giving rise to two copies of SUBR.

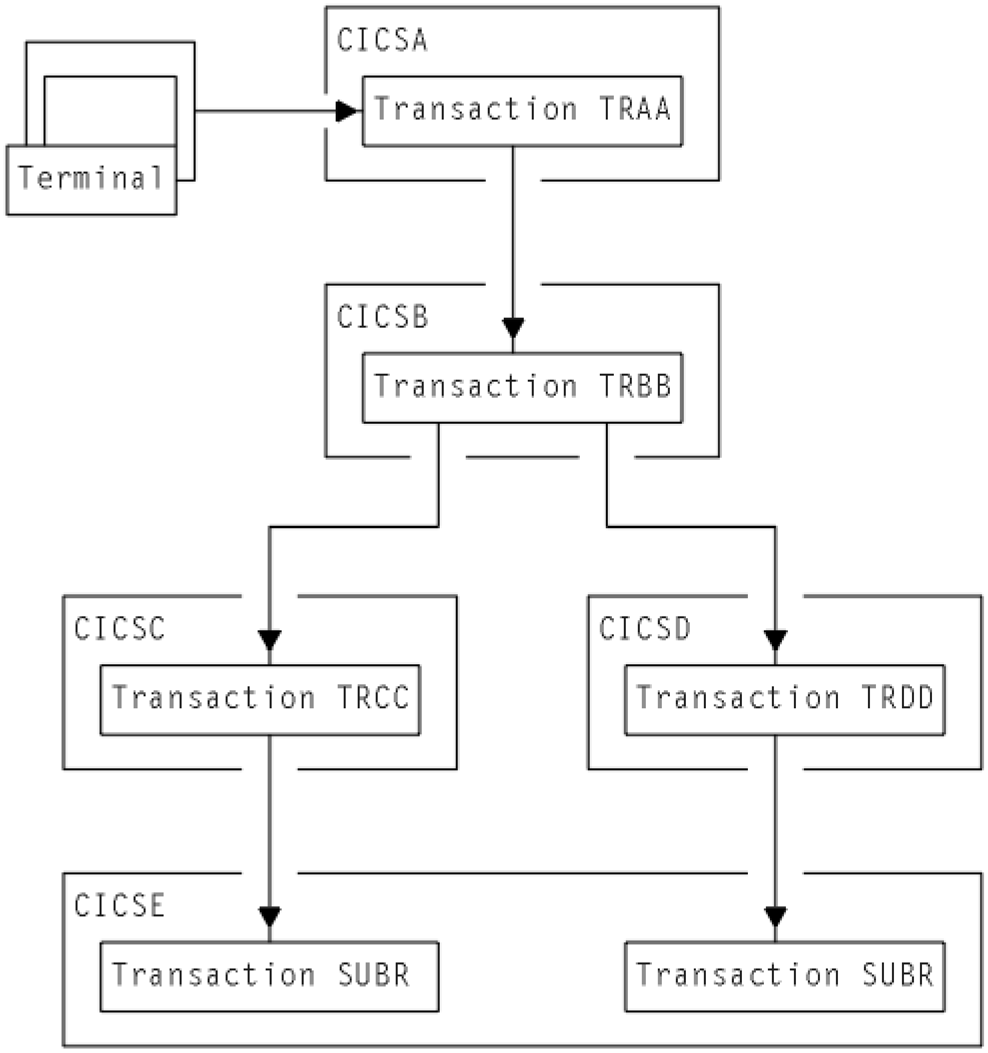

Figure 2. DTP in a distributed process

Notice that, for every transaction, there is only one *inbound* attach request, but that there can be a number of *outbound* attach requests. The session that activates a transaction is called its **principal facility**. A session that is allocated by a transaction to activate another transaction is called its **alternate facility**. Therefore, a transaction can have only one principal facility, but several alternate facilities.

When a transaction initiates a conversation, it is the front-end transaction on that conversation. Its conversation partner is the back-end transaction on the same conversation. It is normally the front-end transaction that dominates, and determines the way the conversation goes. This style of processing is sometimes referred to as the client/server model. (In some books, it is called master/slave.)

Alternatively, the front-end transaction and back-end transaction may switch control between themselves. This style of processing is called **peer-to-peer**. As the name implies, this model describes communication between equals. You are free to select whichever model you need when designing your application; CICS supports both.

Maintaining data integrity

Design your application to cope with the things that can go wrong while a transaction is running, for example, a session failing. The conversation protocol helps you recover from errors and ensures that the two sides remain in step with each other. This use of the protocol is called *synchronization*.

Synchronization allows you to protect recoverable resources such as transient data queues and files, whether they are local or remote. *Whatever goes wrong during the running of a transaction should not leave the associated resources in an inconsistent state*.

An application program can cancel all changes made to recoverable resources since the last known consistent state. This process is called *rollback*. The physical process of recovering resources is called *backout*. The condition that exists as long as there is no loss of consistency between distributed resources is called *data integrity*.

Sometimes you might need to backout changes to resources, even though no error conditions have arisen. Consider an order entry system. While entering an order for a customer, an operator is told by the system that the customer's credit limit would be exceeded if the order went through. Because there is no use continuing until the customer is consulted, the operator presses a function key to abandon the order. The transaction is programmed to respond by returning the data resources to the state they were in at the start of the order transaction.

The point in a process where resources are declared to be in a known consistent state is called a *synchronization point*, often shortened to *sync point*. Sync points are implied at the beginning and end of a transaction. A transaction can define other sync points by program command. All processing between two sync points belongs to a **unit of work** (UOW). In a distributed process, this is also known as a *distributed unit of work*.

When a transaction issues a sync point command, CICS *commits* all changes to recoverable resources associated with that transaction. After the sync point, the transaction can no longer back out changes made since the previous sync point. They have become irreversible.

Although CICS can commit and backout changes to local and remote resources for you, this service must be paid for in performance. If the recovery of resources throughout a distributed process is not a problem (for example, in an inquiry-only application), you can use simpler methods of synchronization.

Synchronization levels

Systems Network Architecture (SNA) defines three levels of synchronization for conversation using the APPC protocol.

The levels are:

- Level 0 – None
- Level 1 – Confirm
- Level 2 – Syncpoint

  Note: Sync level 2 is not supported on single-session connections.

  .

At sync level 0, there is no CICS support for synchronization of remote resources on connected systems. But it is still possible, under the control of the application to achieve some degree of synchronization by interchanging data, using the SEND and RECEIVE commands.

At sync level 1, you can use special commands for communication between the two conversation partners. One transaction can *confirm* the continued presence and readiness of the other. Both transactions are responsible for preserving the data integrity of recoverable resources by issuing syncpoint requests at the appropriate times.

At sync level 2, all syncpoint requests are automatically propagated across multiple systems. CICS implies a syncpoint when it starts a transaction; that is, it initiates logging of changes to recoverable resources, but no control flows take place. CICS takes a syncpoint when one of the transactions terminates normally. One abending transaction causes all to rollback. The transactions themselves can initiate syncpoint or rollback requests. However, a syncpoint or rollback request is propagated to another transaction only when the originating transaction is in conversation with the other transaction, and sync level 2 has been selected.

Bear in mind that syncpoint and rollback are not limited to any one conversation within a transaction. They are propagated on every conversation currently active at sync level 2.

Designing distributed processes

These topics discuss the issues you must consider when designing distributed processes to run under APPC or MRO. These issues include structuring distributed processes and designing conversations.

It is assumed that you are already familiar with the issues involved in designing applications in single CICS systems, as described in What is a CICS application?.

Structuring distributed transactions

As with many design problems, designing a DTP application involves dealing with several conflicting objectives that must be carefully balanced against each other. These include performance, ease of maintenance, reliability, security, connectivity to existing functions, and recovery.

Avoiding performance problems

If performance is the highest priority, you must design your application so that data is processed as close to its source as possible. This avoids unnecessary transmission of data across the network. Alternatively, if processing can be deferred, you might want to consider batching data locally before transmitting.

To maintain performance across the intersystem connection, the conversation must be freed as soon as possible — so that the session can be used by other transactions. In particular, avoid holding a conversation across a terminal wait.

In terminal-attached transactions, pseudo-conversational design improves performance by reducing the amount of time a transaction holds CICS resources. A terminal user is likely to take seconds or even minutes to respond to any request for keyboard input. In contrast, the communication delay associated with a conversation between partner transactions is likely to be only a few milliseconds. It is therefore not necessary to terminate a front-end transaction pending a response from a back-end transaction.

However, a front-end transaction can be terminal-initiated, in which case a pseudo-conversational design might be appropriate. When input from the terminal user is required, terminate the the front-end transaction and its conversations. After the terminal user has responded, the successor front-end transaction can initiate a successor back-end transaction. If the first back-end transaction has to pass information to its successor, the information must either be passed to the front-end transaction or stored locally (for example, in temporary storage).

Stored information must be retrievable by identifiers that are not associated with the particular session used by the conversation. The back-end transaction cannot use a COMMAREA, a RETURN TRANSID, nor a TCTUA for this purpose. Instead, it can construct the identifier of a temporary storage queue by using information obtained from the front-end transaction. You can use the sysid of the principal facility and the identifier of the terminal to which the front-end transaction is attached.

Making maintenance easier

To correct errors or to adapt to the evolving needs of an organization, distributed processes inevitably have o be modified. Whether these changes are made by the original developers or by others, this task is likely to be easier if the distributed processes are relatively simple. So consider minimizing the number of transactions involved in a distributed process.

Going for reliability

If you are particularly concerned with reliability, consider minimizing the number of transactions in the distributed process.

Protecting sensitive data

If the distributed process is to handle security-sensitive data, you could place this data on a single system. Using a single system means that only one of the transactions needs knowledge of how or where the sensitive data is stored. For guidance on implementing security in CICS systems, see Security facilities in CICS.

Maintaining connectivity

If you require connectivity to transactions running in a back-level CICS system, check that the functions required are compatible in both systems.

The following aspects of distributed process design differ from single-system considerations:

Data conversion
For non-EBCDIC APPC logical units, some data conversion might be required on either receipt or sending of data.

Using multiple conversations
When using multiple, serial conversations, CICS might provide different conversation identifiers to the transaction. It is therefore not advisable to use the conversation identifier for naming resources; for example, temporary storage queues.

Safeguarding data integrity

If it is important for you to be able to recover your data when things go wrong, design conversations for sync level 2, and keep the units of work as small as possible. However, this is not always possible, because the size of a UOW is determined largely by the function being performed. Remember that CICS syncpoint processing has no information about the structure and purpose of your application. As an application designer, you must ensure that syncpoints are taken at the right time and place, and to good purpose. If you do, error conditions are unlikely to lead to inconsistencies in recoverable data resources.

Here is an example of a distributed application that transfers the contents of a temporary storage queue from system A to system B, using a pair of transactions (TRAA in system A, and TRBB in system B), and a conversation at synclevel 2:

1. Transaction TRAA in system A reads a record from the temporary storage queue.
2. Transaction TRAA sends the record to system B, and waits for the response.
3. Transaction TRBB in system B receives the record from system A.
4. Transaction TRBB processes the record, and sends a response to system A.
5. Transaction TRAA receives the response, and deletes the record from the temporary storage queue.

These steps are repeated as long as there are records remaining in the queue. When the queue is empty:

1. Transaction TRAA sends a 'last record' indicator to system B.
2. Transaction TRBB sends a response to system A.

There are several points at which you can consider taking a syncpoint. Here are the relative merits of taking a syncpoint at each of these points:

At the start of processing
Because a UOW starts at this point, a syncpoint has no effect. In fact, if TRBB tries to take a syncpoint without having first issued a command to receive data, it will be abended.

After transaction TRAA receives a response
A syncpoint at this point causes CICS to commit a record in system B before it has been deleted from system A. If either system (or the connection between them) fails before the distributed process is completed, data may be duplicated.

Immediately after the record is deleted from the temporary storage queue

Because minimum processing is needed before resources are committed, this may be a safe place to take a syncpoint if the queue is long or the records are large. However, performance may be poor because a syncpoint is taken for each record transmitted.

After transaction TRAA receives the response to the last-record indicator

If you take a syncpoint only when all records have been transmitted, an earlier failure will mean that all data will have to be retransmitted. A distributed process that syncpoints only at this stage will complete more quickly than one that syncpoints after each record is processed, provided no failure occurs. However, it will take longer to recover. If more than two systems are involved in the process, this problem is made worse.

Remember that too many conversations within one distributed transaction complicates error recovery. A complex structure may sometimes be unavoidable, but usually it means that the design could be improved if some thought is given to simplifying the structure of the distributed transaction.

A UOW must be recoverable for the whole process of which it forms a part. All changes made by both partners in every conversation must be backed out if the UOW does not complete successfully. Syncpoints are not arbitrary divisions, but must reflect the functions of the application. Units of work must be designed to preserve consistent resources so that when a transaction fails, **all** resources are restored to their correct state.

Before terminating a sync level-2 conversation, make sure that the partner transaction is able to communicate any errors that it may have found. Not doing so might jeopardize data integrity.

Designing conversations

Once the overall structure of the distributed process has been decided, you can then start to design individual conversations. Designing a conversation involves deciding what functions to put into the front-end transaction and into the back-end transaction, and deciding what should be in a distributed unit of work. So you have to make decisions about how to subdivide the work to be done for your application.

Because a conversation involves transferring data between two transactions, to function correctly, each transaction must know what the other intends. For instance, there is little point in the front-end transaction sending data if all the back-end transaction is designed to do is print the weekly sales report. You must therefore consider each front-end and back-end transaction pair as one software unit.

The sequences of commands you can issue on a conversation are governed by a protocol designed to ensure that commands are not issued in inappropriate circumstances. The protocol is based on the concept of a number of conversation states. A conversation state applies only to one side of a single conversation and not to a transaction as a whole. In each state, there are a number of commands that might reasonably be issued. The command itself, together with its outcome, may cause the conversation to change from one state to another.

To determine the conversation state, you can use either the STATE option on a command or the EXTRACT ATTRIBUTES STATE command. Note, however, that the STATE option is valid only for MRO and APPC sessions, not for LUTYPE6.1 sessions. For programming information about the state values returned by different commands, see CICS API commands.

When a conversation changes state, it is said to have undergone a **state transition**, which generally makes a different set of commands available. The available commands and state transitions are shown in a series of state tables. Which state table you use depends on the protocol, sync level, application programming interface (API), and conversation type that you choose. (Only the APPC protocol gives you a choice of APIs and conversation types.)

"Maintaining data integrity" on page 7 contains guidance on selecting the sync level for a conversation. Syncpointing a distributed process discusses the synchronization commands and their effects.

Selecting the protocol

CICS provides three different protocols that support distributed transaction processing. These protocols define the rules under which two transactions can communicate with each other.

The protocols are:

- **APPC** (advanced program-to-program communication, sometimes referred to as LUTYPE6.2)
- **MRO** (multiregion operation)
- **LUTYPE6.1** (logical unit type 6.1).

Both APPC and LUTYPE6.1 are protocols defined by SNA. They are therefore more widely available for communicating with non-CICS systems. LUTYPE6.1 is the predecessor of APPC; so you should, if possible, avoid using LUTYPE6.1 for new applications. However, some new applications may still need to use LUTYPE6.1 to communicate with existing LUTYPE6.1 applications.

To help you migrate applications from LUTYPE6.1 to APPC, CICS provides a migration path. For more information on this, see Migration of LUTYPE6.1 applications to APPC links.

Choosing between MRO and APPC can be quite simple. The options depend on the configuration of your CICS complex and on the nature of the conversation partner. MRO does not support communication with a partner in a non-CICS system. Further, it supports communication between transactions running in CICS systems in different MVS™ images only if the MVS images are in the same MVS sysplex, and are joined by cross-system coupling facility (XCF) links; the MVS images must be at IBM® MVS/ESA release level 5.1, or later. (For full details of the hardware and software requirements for XCF/MRO, see Installation requirements for XCF/MRO.)

For communication with a partner in another CICS system, where the CICS systems are either in the same MVS image, or in the same MVS/ESA 5.1 (or later) sysplex, you can use either the MRO or the APPC protocol. There are good performance reasons for using MRO. But if there is any possibility that the distributed transactions will need to communicate with partners in other operating systems, it is better to use APPC so that the transaction remains unchanged.

APPC application programs will not run under MRO. Even if both partners are in the same MVS image, CICS will not use MRO facilities but will send conversation data through the communications controller. That involves some z/OS Communications Server overhead. So you must decide whether your application programs are to converse using APPC or MRO and code them accordingly.

Table 2 on page 11 points out the main differences between the MRO and APPC protocols.

Table 2. MRO protocol compared with APPC protocol

MRO	APPC
Function is realized without using a telecommunication access method.	Depends on z/OS Communications Server or similar.
Non-standard architecture.	SNA architecture.
CICS-to-CICS links only.	Links to non-CICS systems possible.
Communicates within single MVS image, or (using XCF/MRO) between MVS images in same sysplex.	Communicates across multiple MVS images or other operating systems.
Sync level 2 forced for the conversation.	Sync level 0, 1, or 2 can be selected.
Program initialization parameter (PIP) data not supported.	PIP data supported.
Data transmission not deferred.	Deferred data transmission.
Partner transaction may be identified in data.	Partner transaction defined by program command.
Performance overhead over a single application.	Even greater performance overhead over a single application.
RECEIVE can be issued only in receive state.	RECEIVE causes conversation turnaround when issued in send state on mapped conversations.
No ISSUE SIGNAL command.	ISSUE SIGNAL command available.

Table 2. MRO protocol compared with APPC protocol (continued)

MRO	APPC
WAIT command has no function.	WAIT command causes transmission of deferred data.

APPC protocol

If you choose to use APPC, you must decide which application programming interface (API) to use; and then which conversation type (basic or mapped) to use.

Selecting the APPC conversation type

APPC conversations can be either *mapped* or *basic*. For CICS-to-CICS applications, you can use mapped conversations. Basic, or *unmapped*, conversations are useful to communicate with systems that do not support mapped conversations. These systems include some APPC devices.

The two protocols are similar. The main difference is the way that user data is formatted for transmission. In mapped conversations, the application sends the data that you want the partner to receive. In basic conversations, the application must include additional control bytes to convert the data to an SNA-defined format called a *generalized data stream* (GDS). Also, in EXEC CICS commands for basic conversations, you must include the keyword GDS.

The following table summarizes the differences between mapped and basic conversations that apply to the CICS API.

Table 3. APPC conversations – mapped or basic?

Mapped	Basic
The conversation partners exchange data that is relevant only to the application.	Both partners must package the user data before sending, and unpackage it on receipt.
All conversations for a transaction share the same EXEC Interface Block for status reporting.	Each conversation has its own area for state information.
The transaction can handle exception conditions or let them default.	The transaction must test for exception conditions in a data area set aside for the purpose.
A RECEIVE command issued in send state causes conversation turnaround.	A RECEIVE command is illegal in send state.
Transactions can be written in any of the supported languages.	Transactions can be written in assembler language or C only.
You can cause a conversation to time out if the partner does not respond. To do this, you specify the RTIMOUT option of the PROFILE definition.	You cannot cause a conversation to time out if the partner does not respond.

Effect of z/OS Communications Server persistent sessions support for DTP conversations on APPC sessions

If you enable z/OS Communications Server persistent sessions support in the local CICS, after a CICS failure APPC sessions are held in recovery pending state until CICS restarts, or until the timeout value set on the PSDINT system initialization parameter expires. DTP applications that use APPC sessions defined as persistent are affected by persistent sessions recovery.

Remote partner programs can cause excessive queuing delays in the partner system if they continue to issue commands on persistent APPC sessions after this CICS has failed. There is no way for the partner to know that persistent sessions recovery is in progress. However, there are various actions you can take to reduce the risk of new work building up for a connection to a persisting CICS system.

Actions on the partner system:

- In DTP applications, requests for sessions are instigated by EXEC CICS ALLOCATE commands. Control the overall number of queued session requests by using:
  - The QUEUELIMIT and MAXQTIME options on the CONNECTION definition
  - An XZIQUE global user exit program.

  These methods are described in Managing allocate queues.
- Control individual session requests by coding the NOQUEUE|NOSUSPEND option on EXEC CICS ALLOCATE commands.
- Force mapped APPC RECEIVE or CONVERSE commands to time out if there is any delay in receiving expected data, by coding the RTIMOUT option on PROFILE definitions.

Action on this system:

- Code a PSDINT value that takes into account the number of your APPC sessions to partner systems.

After a restart, LU6.2 session names, in the range -AAA to -999, are allocated on a "first free" basis (rather than on a "next in the sequence" followed by "last free" basis). This may affect applications that use LU6.2 CONVIDs as external qualifiers.

For further information about z/OS Communications Server persistent sessions support, see Recovery with z/OS Communications Server persistent sessions.

What is a conversation and what makes it necessary?

In DTP, transactions pass data to each other directly. While one sends, the other receives. The exchange of data between two transactions is called a **conversation**.

Although several transactions can be involved in a single distributed process, communication between them breaks down into a number of self-contained conversations between pairs. Each such conversation uses a CICS resource known as a **session**.

Conversation initiation and transaction hierarchy

A transaction starts a conversation by requesting the use of a session to a remote system. Having obtained the session, it causes an attach request to be sent to the other system to activate the transaction that is to be the conversation partner.

A transaction can initiate any number of other transactions, and hence, conversations. In a complex process, a distinct hierarchy emerges, with the terminal-initiated transaction at the very top. Figure 3 on page 14 shows a possible configuration. Transaction TRAA is attached over the terminal session. Transaction TRAA attaches transaction TRBB, which, in turn, attaches transactions TRCC and TRDD. Both these transactions attach the same transaction, SUBR, in system CICSE. This gives rise to two different tasks of SUBR.

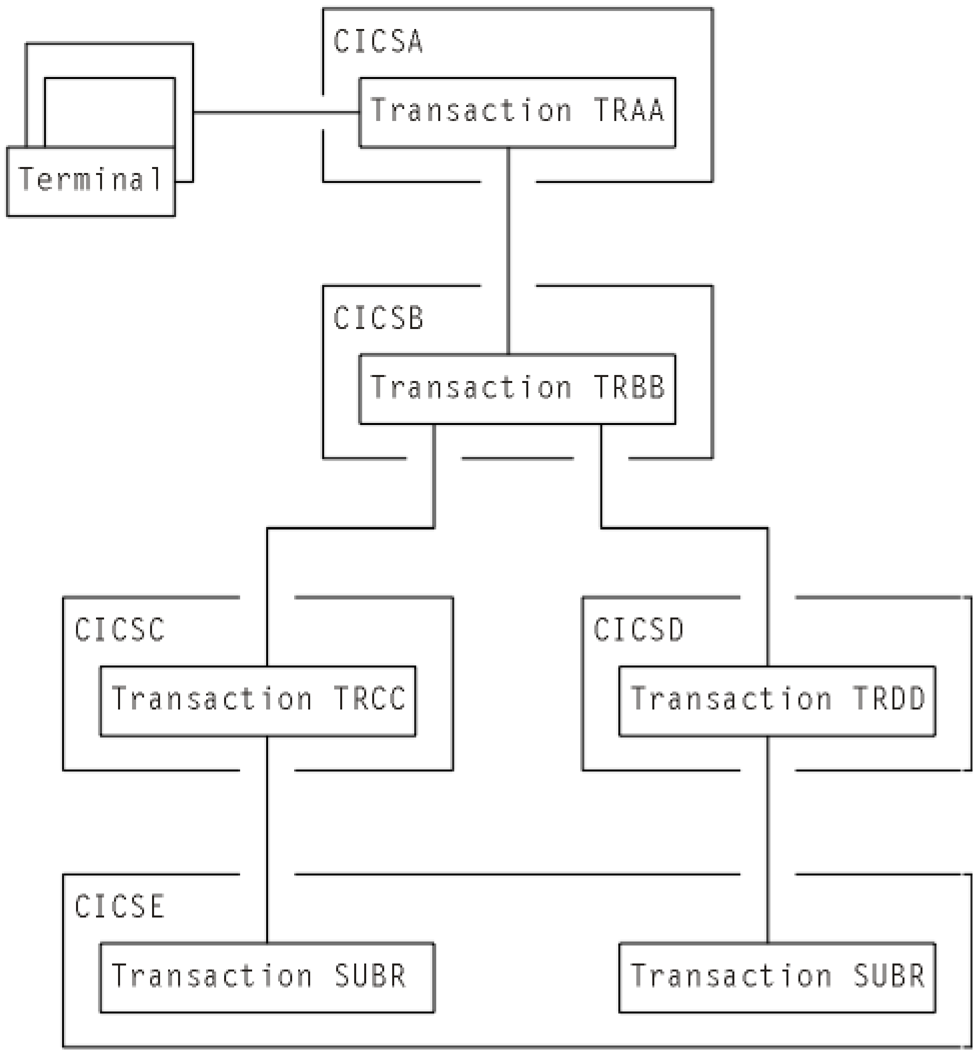

Figure 3. DTP in a multisystem configuration

The structure of a distributed process is determined dynamically by program; it cannot be predefined. Notice that, for every transaction, there is only one inbound attach request, but there can be any number of outbound attach requests. The session that activates a transaction is called its **principal facility**. A session that is allocated by a transaction to activate another transaction is called its **alternate facility**. Therefore, a transaction can have only one principal facility, but any number of alternate facilities.

When a transaction initiates a conversation, it is the **front end** on that conversation. Its conversation partner is the **back end** on the same conversation. (Some books refer to the front end as the initiator and the back end as the recipient.) It is normally the front end that dominates, and determines the way the conversation goes. You can arrange for the back end to take over if you want, but, in a complex process, this can cause unnecessary complication. This is further explained in the discussion on synchronization later in this chapter.

Dialog between two transactions

A conversation transfers data from one transaction to another.

For this to function properly, each transaction must know what the other intends. It would be nonsensical for the front end to send data if all the back end wants to do is print out the weekly sales report. It is therefore necessary to design, code, and test front end and back end as one software unit. The same applies when there are several conversations and several transaction programs. Each new conversation adds to the complexity of the overall design.

In the example in “Advantages over function shipping and transaction routing” on page 1, the DTP solution is to transmit the contents of the transient data queue from the front end to the back end. The front end issues a SEND command for each record that it takes off the queue. The back end issues RECEIVE commands until it receives an indication that the transmission has ended.

In practice, most conversations transfer a file of data from one transaction to another. The next stage of complexity is to cause the back end to return data to the front end, perhaps the result of some processing. Here the front end is programmed to request conversation turnaround at the appropriate point.

Control flows and brackets

During a conversation, data passes over the link in both directions.

A single transmission is called a **flow**. Issuing a SEND command does not always cause a flow. This is because the transmission of user data can be deferred; that is, held in a buffer until some event takes place. The APPC architecture defines data formats and packaging. CICS handles these things for you, and they concern you only if you need to trace flows for debugging.

The APPC architecture defines a data header for each transmission, which holds information about the purpose and structure of the data following. The header also contains bit indicators to convey control information to the other side. For example, if one side wants to tell the other that it can start sending, CICS sets a bit in the header that signals a change of direction in the conversation.

To keep flows to a minimum, non-urgent control indicators are accumulated until it is necessary to send user data, at which time they are added to the header.

For the formats of the headers and control indicators used by APPC, see Systems Network Architecture Formats (GA27-3136).

In complex procedures, such as establishing syncpoints, it is often necessary to send control indicators when there is no user data available to send. This is called a **control flow**.

begin_bracket marks the start of a conversation; that is, when a transaction is attached. conditional_end_bracket ends a conversation. End bracket is conditional because the conversation can be reopened under some circumstances. A conversation is **in bracket** when it is still active.

MRO is not unlike APPC in its internal organization. It is based on LUTYPE6.1, which is also an SNA-defined architecture.

Conversation state and error detection

As a conversation progresses, it moves from one state to another within both conversing transactions.

The conversation state determines the commands that may be issued. For example, it is no use trying to send or receive data if there is no session linking the front end to the back end. Similarly, if the back end signals end of conversation, the front end cannot receive any more data on the conversation.

Either end of the conversation can cause a change of state, usually by issuing a particular command from a particular state. CICS tracks these changes, and stops transactions from issuing the wrong command in the wrong state.

Synchronization

There are many things that can go wrong during the running of a transaction. The conversation protocol helps you to recover from errors and ensures that the two sides remain in step with each other. This use of the protocol is called **synchronization**.

Synchronization allows you to protect resources such as transient data queues and files. If anything goes wrong during the running of a transaction, the associated resources should not be left in an inconsistent state.

Examples of use

Suppose, for example, that a transaction is transmitting a queue of data to another system to be written to a DASD file. Suppose also that for some reason, not necessarily connected with the intercommunication activity, the receiving transaction is abended.

Even if a further abend can be prevented, there is the problem of how to continue the process without loss of data. It is uncertain how many queue items have been received and how many have been correctly written to the DASD file. The only safe way of continuing is to go back to a point where you know that the contents of the queue are consistent with the contents of the file. However, you then have two problems. On one side, you need to restore the queue entries that you have sent; on the other side, you need to delete the corresponding entries in the DASD file.

The cancelation by an application program of all changes to recoverable resources since the last known consistent state is called **rollback**. The physical process of recovering resources is called **backout**. The condition that exists as long as there is no loss of consistency between distributed resources is called **data integrity**.

There are cases in which you may want to recover resources, even though there are no error conditions. Consider an order entry system. While entering an order for a customer, an operator is told by the system that the customer's credit limit would be exceeded if the order went through. Because there is no use continuing until the customer is consulted, the operator presses a function key to abandon the order. The transaction is programmed to respond by restoring the data resources to the state they were in at the start of the order.

Taking syncpoints

If you were to log your own data movements, you could arrange backout of your files and queues.

However, it would involve some very complex programming, which you would have to repeat for every similar application. To save you this overhead, CICS arranges resource recovery for you. LU management works with resource management in ensuring that resources can be restored.

The points in the process where resources are declared to be in a known consistent state are called **synchronization points**, often shortened to **syncpoints**. Syncpoints are implied at the beginning and end of a transaction. A transaction can define other syncpoints by program command. All processing between two consecutive syncpoints belongs to a **unit of work** (UOW).

Taking a syncpoint **commits** all recoverable resources. This means that all systems involved in a distributed process erase all the information they have been keeping about data movements on recoverable resources. Now backout is no longer possible, and all changes to the resources since the last syncpoint are made irreversible.

Although CICS commits and backs out changes to resources for you, the service must be paid for in performance. You might have transactions that do not need such complexity, and it would be wasteful to employ it. If the recovery of resources is not a problem, you can use simpler methods of synchronization.

The three sync levels

The APPC architecture defines three levels of synchronization (called **sync levels**).

- Level 0 – none
- Level 1 – confirm
- Level 2 – syncpoint

At sync level 0, there is no system support for synchronization. It is nevertheless possible to achieve some degree of synchronization through the interchange of data, using the SEND and RECEIVE commands.

If you select sync level 1, you can use special commands for communication between the two conversation partners. One transaction can *confirm* the continued presence and readiness of the other. The user is responsible for preserving the data integrity of recoverable resources.

The level of synchronization described earlier in this section corresponds to sync level 2. Here, system support is available for maintaining the data integrity of recoverable resources.

CICS implies a syncpoint when it starts a transaction; that is, it initiates logging of changes to recoverable resources, but no control flows take place. CICS takes a full syncpoint when a transaction is normally terminated. Transaction abend causes rollback. The transactions themselves can initiate syncpoint or rollback requests. However, a syncpoint or rollback request is propagated to another transaction only when the originating transaction is in conversation with the other transaction, and if sync level 2 has been selected for the conversation between them.

Remember that syncpoint and rollback are not peculiar to any one conversation within a transaction. They are propagated on every sync level 2 conversation that is currently *in bracket*.

MRO or APPC for DTP?

You can program DTP applications for both MRO and APPC links. The two conversation protocols are not identical. Although you seldom have the choice for a particular application, an awareness of the differences and similarities will help you to make decisions about compatibility.

Choosing between MRO and APPC can be quite simple. The options depend on the configuration of your CICS complex and on the nature of the conversation partner. You cannot use MRO to communicate with a partner in a non-CICS system. Further, it supports communication between transactions running in CICS systems in different MVS images only if the MVS images are in the same MVS sysplex, and are joined by cross-system coupling facility (XCF) links. For full details of the hardware and software requirements for XCF/MRO, see Installation requirements for XCF/MRO.

For communication with a partner in another CICS system, where the CICS systems are either in the same MVS image, or in the same sysplex, you can use either the MRO or the APPC protocol. There are good performance reasons for using MRO. But if there is any possibility that the distributed transactions will need to communicate with partners in other operating systems, it is better to use APPC so that the transaction remains unchanged.

Table 4 on page 17 summarizes the main differences between the two protocols.

Table 4. MRO compared with APPC

MRO	**APPC**
Function is realized within CICS	Depends on the z/OS Communications Server or similar
Nonstandard architecture	SNA architecture
CICS-to-CICS links only	Links to non-CICS systems possible
Communicates within single MVS image, or (using XCF/MRO) between MVS images in same sysplex	Communicates across multiple MVS images and other operating systems
PIP data not supported	PIP data supported
Data transmission not deferred	Deferred data transmission
Partner transaction identified in data	Partner transaction defined by program command
RECEIVE can only be issued in receive state	RECEIVE causes conversation turnaround when issued in send state on mapped conversations
No expedited flow possible	ISSUE SIGNAL command flows expedited
WAIT command has no function	WAIT command causes transmission of deferred data

APPC mapped or basic?

APPC conversations can be either *mapped* or *basic*. For CICS-to-CICS applications, you can use mapped conversations. Basic, or *unmapped*, conversations are useful to communicate with systems that do not support mapped conversations. These systems include some APPC devices.

The two protocols are similar. The main difference is the way that user data is formatted for transmission. In mapped conversations, the application sends the data that you want the partner to receive. In basic conversations, the application must include additional control bytes to convert the data to an SNA-defined format called a *generalized data stream* (GDS). Also, in EXEC CICS commands for basic conversations, you must include the keyword GDS.

Table 5 on page 18 summarizes the differences between mapped and basic conversations that apply to the CICS API.

CPI Communications has different rules (see "EXEC CICS or CPI Communications?" on page 18.

Table 5. APPC conversations – mapped or basic?

Mapped	**Basic**
The conversation partners exchange data that is relevant only to the application.	Both partners must package the user data before sending, and unpackage it on receipt.
All conversations for a transaction share the same EXEC Interface Block for status reporting.	Each conversation has its own area for state information.
The transaction can handle exception conditions or let them default.	The transaction must test for exception conditions in a data area set aside for the purpose.
A RECEIVE command issued in send state causes conversation turnaround.	A RECEIVE command is illegal in send state.
Transactions can be written in any of the supported languages.	Transactions can be written in assembler language or C only.
You can cause a conversation to time out if the partner does not respond. To do this, you specify the RTIMOUT option of the PROFILE definition.	You cannot cause a conversation to time out if the partner does not respond.

EXEC CICS or CPI Communications?

CICS gives you a choice of two application programming interfaces (APIs) for coding your DTP conversations on APPC sessions.

The first, the **CICS API**, is the programming interface of the CICS implementation of the APPC architecture. It consists of EXEC CICS commands and can be used with all CICS-supported languages. The second, **Common Programming Interface Communications** (CPI Communications) is the communication interface defined for the SAA environment. It consists of a set of defined verbs, in the form of program calls, which are adapted for the language being used.

Table 6 on page 18 compares the two methods to help you to decide which API to use for a particular application.

Table 6. CICS API compared with CPI Communications

CICS API	**CPI Communications**
Portability between different members of the CICS family.	Portability between systems that support SAA facilities.

Table 6. CICS API compared with CPI Communications (continued)

CICS API	**CPI Communications**
Basic conversations can be programmed only in assembler language or C.	Basic conversations can be programmed in any of the available languages.
Sync levels 0, 1, and 2 supported.	Sync levels 0, 1, and 2 supported, *except for transaction routing, for which only sync levels 0 and 1 are supported*.
PIP data supported.	PIP data not supported.
Only a few conversation characteristics are programmable. The rest are defined by resource definition.	Most conversation characteristics can be changed dynamically by the transaction program.
Can be used on the principal facility to a transaction started by ATI.	Cannot be used on the principal facility to a transaction started by ATI.
Limited compatibility with MRO.	No compatibility with MRO.

You can mix CPI Communications calls and EXEC CICS commands in the same transaction, but not on the same side of the same conversation. You can implement a distributed transaction where one partner to a conversation uses CPI Communications calls and the other uses the CICS API. In such a case, it would be up to you to ensure that the APIs on both sides map consistently to the APPC architecture.

Chapter 2. Writing programs for APPC mapped conversations

These topics describe the CICS APIs available for DTP programming using APPC mapped conversations.

Conversation initiation

The front-end transaction is responsible for acquiring a session, specifying the conversation characteristics and requesting the startup of the back-end transaction in the remote system.

Allocating a session to the conversation

Initially, there is no conversation, and therefore no conversation state. By issuing an ALLOCATE command, the front-end transaction acquires a session to start a new conversation.

The RESP value returned should be checked to ensure that a session has been allocated. If the session is successfully allocated, DFHRESP(NORMAL), the conversation is in **allocated state** (state 1) and the session identifier (**convid**) in EIBRSRCE must be saved immediately.

The convid must be used in subsequent commands for this conversation. Figure 4 on page 22 shows an example of an ALLOCATE command.

Note: If the remote system is using z/OS Communications Server persistent session support, you may need to code a timeout value on the ALLOCATE command. See Effect of z/OS Communications Server persistent sessions support for DTP conversations on APPC sessions.

Using ATI to allocate a session

Front-end transactions are often initiated from terminals. But it is also possible to use the **`EXEC CICS START`** command to initiate a front-end transaction on an APPC session.

When this is done, and the front-end transaction is successfully started, a conversation can continue as if an ALLOCATE command had been issued. The only difference is that, when ATI is used, the APPC session is the front-end transaction's principal facility.

```
 * ...
DATA DIVISION.
WORKING-STORAGE SECTION.
* ...
01 FILLER.
02 WS-CONVID PIC X(4).
02 WS-RESP PIC S9(8) COMP.
02 WS-STATE PIC S9(8) COMP.
02 WS-SYSID PIC X(4) VALUE 'SYSB'.
02 WS-PROC PIC X(4) VALUE 'BBBB'.
02 WS-LEN-PROCN PIC S9(4) COMP VALUE +4.
02 WS-SYNC-LVL PIC S9(4) COMP VALUE +2.
* ...
PROCEDURE DIVISION.
* ...
EXEC CICS ALLOCATE SYSID(WS-SYSID) RESP(WS-RESP)
END-EXEC.
IF WS-RESP = DFHRESP(NORMAL)
THEN MOVE EIBRSRCE TO WS-CONVID
ELSE
* ... No session allocated. Examine RESP code.
END-IF.
* ...
EXEC CICS CONNECT PROCESS CONVID(WS-CONVID)
STATE(WS-STATE) RESP(WS-RESP)
PROCNAME(WS-PROC)
PROCLENGTH(WS-LEN-PROCN)
SYNCLEVEL(WS-SYNC-LVL)
END-EXEC.
IF WS-RESP = DFHRESP(NORMAL)
THEN
* ... No errors. Check EIB flags.
ELSE
* ... Conversation not started. Examine RESP code.
END-IF.
```

Figure 4. Starting an APPC mapped conversation at sync level 2

Connecting the partner transaction

When the front-end transaction has acquired a session, the next step is to initiate the partner transaction.

The state tables show that, in the **allocated state** (state 1), one of the commands available is CONNECT PROCESS. This command is used to attach the required back-end transaction. It should be noted that the results of the CONNECT PROCESS are placed in the send buffer and are not sent immediately to the partner system. Transmission occurs when the send buffer is flushed, either by sending more data than fits in the send buffer or by issuing a WAIT CONVID command.

A successful CONNECT PROCESS causes the conversation to switch to **send state** (state 2). The program fragment in Figure 4 on page 22 shows an example of a CONNECT PROCESS command.

Note: For clarity, the **EXEC CICS ALLOCATE** and **CONNECT PROCESS** commands shown in Figure 4 on page 22 identify the partner LU and transaction explicitly. To avoid doing this, you could use the PARTNER option of these commands. This specifies a set of definitions that include the names of the partner LU, the communication profile to be used on the session, and the partner transaction. Thus, in Figure 4 on page 22 , the PARTNER option could be used instead of SYSID on the **EXEC CICS ALLOCATE** command, and instead of PROCNAME and PROCLENGTH on the **EXEC CICS CONNECT PROCESS** command. The advantage of using PARTNER is that it makes your DTP programs more maintainable: the details of each partner program can be held in a single definition.

Initial data for the back-end transaction

While connecting the back-end transaction, the front-end transaction can send initial data to it. This kind of data, called *program initialization parameters* (PIPs), is placed in specially formatted structures and specified on the CONNECT PROCESS command. The PIPLIST (along with PIPLENGTH) option of the CONNECT PROCESS command is used to send PIPs to the back-end transaction.

To examine any PIPs received, the back-end transaction uses the EXTRACT PROCESS command.

PIP data is used only by the two connected transactions and not by the CICS systems. APPC systems other than CICS may not support PIP, or may support it differently.

The PIP data must be formatted into one or more subfields according to the SNA rules. The content of each subfield is defined by the application developer. You should format PIP data as follows:

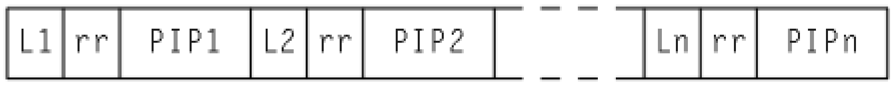

Figure 5. Format of PIP data

CICS inserts information into the reserved fields to make the PIP architecturally correct. The PIPLENGTH option must specify the total length of the PIP list and must be between 4 and 32763.

Back-end transaction initiation

The back-end transaction is initiated as a result of the front end transaction’s CONNECT PROCESS command.

Initially, the back-end transaction should determine the convid. This is not strictly necessary because the session is the back-end transaction’s principal facility making the CONVID parameter optional for DTP commands on this conversation. However, the convid is useful for audit trails. Also, if the back-end transaction is involved in more than one conversation, always specifying the CONVID option improves program readability and problem determination.

Figure 6 on page 24 shows a fragment of a back-end transaction that obtains the conversation identifier. The example uses the ASSIGN command for this purpose; another way is to access the information in EIBTRMID.

The back-end transaction can also retrieve its transaction name by issuing the EXTRACT PROCESS command. In the example shown in Figure 6 on page 24 , CICS places the transaction name in WS-PROC and the length of the name in WS-LEN-PROCN. With the EXTRACT PROCESS, the back-end transaction can also retrieve the sync level at which the conversation was started. In the example, CICS places the sync level in WS-SYNC-LVL.

Both the ASSIGN and the EXTRACT PROCESS commands are discussed here only to give you some idea of what you can do in the back-end transaction. They are not essential. The back-end transaction starts in **receive state** (state 5), and must issue a RECEIVE command. By doing this, the back-end transaction receives whatever data the front-end transaction has sent and allows CICS to raise EIB flags and change the conversation state to reflect any request the front-end transaction has issued.

```
 * ...
DATA DIVISION.
WORKING-STORAGE SECTION.
* ...
01 FILLER.
02 WS-CONVID PIC X(4).
02 WS-STATE PIC S9(7) COMP.
02 WS-SYSID PIC X(4) VALUE 'SYSB'.
02 WS-PROC PIC X(4) VALUE 'BBBB'.
02 WS-LEN-PROCN PIC S9(4) COMP VALUE +4.
02 WS-SYNC-LVL PIC S9(4) COMP VALUE +2.
* ...
01 FILLER.
02 WS-RECORD PIC X(100).
02 WS-MAX-LEN PIC S9(4) COMP VALUE +100.
02 WS-RCVD-LEN PIC S9(4) COMP VALUE +0.
* ...
PROCEDURE DIVISION.
* ...
EXEC CICS ASSIGN FACILITY(WS-CONVID) END-EXEC.
* ...
* Extract the conversation characteristics.
*
EXEC CICS EXTRACT PROCESS PROCNAME(WS-PROC)
PROCLENGTH(WS-LEN-PROCN)
SYNCLEVEL(WS-SYNC-LVL)
END-EXEC.
* ...
* Receive data from the front-end transaction.
*
EXEC CICS RECEIVE CONVID(WS-CONVID) STATE(WS-STATE)
INTO(WS-RECORD) MAXLENGTH(WS-MAX-LEN)
NOTRUNCATE LENGTH(WS-RCVD-LEN)
END-EXEC.
*
* ... Check outcome of RECEIVE.
* ...
```

Figure 6. Startup of a back-end APPC mapped transaction at sync level 2

What happens if the back-end transaction fails to start

It is possible that the back-end transaction fails to start. However there is a transmission delay mechanism in APPC, which informs the front-end transaction of this fact when the session has been active long enough for responses from the back-end system to have been received.

The front-end transaction is informed of the failure with a TERMERR condition in response to a DTP command. EIBERR, EIBFREE, and EIBERRCD are set (see Table 11 on page 32 for the possible values of EIBERRCD).

Before sending data, the front-end transaction should find out whether the back-end transaction has started successfully. One way of doing this is to issue a SEND CONFIRM command directly after the CONNECT PROCESS command. This causes the front-end transaction to suspend until the back-end transaction responds or the failure notification, as described, is received. SEND CONFIRM is discussed in "How to synchronize a conversation using CONFIRM commands" on page 29.

Transferring data on the conversation

These topics discuss how to pass data between the front- and back-end transactions, and provide a program fragment illustrating the commands described and the suggested response code checking.

Sending data to the partner transaction

Data is sent to the partner transaction using the **SEND** command.

The SEND command is valid only in **send state** (state 2). Because a successful simple SEND leaves the conversation in **send state** (state 2), it is possible to issue a number of successive sends. The data from the simple SEND command is initially stored in a local CICS buffer which is "flushed" either when this buffer is full or when the transaction requests transmission. The transaction can request transmission either by using a WAIT CONVID command or by using the WAIT option on the SEND command. The reason data transmission is deferred is to reduce the number of calls to the network. However, the application should use WAIT if the partner transaction requires the data to continue processing.

An example of a simple SEND command can be seen in Figure 7 on page 25.

```
 * ...
DATA DIVISION.
WORKING-STORAGE SECTION.
* ...
01 FILLER.
02 WS-CONVID PIC X(4).
02 WS-STATE PIC S9(7) COMP.
* ...
01 FILLER.
02 WS-SEND-AREA PIC X(70).
02 WS-SEND-LEN PIC S9(4) COMP VALUE +70.
* ...
01 FILLER.
02 WS-RCVD-AREA PIC X(100).
02 WS-MAX-LEN PIC S9(4) COMP VALUE +100.
02 WS-RCVD-LEN PIC S9(4) COMP VALUE +0.
* ...
PROCEDURE DIVISION.
* ...
EXEC CICS SEND CONVID(WS-CONVID) STATE(WS-STATE)
FROM(WS-SEND-AREA) LENGTH(WS-SEND-LEN)
END-EXEC.
* ... Check outcome of SEND.
* ...
*
EXEC CICS SEND CONVID(WS-CONVID) STATE(WS-STATE)
INVITE WAIT
END-EXEC.
* ...
* Receive data from the partner transaction.
*
EXEC CICS RECEIVE CONVID(WS-CONVID) STATE(WS-STATE)
INTO(WS-RCVD-AREA) MAXLENGTH(WS-MAX-LEN)
NOTRUNCATE LENGTH(WS-RCVD-LEN)
END-EXEC.
*
* ... Check outcome of RECEIVE.
* ...
```

Figure 7. Transferring data on a conversation at sync level 2

Switching from sending to receiving data

There are several ways of switching from **send state** to **receive state** .

One possibility is to use a RECEIVE command. The state tables show that CICS supplies the INVITE and WAIT when a SEND is followed immediately by a RECEIVE.

Another possibility is to use a SEND INVITE command. The state tables show that after SEND INVITE the conversation switches to **pendreceive state** (state 3). The column for state 3 shows that a WAIT CONVID command switches the conversation to **receive state** (state 5).

Still another possibility is to specify the INVITE and WAIT options on the SEND command. The state tables show that after SEND INVITE WAIT, the conversation switches to **receive state** (state 5).

An example of a SEND INVITE WAIT command can be seen in Figure 7 on page 25. Figure 8 on page 26 illustrates the response-testing sequence after a SEND INVITE WAIT with the STATE option. For more information on response testing, see “Checking the outcome of a DTP command” on page 32.

```
 * ...
DATA DIVISION.
WORKING-STORAGE SECTION.
* ...
01 FILLER.
02 WS-RESP PIC S9(7) COMP.
02 WS-STATE PIC S9(7) COMP.
* ...
PROCEDURE DIVISION.
* ...
* Check return code from SEND INVITE WAIT
IF WS-RESP = DFHRESP(NORMAL)
THEN
* ... Request successful
IF EIBERR = LOW-VALUES
THEN
* ... No errors, check state
IF WS-STATE = DFHVALUE(RECEIVE)
THEN
* ... SEND OK, continue processing
ELSE
* ... Logic error, should never happen
END-IF
ELSE
* ... Error indicated
EVALUATE WS-STATE
WHEN DFHVALUE(ROLLBACK)
* ... ROLLBACK received
WHEN DFHVALUE(RECEIVE)
* ... ISSUE ERROR received, reason in EIBERRCD
WHEN OTHER
* ... Logic error, should never happen
END-EVALUATE
END-IF
ELSE
* ... Examine RESP code for source of error.
END-IF.
```

Figure 8. Checking the outcome of a SEND INVITE WAIT command

Receiving data from the partner transaction

The RECEIVE command is used to receive data from the connected partner.

The rows in the state tables for the RECEIVE command show the EIB fields that should be tested after issuing a RECEIVE command. As well as showing which field should be tested, the state tables also show the order in which the tests should be made.

As an alternative to testing the EIB fields it is possible to test the resulting conversation state; this is shown in Figure 9 on page 27. The conversation state can be meaningfully tested only after issuing a command with the STATE option or by using the EXTRACT ATTRIBUTES STATE command. Note that the RESP value returned and EIBERR should always be tested. If EIBNODAT is set on (X'FF'), no data has been received. For more information about response testing, see "Checking the outcome of a DTP command" on page 32 . For information about testing the conversation state, see "Testing the conversation state" on page 36.

An example of a RECEIVE command with the STATE option can be seen in Figure 7 on page 25. Figure 9 on page 27 illustrates the response-testing and state-testing sequence.

Note: In the same way as it is possible to send the INVITE, LAST, and CONFIRM commands with data, it is also possible to receive them with data. It is also possible to receive a syncpoint request with data. However, ISSUE ERROR, ISSUE ABEND, and conversation failure are never received with data.

```
 * ...
WORKING-STORAGE SECTION.
* ...
01 FILLER.
02 WS-RESP PIC S9(8) COMP.
02 WS-STATE PIC S9(8) COMP.
* ...
PROCEDURE DIVISION.
* ...
* Check return code from RECEIVE
IF WS-RESP = DFHRESP(EOC)
OR WS-RESP = DFHRESP(NORMAL)
THEN
* ... Request successful
IF EIBERR = LOW-VALUES
THEN
* ... No errors, check state
EVALUATE WS-STATE
WHEN DFHVALUE(SYNCFREE)
* ... Partner issued SYNCPOINT and LAST
WHEN DFHVALUE(SYNCRECEIVE)
* ... Partner issued SYNCPOINT
WHEN DFHVALUE(SYNCSEND)
* ... Partner issued SYNCPOINT and INVITE
WHEN DFHVALUE(CONFFREE)
* ... Partner issued CONFIRM and LAST
WHEN DFHVALUE(CONFRECEIVE)
* ... Partner issued CONFIRM
WHEN DFHVALUE(CONFSEND)
* ... Partner issued CONFIRM and INVITE
WHEN DFHVALUE(FREE)
* ... Partner issued LAST or FREE
WHEN DFHVALUE(SEND)
* ... Partner issued INVITE
WHEN DFHVALUE(RECEIVE)
* ... No state change. Check EIBCOMPL.
WHEN OTHER
* ... Logic error, should never happen
END-EVALUATE.
ELSE
* ... Error indicated
EVALUATE WS-STATE
WHEN DFHVALUE(ROLLBACK)
* ... ROLLBACK received
WHEN DFHVALUE(RECEIVE)
* ... ISSUE ERROR received, reason in EIBERRCD
WHEN OTHER
* ... Logic error, should never happen
END-EVALUATE
END-IF
ELSE
* ... Examine RESP code for source of error
END-IF.
```

Figure 9. Checking the outcome of a RECEIVE command

The CONVERSE command

The CONVERSE command combines the functions SEND INVITE WAIT and RECEIVE. This command is useful when one transaction needs a response from the partner transaction to continue processing.

Communicating errors across a conversation

The APPC mapped API provides commands to enable transactions to pass error notification across a conversation.

There are three commands depending on the severity of the error. The most severe, ISSUE ABEND, causes the conversation to terminate abnormally and is described in "Emergency termination of a conversation" on page 31 .

Requesting INVITE from the partner transaction

If a transaction is receiving data on a conversation and wants to send, it can use the ISSUE SIGNAL command to request that the partner transaction does a SEND INVITE.

W hen the ISSUE SIGNAL request is received, EIBSIG= X'FF' and the SIGNAL condition is raised. It should be noted that on receipt of SIGNAL a transaction is **not** obliged to issue SEND INVITE.

Demanding INVITE from the partner transaction

If a transaction needs to send an immediate error notification to the partner transaction it can use the ISSUE ERROR command.

This command is also one of the preferred negative responses to SEND CONFIRM. However it should **not** be used to reject ISSUE PREPARE, SYNCPOINT or SYNCPOINT ROLLBACK. When the ISSUE ERROR is received, EIBERR= X'FF' and the first two bytes of EIBERRCD are X'0889' . This error condition cannot be processed by HANDLE CONDITION (or RESP).

If an ISSUE ERROR command is used in **receive state** (state 5), all incoming data is purged until an INVITE, SYNCPOINT, or LAST command is received. If LAST is received, no error indication is sent to the partner transaction, EIBFREE= X'FF' and the conversation is switched to **free state** (state 12).

If LAST is not received, the conversation is switched to **send state** (state 2). It is normal programming practice to communicate the reason for the ISSUE ERROR to the partner transaction. The CONVERSE command could be used to send an appropriate error message and receive a reply.

Because ISSUE ERROR is allowed in both **send state** (state 2) and **receive state** (state 5), it is possible for both communicating transactions to use ISSUE ERROR at the same time. When this occurs, only one of the ISSUE ERROR commands is effective. The other is purged with incoming data. However both ISSUE ERROR commands will appear to have completed successfully and the transaction whose ISSUE ERROR was purged will pick up EIBERR= X'FF' on a subsequent command.

Safeguarding data integrity

If it is important to safeguard data integrity across connected transactions, then the CICS synchronization commands should be used.

The commands shown in Table 7 on page 28 are available.

Table 7. Synchronization commands for APPC mapped conversations

Conversation sync level	**Commands**
0	None
1	SEND CONFIRM ISSUE CONFIRMATION
2	SEND CONFIRM ISSUE CONFIRMATION SYNCPOINT ISSUE PREPARE SYNCPOINT ROLLBACK SAA verbs: SRRCMIT SRRBACK

How to synchronize a conversation using CONFIRM commands

The RECEIVE command is used to receive data from the connected partner.

A confirmation exchange affects a single specified conversation and involves two commands.

1. The conversation that is in **send state** (state 2) issues a SEND CONFIRM command causing a request for confirmation to be sent to the partner transaction. The transaction suspends awaiting a response.
2. The partner transaction receives a request for confirmation. It can then respond positively by issuing an ISSUE CONFIRMATION command. Alternatively, it can respond negatively by using the ISSUE ERROR or ISSUE ABEND commands.

Requesting confirmation

The CONFIRM option of the SEND command flushes the conversation send buffer; that is, it causes a transmission to occur. When the conversation is in **send state** (state 2), you can send data with the SEND CONFIRM command. You can also specify either the INVITE or the LAST option.

The **send state** (state 2) column of the state table for APPC mapped conversations at sync level 1 (see "State tables for APPC mapped conversations at sync level 1" on page 39) shows what happens for the possible combinations of the CONFIRM, INVITE, and LAST options. After a SEND CONFIRM command, without the INVITE or LAST options, the conversation remains in **send state** (state 2). If the INVITE option is used, the conversation switches to **receive state** (state 5). If the LAST option is used, the conversation switches to **free state** (state 12).

A similar effect to SEND LAST CONFIRM can by achieved by using the command sequence:

```
 SEND LAST
SEND CONFIRM
```

Note from the state tables that the SEND LAST puts the conversation into **pendfree state** (state 4), so data cannot be sent with a SEND CONFIRM command used in this way.

The form of command used depends on how the conversation is to continue if the required confirmation is received. However, the response from SEND CONFIRM **must** always be checked. See "Checking the response to SEND CONFIRM" on page 30.

Receiving and replying to a confirmation request

On receipt of a confirmation request, the EIB and conversation state will be set depending on the request issued by the partner transaction.

The EIB, the conversation state, and the contents of the EIBCONF, EIBRECV, and EIBFREE fields are shown in Table 8 on page 29.

Table 8. Indications of a confirmation request

Command issued by partner transaction	**Conversation state on receipt of request**	**EIBCONF on receipt of request**	**EIBRECV on receipt of request**	**EIBFREE on receipt of request**
SEND CONFIRM	confreceive (state 6)	X'FF'	X'FF'	X'00'
SEND INVITE CONFIRM	confsend (state 7)	X'FF'	X'00'	X'00'
SEND LAST CONFIRM	conffree (state 8)	X'FF'	X'00'	X'FF'

There are three ways of replying:

1. Reply positively with an ISSUE CONFIRMATION command.
2. Reply negatively with an ISSUE ERROR command. This reply puts the conversation into **send state** (state 2) regardless of the partner transaction request.
3. Abnormally end the conversation with an ISSUE ABEND command. This makes the conversation unusable and a FREE command must be issued immediately.

Checking the response to SEND CONFIRM

After issuing SEND [INVITE|LAST] CONFIRM, it is important to test EIBERR to determine the partner's response.

Table 9 on page 30 shows how the partner's response is indicated by EIB flags and the conversation states.

Table 9. Indications of responses to SEND CONFIRM

Command issued in reply by partner transaction	Conversation state on receipt of response	EIBERR on receipt of response	EIBFREE on receipt of response
ISSUE CONFIRMATION	dependent on original SEND [INVITE|LAST] CONFIRM request	X'00'	X'00'
ISSUE ERROR	receive (state 5)	X'FF'	X'00'
ISSUE ABEND	free (state 12)	X'FF'	X'FF'

If EIBERR=X'00', the partner has replied ISSUE CONFIRMATION.

If the partner replies ISSUE ERROR, this is indicated by EIBERR=X'FF ' and the first two bytes of EIBERRCD = X'0889'. When the partner replies ISSUE ERROR in response to SEND LAST CONFIRM, the LAST option is ignored and the conversation is **not** terminated. The conversation state is switched to **receive state** (state 5).

If the partner replies ISSUE ABEND, your transaction will be abended AZCH. In addition, EIBERR and EIBFREE are set, and the first two bytes of EIBERRCD= X'0864'. The conversation is switched to **free state** .

How to synchronize conversations using SYNCPOINT commands

Data synchronization (the SYNCPOINT and SYNCPOINT ROLLBACK commands) affects all connected conversations at sync level 2.

The use of these commands in DTP is described in "Syncpointing a distributed process" on page 97.

Ending the conversation

A conversation can end in two ways, unexpectedly, or under transaction control.

To end a conversation, one transaction issues a request for termination and the other receives this request. Once this has happened the conversation is unusable and both transactions must issue a FREE command to release the session.

Normal termination of a conversation

The SEND LAST command is used to terminate a conversation. It should be used in conjunction with either the WAIT or CONFIRM options, the SYNCPOINT command, or the WAIT CONVID command (depending on the conversation sync level).

Table 10. Command sequences for ending a conversation

Sync level	Command sequence
0	SEND LAST WAIT FREE
1	SEND LAST CONFIRM FREE

Table 10. Command sequences for ending a conversation (continued)

Sync level	Command sequence
2	SEND LAST SYNCPOINT FREE

It is important that the SEND LAST command for sync level 2 is **not** accompanied by WAIT or CONFIRM because either of these options will cause the conversation to end before the subsequent syncpoint has propagated to the partner transaction. This may mean that protected resources of one transaction could be committed while those in the partner transaction could be backed out. The resulting state errors may also lead to the session being unbound.

From the state tables it can be seen that it is possible to end a conversation by issuing the FREE command, provided the conversation is in **send state** (state 2). This will generate an implicit SEND LAST WAIT command before the FREE is executed and is therefore not recommended for conversations using sync levels 1 and 2.

Note: A distributed transaction should not end a conversation by issuing an EXEC CICS RETURN command, but instead follow the sequence of commands shown in Table 10 on page 30. The issue of an EXEC CICS RETURN could lead to one or both transactions ending abnormally.

Emergency termination of a conversation

The ISSUE ABEND command provides a means of abnormally ending the conversation. It is valid for all levels of synchronization, but should be avoided at sync level 2, because its use at the wrong time can lead to a loss of data integrity.

ISSUE ABEND can be issued by either transaction, irrespective of whether it is in send or receive state, at any time after the conversation has started. For a conversation in **send state** (state 2), any deferred data that is waiting for transmission is flushed before the ISSUE ABEND command is transmitted.

The transaction that issues the ISSUE ABEND command is not itself abended. It must, however, issue a FREE command for the conversation unless it is designed to terminate immediately.

If an ISSUE ABEND command is issued in **receive state** (state 5), CICS purges all incoming data until an INVITE, syncpoint request, or LAST indicator is received. If LAST is received, no abend indication is sent to the partner transaction.

If an ISSUE ABEND is received, CICS abends the transaction with abend code AZCH, sets on EIBERR(= X'FF'),EIBFREE(= X'FF'), and places X'0864' in the first two bytes of EIBERRCD.

Unexpected termination of a conversation

If a partner system fails, or a session goes out of service in the middle of a DTP conversation, the conversation is terminated abnormally and the TERMERR condition is raised on the next command that accesses the conversation.

In addition, EIBERR and EIBFREE are set on (X'FF') and EIBERRCD contains a value representing the reason for the error, as follows:

X'08640001'
: partner system with persistent session support has failed and restarted

X'1008600B'
: session has failed due to a protocol error

X'A0000100'
: temporary session failure

X'A0010100'
: RTIMOUT timeout value was exceeded.

Checking the outcome of a DTP command

Checking the response from a DTP command can be separated into three stages.

The stages are :

1. Testing for request failure
2. Testing for indicators received on the conversation
3. Testing the conversation state.

Testing for request failure is the same as for other EXEC CICS commands in that conditions are raised and can be handled using HANDLE CONDITION or RESP. EIBRCODE will also contain an error code. Note that when an ISSUE ABEND has been received, and it is to be handled, a HANDLE ABEND should be used rather than a HANDLE CONDITION.

If the request has not failed, it is then possible to test for indicators received on the conversation. These are returned to the application in the EIB. The following EIB fields are relevant to all DTP commands:

EIBERR
when set to X'FF' indicates an error has occurred on the conversation. The reason is in EIBERRCD. This could be as a result of an ISSUE ERROR, ISSUE ABEND, or SYNCPOINT ROLLBACK command issued by the partner transaction. EIBERR can be set as a result of any command that can be issued while the conversation is in **receive state** (state 5) or following any command that causes a transmission to the partner system. It is safest to test EIBERR in conjunction with EIBFREE and EIBSYNRB after every DTP command.

EIBERRCD
contains the error code associated with EIBERR. If EIBERR is not set, this field is not used.

EIBFREE
when set to X'FF' indicates that the partner transaction had ended the conversation. It should be tested along with EIBERR and EIBSYNC to find out exactly how to end the conversation.

EIBSIG
when set to X'FF' indicates the partner transaction or system has issued an ISSUE SIGNAL command.

EIBSYNRB
when set to X'FF' indicates the partner transaction or system has issued a SYNCPOINT ROLLBACK command. (This is relevant only for conversations at sync level 2.)

Table 11 on page 32 shows how these EIB fields interact.

Table 11. Interaction between some EIB fields—all DTP commands

EIB- ERR	EIB- FREE	EIB- SYNRB	EIBERRCD	Description
X'FF'	X'00'	X'00'	X'08890000' X'08890001'	The partner transaction has sent ISSUE ERROR
X'FF'	X'00'	X'00'	X'08890100' X'08890101'	The partner system has sent ISSUE ERROR
X'FF'	X'FF'	X'00'	X'08640000'	The partner transaction has sent ISSUE ABEND
X'FF'	X'FF'	X'00'	X'08640001'	The partner system has sent ISSUE ABEND
X'FF'	X'FF'	X'00'	X'08640002'	A partner resource has timed out
X'FF'	X'FF'	X'00'	X'1008600B'	The session has failed due to a protocol error
X'FF'	X'FF'	X'00'	X'A0000100'	A temporary session failure

Table 11. Interaction between some EIB fields—all DTP commands (continued)

EIB- ERR	EIB- FREE	EIB- SYNRB	EIBERRCD	Description
X'FF'	X'FF'	X'00'	X'A0010100'	RTIMOUT has been triggered. (The task has timed out while waiting for terminal input.)
X'FF'	X'FF'	X'00'	X'10086032'	The PIP data sent with the CONNECT PROCESS was incorrectly specified
X'FF'	X'FF'	X'00'	X'10086034'	The partner system does not support mapped conversations
X'FF'	X'FF'	X'00'	X'080F6051'	The partner transaction failed security check
X'FF'	X'FF'	X'00'	X'10086041'	The partner transaction does not support the sync level requested on the CONNECT PROCESS
X'FF'	X'FF'	X'00'	X'10086021'	The partner transactions name is not recognized by the partner system
X'FF'	X'FF'	X'00'	X'084C0000'	The partner system cannot start the partner transaction
X'FF'	X'FF'	X'00'	X'084B6031'	The partner system is temporarily unable to start the partner transaction
X'FF'	X'00'	X'FF'	X'08240000'	The partner transaction or system has issued SYNCPOINT ROLLBACK
X'00'	X'00'	—	—	The command completed successfully.

In addition, the following EIB fields are relevant only to the RECEIVE and CONVERSE commands:

EIBCOMPL
when set to X'FF' indicates that all the data sent at one time has been received. This field is used in conjunction with the RECEIVE NOTRUNCATE command.

EIBCONF
when set to X'FF' indicates that the partner transaction has issued a SEND CONFIRM command and requires a response.

EIBEOC
when set to X'FF' indicates that an end-of-chain indicator has been received. This field is normally associated with a successful RECEIVE command.

EIBNODAT
when set to X'FF' indicates that no application data has been received.

EIBRECV
is only used when EIBERR is not set. When EIBRECV is on (X'FF'), another RECEIVE is required.

EIBSYNC
when set to X'FF' indicates that the partner transaction or system has requested a syncpoint. (This is relevant only for conversations at sync level 2.)

Table 12 on page 34 shows how some of these EIB fields interact for RECEIVE and CONVERSE commands.

Table 12. Interaction between some EIB fields—RECEIVE and CONVERSE commands only

EIB- ERR	EIB- FREE	EIB- RECV	EIB- SYNC	EIB- CONF	Description
X'00'	X'00'	X'00'	X'00'	X'00'	The partner transaction or system has issued SEND INVITE WAIT. The local program is now in send state.
X'00'	X'00'	X'00'	X'FF'	X'00'	The partner transaction or system has issued SEND INVITE, followed by a SYNCPOINT. The local program is now in syncsend state.
X'00'	X'00'	X'00'	X'00'	X'FF'	The partner transaction or system has issued SEND INVITE CONFIRM. The local program is now in confsend state.
X'00'	X'00'	X'FF'	X'00'	X'00'	The partner transaction or system has issued SEND or SEND WAIT. The local program is in receive state.
X'00'	X'00'	X'FF'	X'FF'	X'00'	The partner transaction or system has issued a SYNCPOINT. The local program is in syncreceive state.
X'00'	X'00'	X'FF'	X'00'	X'FF'	The partner transaction or system has issued a SEND CONFIRM. The local program is in confreceive state.
X'00'	X'FF'	X'00'	X'00'	X'00'	The partner transaction or system has issued a SEND LAST WAIT. The local program is in free state.
X'00'	X'FF'	X'00'	X'FF'	X'00'	The partner transaction or system has issued a SEND LAST followed by a SYNCPOINT. The local program is in syncfree state.
X'00'	X'FF'	X'00'	X'00'	X'FF'	The partner transaction or system has issued a SEND LAST CONFIRM. The local program is in conffree state.

After analyzing the EIB fields, you can test the conversation state to determine which DTP commands you can issue next. See “State transitions in APPC mapped conversations” on page 36.

Checking EIB fields and the conversation state

Most of the information supplied by EIB indicator fields can also be obtained from the conversation state.

Although the conversation state is easier to test, you cannot ignore EIBERR (and EIBERRCD).

For example, if after a SEND INVITE WAIT or a RECEIVE command has been issued, the conversation is in **receive state** (state 5), only EIBERR indicates that the partner transaction has sent an ISSUE ERROR. This is illustrated in Figure 8 on page 26 and Figure 9 on page 27.

It should be noted that the state tables provided contain not only states and commands issued, but also relevant EIB field settings. The order in which these EIB fields are shown provides a sensible sequence of checks for an application.

Summary of CICS commands for APPC mapped conversations

The CICS application programming interface provides a set of commands for use in APPC mapped conversations.

Table 13. Summary of CICS commands used in mapped conversations

Use to ...	Sync levels	CICS command	More information
Acquire a session.	0,1,2	ALLOCATE	"Allocating a session to the conversation" on page 21
Initiate a conversation.	0,1,2	CONNECT PROCESS	"Connecting the partner transaction" on page 22
Access session-related information.	0,1,2	EXTRACT PROCESS	"Back-end transaction initiation" on page 23
Send data and control information to the conversation partner.	0,1,2	SEND	"Sending data to the partner transaction" on page 24
Receive data from the conversation partner.	0,1,2	RECEIVE	"Receiving data from the partner transaction" on page 26
Send and receive data on the conversation.	0,1,2	CONVERSE	"The CONVERSE command" on page 27
Transmit any deferred data or control indicators.	0,1,2	WAIT CONVID	"Sending data to the partner transaction" on page 24
Reply positively to SEND CONFIRM.	1,2	ISSUE CONFIRMATION	"Receiving and replying to a confirmation request" on page 29
Prepare a conversation partner for syncpointing.	2	ISSUE PREPARE	"The ISSUE PREPARE command" on page 97
Inform the conversation partner of a program-detected error.	0,1,2	ISSUE ERROR	"Demanding INVITE from the partner transaction" on page 28
Signal an unusual condition to the conversation partner, usually against the flow of data.	0,1,2	ISSUE SIGNAL	"Requesting INVITE from the partner transaction" on page 28
Inform the conversation partner that the conversation should be abandoned.	0,1,2	ISSUE ABEND	"Emergency termination of a conversation" on page 31
Free the session.	0,1,2	FREE	"Ending the conversation" on page 30
Inform all conversation partners of readiness to commit changes to recoverable resources.	2	SYNCPOINT	"Syncpointing a distributed process" on page 97
Inform conversation partners of the need to back out changes to recoverable resources.	2	SYNCPOINT ROLLBACK	"The SYNCPOINT ROLLBACK command" on page 98

For programming information about CICS commands, see CICS command summary.

State transitions in APPC mapped conversations

These topics shows the state transitions that occur when transactions engage in APPC mapped conversations under the EXEC CICS API.

The state transitions are presented in the form of state tables; and there is one table for each of the three allowable sync levels.

The state tables provide the following information for writing a DTP program. Firstly, they show which commands can be issued from each conversation state. Secondly, they show the state transitions that can occur and the EIB fields that can be set as a result of issuing a command.

How to use the state tables

The state tables show the commands you can issue, the EIB flags that can be set when the command is issued, and the conversation states.

The commands you can issue, coupled with the EIB flags that can be set after execution, are shown in the first column of each table. Alongside each command, the EIB fields shown are in the order in which the application should test them. The possible conversation states are shown across the top of the table. The states correspond to the columns of the table. The intersection of row (command and EIB flag) and column (state) represents the state transition, if any, that occurs when that command returning a particular EIB flag is issued in that state.

A number at an intersection indicates the state number of the next state. Other symbols represent other conditions, as follows:

Symbol	Meaning
N/A	Cannot occur.
×	The EIB flag is any one that has not been covered in earlier rows, or it is irrelevant (but see the note on EIBSIG if you want to use ISSUE SIGNAL).
Abend *code*	The command is not valid in this state. Issuing a command in a state in which it is not valid usually causes an ATCV abend. When a different abend code applies, this is shown in the tables.
INVREQ	The command is not valid in this state. An INVREQ condition is returned.
=	Remains in current state.
End	End of conversation.

Initial conversation states

Before a session is allocated, there is no conversation, and therefore no conversation state.

The **`EXEC CICS ALLOCATE`** command gets a session to start a new conversation and does not affect any conversation that is already in progress, hence the **ALLOCATE** command does not appear in the tables. After the **ALLOCATE** command is successfully issued, the new conversation in the front-end transaction is in **ALLOCATED** state.

The back-end transaction starts in **RECEIVE** state after the front-end transaction has successfully issued the **`CONNECT PROCESS`** command.

Testing the conversation state

There are two ways for a transaction to inquire on the current state of one of its conversations.

The first is to use the **`EXEC CICS EXTRACT ATTRIBUTES STATE`** command and the second is to use the STATE parameter on the DTP commands. In both cases the current state is returned to the application in a CICS value data area (cvda). Table 14 on page 37 shows how the cvda codes relate to the conversation state. The table also shows the symbolic names defined for these cvda values.

Table 14. The conversation states

States used in this book		States used in DTP programs	
State name	**State number**	**Symbolic name**	**cvda code**
Allocated	1	DFHVALUE(ALLOCATED)	81
Send	2	DFHVALUE(SEND)	90
Pendreceive	3	DFHVALUE(PENDRECEIVE)	87
Pendfree	4	DFHVALUE(PENDFREE)	86
Receive	5	DFHVALUE(RECEIVE)	88
Confreceive	6	DFHVALUE(CONFRECEIVE)	83
Confsend	7	DFHVALUE(CONFSEND)	84
Conffree	8	DFHVALUE(CONFFREE)	82
Syncreceive	9	DFHVALUE(SYNCRECEIVE)	92
Syncsend	10	DFHVALUE(SYNCSEND)	93
Syncfree	11	DFHVALUE(SYNCFREE)	91
Free	12	DFHVALUE(FREE)	85
Rollback	13	DFHVALUE(ROLLBACK)	89

State tables for APPC mapped conversations at sync level 0

Tables showing the state transitions that occur when transactions engage in APPC mapped conversations at sync level 0, under the EXEC CICS API.

The ISSUE SIGNAL command and the EIBSIG flag

In the tables, the EIBSIG flag is not mentioned. This is because its use is optional and is entirely a matter of agreement between the two conversation partners. In the worst case, it can occur at any time after every command that affects the EIB flags. However, used for the purpose for which it was intended, it usually occurs after a SEND command. Its priority in the order of testing depends on the role you give it in the application.

The EIBSIG flag is set when the partner issues the **`ISSUE SIGNAL`** command.

The `RECEIVE NOTRUNCATE` command

The **`RECEIVE NOTRUNCATE`** command returns a zero value in EIBCOMPL to indicate that the user buffer was too small to contain all the data received from the partner transaction. Normally, you would continue to issue **`RECEIVE NOTRUNCATE`** commands until the last section of data is passed to you, which is indicated by EIBCOMPL = X'FF' . If NOTRUNCATE is not specified, and the data area specified by the RECEIVE command is too small to contain all the data received, CICS truncates the data and sets the LENGERR condition.

State tables

Table 15. States 1 - 6

		Command returns	**ALLOC-ATED**	**SEND**	**PEND-RECEIVE**	**PEND-FREE**	**RECEIVE**	**CONF-RECEIVE**
Command issued	**EIB flags returned**		**State 1**	**State 2**	**State 3**	**State 4**	**State 5**	**State 6**
CONNECT PROCESS	EIBERR + EIBFREE	Immediately	12	Abend	Abend	Abend	Abend	N/A

Table 15. States 1 - 6 (continued)

Command issued	EIB flags returned	Command returns	ALLOC- ATED	SEND	PEND- RECEIVE	PEND- FREE	RECEIVE	CONF- RECEIVE
			State 1	State 2	State 3	State 4	State 5	State 6
CONNECT PROCESS	×	Immediately	2	Abend	Abend	Abend	Abend	N/A
EXTRACT PROCESS (back-end transaction only)	×	Immediately	=	=	=	=	=	N/A
EXTRACT ATTRIBUTES	×	Immediately	=	=	=	=	=	N/A
SEND (any valid form)	EIBERR + EIBFREE	After error detected	Abend	12	Abend	Abend	Abend	N/A
SEND (any valid form)	EIBERR	After error detected	Abend	5	Abend	Abend	Abend	N/A
SEND INVITE WAIT	×	After data flows	Abend	5	Abend	Abend	Abend	N/A
SEND INVITE	×	After data buffered	Abend	3	Abend	Abend	Abend	N/A
SEND LAST WAIT	×	After data flows	Abend	12	Abend	Abend	Abend	N/A
SEND LAST	×	After data buffered	Abend	4	Abend	Abend	Abend	N/A
SEND WAIT	×	After data flows	Abend	=	Abend	Abend	Abend	N/A
SEND	×	After data buffered	Abend	=	Abend	Abend	Abend	N/A
RECEIVE	EIBERR + EIBFREE	After error detected	Abend	12	12	Abend	12	N/A
RECEIVE	EIBERR	After error detected	Abend	5	5	Abend	=	N/A
RECEIVE	EIBFREE	After error detected	Abend	12	12	Abend	12	N/A
RECEIVE	EIBRECV	When data available	Abend	5	5	Abend	=	N/A
RECEIVE NOTRUNCATE	EIBCOMPL	When data available	Abend	5	5	Abend	=	N/A
RECEIVE	×	When data available	Abend	=	2	Abend	2	N/A
CONVERSE (equivalent to SEND INVITE WAIT followed by RECEIVE)	As for RECEIVE		As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE
ISSUE ERROR	EIBFREE	After response from partner	Abend	12	12	Abend	12	N/A
ISSUE ERROR	×	After response from partner	Abend	=	2	Abend	2	N/A
ISSUE ABEND	×	Immediately	Abend	12	12	12	12	N/A
ISSUE SIGNAL	×	Immediately	Abend	=	=	Abend	=	N/A
WAIT CONVID	×	Immediately	Abend	=	5	12	Abend	N/A
FREE	×	Immediately	End	End	Abend	End	Abend	N/A

Table 16. States 7 - 13

Command issued	EIB flags returned	CONF- SEND	CONF- FREE	SYNC- RECEIVE	SYNC- SEND	SYNC- FREE	FREE	ROLL- BACK
		State 7	State 8	State 9	State 10	State 11	State 12	State 13
CONNECT PROCESS	EIBERR + EIBFREE	N/A	N/A	N/A	N/A	N/A	Abend	N/A
CONNECT PROCESS	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
EXTRACT PROCESS (back-end transaction only)	×	N/A	N/A	N/A	N/A	N/A	=	N/A
EXTRACT ATTRIBUTES	×	N/A	N/A	N/A	N/A	N/A	=	N/A

Table 16. States 7 - 13 (continued)

Command issued	EIB flags returned	CONF- SEND	CONF- FREE	SYNC- RECEIVE	SYNC- SEND	SYNC- FREE	FREE	ROLL- BACK
		State 7	State 8	State 9	State 10	State 11	State 12	State 13
SEND (any valid form)	EIBERR + EIBFREE	N/A	N/A	N/A	N/A	N/A	Abend	N/A
SEND (any valid form)	EIBERR	N/A	N/A	N/A	N/A	N/A	Abend	N/A
SEND INVITE WAIT	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
SEND INVITE	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
SEND LAST WAIT	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
SEND LAST	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
SEND WAIT	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
SEND	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
RECEIVE	EIBERR + EIBFREE	N/A	N/A	N/A	N/A	N/A	Abend	N/A
RECEIVE	EIBERR	N/A	N/A	N/A	N/A	N/A	Abend	N/A
RECEIVE	EIBFREE	N/A	N/A	N/A	N/A	N/A	Abend	N/A
RECEIVE	EIBRECV	N/A	N/A	N/A	N/A	N/A	Abend	N/A
RECEIVE NOTRUNCATE	EIBCOMPL	N/A	N/A	N/A	N/A	N/A	Abend	N/A
RECEIVE	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
CONVERSE (equivalent to SEND INVITE WAIT followed by RECEIVE)	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE
ISSUE ERROR	EIBFREE	N/A	N/A	N/A	N/A	N/A	Abend	N/A
ISSUE ERROR	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
ISSUE ABEND	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
ISSUE SIGNAL	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
WAIT CONVID	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
FREE	×	N/A	N/A	N/A	N/A	N/A	End	N/A

State tables for APPC mapped conversations at sync level 1

Tables showing the state transitions that occur when transactions engage in APPC mapped conversations at sync level 1, under the EXEC CICS API.

The `ISSUE SIGNAL` command and the EIBSIG flag

In the tables, the EIBSIG flag is not mentioned. This is because its use is optional and is entirely a matter of agreement between the two conversation partners. In the worst case, it can occur at any time after every command that affects the EIB flags. However, used for the purpose for which it was intended, it usually occurs after a SEND command. Its priority in the order of testing depends on the role you give it in the application.

The EIBSIG flag is set when the partner issues the **`ISSUE SIGNAL`** command.

The `RECEIVE NOTRUNCATE` command

The **`RECEIVE NOTRUNCATE`** command returns a zero value in EIBCOMPL to indicate that the user buffer was too small to contain all the data received from the partner transaction. Normally, you would continue to issue **`RECEIVE NOTRUNCATE`** commands until the last section of data is passed to you, which is indicated by EIBCOMPL = X'FF' . If NOTRUNCATE is not specified, and the data area specified by the RECEIVE command is too small to contain all the data received, CICS truncates the data and sets the LENGERR condition.

State tables

Table 17. States 1 - 6

		Command returns	ALLOC-ATED	SEND	PEND-RECEIVE	PEND-FREE	RECEIVE	CONF-RECEIVE
Command issued	**EIB flags returned**		**State 1**	**State 2**	**State 3**	**State 4**	**State 5**	**State 6**
CONNECT PROCESS	EIBERR + EIBFREE	Immediately	12	Abend	Abend	Abend	Abend	Abend
CONNECT PROCESS	×	Immediately	2	Abend	Abend	Abend	Abend	Abend
EXTRACT PROCESS (back-end transaction only)	×	Immediately	Abend	=	=	=	=	=
EXTRACT ATTRIBUTES	×	Immediately	=	=	=	=	=	=
SEND (any valid form)	EIBERR + EIBFREE	After error flow detected	Abend	12	12	12	Abend	Abend
SEND (any valid form)	EIBERR	After error flow detected	Abend	5	5	5	Abend	Abend
SEND INVITE WAIT	×	After data flows	Abend	5	Abend	Abend	Abend	Abend
SEND INVITE CONFIRM	×	After response from partner	Abend	5	Abend	Abend	Abend	Abend
SEND INVITE	×	After data buffered	Abend	3	Abend	Abend	Abend	Abend
SEND LAST WAIT	×	After data flows	Abend	12	Abend	Abend	Abend	Abend
SEND LAST CONFIRM	×	After response from partner	Abend	12	Abend	Abend	Abend	Abend
SEND LAST	×	After data buffered	Abend	4	Abend	Abend	Abend	Abend
SEND WAIT	×	After data flows	Abend	=	Abend	Abend	Abend	Abend
SEND CONFIRM	×	After response from partner	Abend	=	5	12	Abend	Abend
SEND	×	After data buffered	Abend	=	Abend	Abend	Abend	Abend
RECEIVE	EIBERR + EIBFREE	After error detected	Abend	12	12	Abend	12	Abend
RECEIVE	EIBERR	After error detected	Abend	5	5	Abend	=	Abend
RECEIVE	EIBCONF + EIBFREE	After confirm flow detected	Abend	8	8	Abend	8	Abend
RECEIVE	EIBCONF + EIBRECV	After confirm flow detected	Abend	6	6	Abend	6	Abend
RECEIVE	EIBCONF	After confirm flow detected	Abend	7	7	Abend	7	Abend
RECEIVE	EIBFREE	After error detected	Abend	12	12	Abend	12	Abend
RECEIVE	EIBRECV	When data available	Abend	5	5	Abend	=	Abend
RECEIVE NOTRUNCATE	EIBCOMPL	When data available	Abend	5	5	Abend	=	Abend
RECEIVE	×	When data available	Abend	=	2	Abend	2	Abend
CONVERSE (equivalent to SEND INVITE WAIT followed by RECEIVE)	As for RECEIVE		As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE
ISSUE CONFIRMATION	×	Immediately	Abend	Abend	Abend	Abend	Abend	5
ISSUE ERROR	EIBFREE	After response from partner	Abend	12	12	Abend	12	12
ISSUE ERROR	×	After response from partner	Abend	=	2	Abend	2	2
ISSUE ABEND	×	Immediately	Abend	12	12	12	12	12

Table 17. States 1 - 6 (continued)

Command issued	EIB flags returned	Command returns	ALLOC-ATED	SEND	PEND-RECEIVE	PEND-FREE	RECEIVE	CONF-RECEIVE
			State 1	State 2	State 3	State 4	State 5	State 6
ISSUE SIGNAL	×	Immediately	Abend	=	=	Abend	=	=
WAIT CONVID	×	Immediately	Abend	=	5	12	Abend	Abend
FREE	×	Immediately	End	End	Abend	End	Abend	Abend

Table 18. States 7 - 13

Command issued	EIB flags returned	CONF- SEND	CONF-FREE	SYNC-RECEIVE	SYNC-SEND	SYNC-FREE	FREE	ROLL-BACK
		State 7	State 8	State 9	State 10	State 11	State 12	State 13
CONNECT PROCESS	EIBERR + EIBFREE	Abend	Abend	N/A	N/A	N/A	Abend	N/A
CONNECT PROCESS	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
EXTRACT PROCESS (back-end transaction only)	×	=	=	N/A	N/A	N/A	=	N/A
EXTRACT ATTRIBUTES	×	=	=	N/A	N/A	N/A	=	N/A
SEND (any valid form)	EIBERR + EIBFREE	Abend	Abend	N/A	N/A	N/A	Abend	N/A
SEND (any valid form)	EIBERR	Abend	Abend	N/A	N/A	N/A	Abend	N/A
SEND INVITE WAIT	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
SEND INVITE CONFIRM	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
SEND INVITE	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
SEND LAST WAIT	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
SEND LAST CONFIRM	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
SEND LAST	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
SEND WAIT	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
SEND CONFIRM	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
SEND	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
RECEIVE	EIBERR + EIBFREE	Abend	Abend	N/A	N/A	N/A	Abend	N/A
RECEIVE	EIBERR	Abend	Abend	N/A	N/A	N/A	Abend	N/A
RECEIVE	EIBCONF + EIBFREE	Abend	Abend	N/A	N/A	N/A	Abend	N/A
RECEIVE	EIBCONF + EIBRECV	Abend	Abend	N/A	N/A	N/A	Abend	N/A
RECEIVE	EIBCONF	Abend	Abend	N/A	N/A	N/A	Abend	N/A
RECEIVE	EIBFREE	Abend	Abend	N/A	N/A	N/A	Abend	N/A
RECEIVE	EIBRECV	Abend	Abend	N/A	N/A	N/A	Abend	N/A
RECEIVE NOTRUNCATE	EIBCOMPL	Abend	Abend	N/A	N/A	N/A	Abend	N/A
RECEIVE	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
CONVERSE (equivalent to SEND INVITE WAIT followed by RECEIVE)		As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE
ISSUE CONFIRMATION	×	2	12	N/A	N/A	N/A	Abend	N/A
ISSUE ERROR	EIBFREE	12	12	N/A	N/A	N/A	Abend	N/A
ISSUE ERROR	×	2	2	N/A	N/A	N/A	Abend	N/A
ISSUE ABEND	×	12	12	N/A	N/A	N/A	Abend	N/A
ISSUE SIGNAL	×	=	=	N/A	N/A	N/A	Abend	N/A
WAIT CONVID	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
FREE	×	Abend	Abend	N/A	N/A	N/A	End	N/A

State tables for APPC mapped conversations at sync level 2

Tables showing the state transitions that occur when transactions engage in APPC mapped conversations at sync level 2, under the EXEC CICS API.

The ISSUE SIGNAL command and the EIBSIG flag

In the tables, the EIBSIG flag is not mentioned. This is because its use is optional and is entirely a matter of agreement between the two conversation partners. In the worst case, it can occur at any time after every command that affects the EIB flags. However, used for the purpose for which it was intended, it usually occurs after a SEND command. Its priority in the order of testing depends on the role you give it in the application.

The EIBSIG flag is set when the partner issues the **ISSUE SIGNAL** command.

The RECEIVE NOTRUNCATE command

The **RECEIVE NOTRUNCATE** command returns a zero value in EIBCOMPL to indicate that the user buffer was too small to contain all the data received from the partner transaction. Normally, you would continue to issue **RECEIVE NOTRUNCATE** commands until the last section of data is passed to you, which is indicated by EIBCOMPL = X'FF' . If NOTRUNCATE is not specified, and the data area specified by the RECEIVE command is too small to contain all the data received, CICS truncates the data and sets the LENGERR condition.

Restrictions when using APPC transaction routing

Where APPC transaction routing is in use, the ISSUE SIGNAL command is invalid in the following states:

SYNC-RECEIVE
SYNC-SEND
SYNC-FREE

State changes for the SYNCPOINT and SYNCPOINT ROLLBACK commands

When the SYNCPOINT and SYNCPOINT ROLLBACK commands are issued, they are propagated on, and affect the state of, all the conversations that are currently active for the task, including MRO conversations.

Following rollback, the conversation can be in **SEND** or **RECEIVE** state, depending on the conversation state at the start of the current distributed unit of work. The conversation can be in **FREE** state if it ended abnormally due to session failure or due to deallocate abend being received, or if the partner transaction issued a SEND LAST WAIT or FREE command.

After a syncpoint or rollback, it is advisable to determine the conversation state before issuing any further commands against the conversation.

State changes following the ISSUE PREPARE command

Although ISSUE PREPARE can return with the conversation in either **SYNCSEND** state, **SYNCRECEIVE** state, or **SYNCFREE** state, the only commands allowed on that conversation following an ISSUE PREPARE are SYNCPOINT and SYNCPOINT ROLLBACK. All other commands Abend.

State tables

Table 19. States 1 - 6

		Command returns	ALLO-CATED	SEND	PEND-RECEIVE	PEND-FREE	RECEIVE	CONF-RECEIVE
Command issued	**EIB flag returned**		**State 1**	**State 2**	**State 3**	**State 4**	**State 5**	**State 6**
CONNECT PROCESS	EIBERR + EIBFREE	Immediately	12	Abend	Abend	Abend	Abend	Abend
CONNECT PROCESS	×	Immediately	2	Abend	Abend	Abend	Abend	Abend

Table 19. States 1 - 6 (continued)								
		Command returns	**ALLO-CATED**	**SEND**	**PEND-RECEIVE**	**PEND-FREE**	**RECEIVE**	**CONF-RECEIVE**
Command issued	**EIB flag returned**		**State 1**	**State 2**	**State 3**	**State 4**	**State 5**	**State 6**
EXTRACT PROCESS (back-end transaction only)	×	Immediately	=	=	=	=	=	=
EXTRACT ATTRIBUTES	×	Immediately	=	=	=	=	=	=
SEND (any valid form)	EIBERR + EIBSYNRB	After error flow detected	Abend	13	13	13	Abend	Abend
SEND (any valid form)	EIBERR + EIBFREE	After error flow detected	Abend	12	12	12	Abend	Abend
SEND (any valid form)	EIBERR	After error flow detected	Abend	5	5	5	Abend	Abend
SEND INVITE WAIT	×	After data flows	Abend	5	Abend	Abend	Abend	Abend
SEND INVITE CONFIRM	×	After response from partner	Abend	5	Abend	Abend	Abend	Abend
SEND INVITE	×	After data buffered	Abend	3	Abend	Abend	Abend	Abend
SEND LAST WAIT	×	After data flows	Abend	12	Abend	Abend	Abend	Abend
SEND LAST CONFIRM	×	After response from partner	Abend	12	Abend	Abend	Abend	Abend
SEND LAST	×	After data buffered	Abend	4	Abend	Abend	Abend	Abend
SEND WAIT	×	After data flows	Abend	=	Abend	Abend	Abend	Abend
SEND CONFIRM	×	After response from partner	Abend	=	5	12	Abend	Abend
SEND	×	After data buffered	Abend	=	Abend	Abend	Abend	Abend
RECEIVE	EIBERR + EIBSYNRB	After rollback flow detected	Abend	13	13	Abend	13	Abend
RECEIVE	EIBERR + EIBFREE	After error detected	Abend	12	12	Abend	12	Abend
RECEIVE	EIBERR	After error detected	Abend	5	5	Abend	=	Abend
RECEIVE	EIBSYNC + EIBFREE	After sync flow detected	Abend	11	11	Abend	11	Abend
RECEIVE	EIBSYNC + EIBRECV	After sync flow detected	Abend	9	9	Abend	9	Abend
RECEIVE	EIBSYNC	After sync flow detected	Abend	10	10	Abend	10	Abend
RECEIVE	EIBCONF + EIBFREE	After confirm flow detected	Abend	8	8	Abend	8	Abend
RECEIVE	EIBCONF + EIBRECV	After confirm flow detected	Abend	6	6	Abend	6	Abend
RECEIVE	EIBCONF	After confirm flow detected	Abend	7	7	Abend	7	Abend
RECEIVE	EIBFREE	After error flow detected	Abend	12	12	Abend	12	Abend
RECEIVE	EIBRECV	When data available	Abend	5	5	Abend	=	Abend
RECEIVE NOTRUNCATE	EIBCOMPL	When data available	Abend	5	5	Abend	=	Abend
RECEIVE	×	When data available	Abend	=	2	Abend	2	Abend
CONVERSE (equivalent to SEND INVITE WAIT followed by RECEIVE)	As for RECEIVE		As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE

Table 19. States 1 - 6 (continued)

Command issued	EIB flag returned	Command returns	ALLO-CATED	SEND	PEND-RECEIVE	PEND-FREE	RECEIVE	CONF-RECEIVE
			State 1	State 2	State 3	State 4	State 5	State 6
ISSUE CONFIRMATION	×	Immediately	Abend	Abend	Abend	Abend	Abend	5
ISSUE ERROR	EIBFREE	After response from partner	Abend	12	12	Abend	12	12
ISSUE ERROR	×	After response from partner	Abend	=	2	Abend	2	2
ISSUE ABEND	×	Immediately	Abend	12	12	12	12	12
ISSUE SIGNAL	×	Immediately	Abend	=	=	Abend	=	=
ISSUE PREPARE	EIBERR + EIBSYNRB	After response from partner	INVREQ	13	13	13	INVREQ	INVREQ
ISSUE PREPARE	EIBERR + EIBFREE	After error detected	INVREQ	12	12	12	INVREQ	INVREQ
ISSUE PREPARE	EIBERR	After error detected	INVREQ	5	5	5	INVREQ	INVREQ
ISSUE PREPARE	×	After response from partner	INVREQ	10	9	11	INVREQ	INVREQ
SYNCPOINT	EIBRLDBK	After response from partner	=	2 or 5	2 or 5	2 or 5	Abend ASP2	Abend ASP2
SYNCPOINT	×	After response from partner	=	=	5	12	Abend ASP2	Abend ASP2
SYNCPOINT ROLLBACK	×	After rollback across UOW	=	2 or 5	2 or 5	2 or 5	2 or 5	2 or 5
WAIT CONVID	×	Immediately	Abend	=	5	12	Abend	Abend
FREE	×	Immediately	End	End	Abend	End	Abend	Abend

Table 20. States 7 - 13

Command issued	EIB flag returned	CONF-SEND	CONF-FREE	SYNC-RECEIVE	SYNC-SEND	SYNC-FREE	FREE	ROLL-BACK
		State 7	State 8	State 9	State 10	State 11	State 12	State 13
CONNECT PROCESS	EIBERR + EIBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend
CONNECT PROCESS	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
EXTRACT PROCESS (back-end transaction only)	×	=	=	=	=	=	=	=
EXTRACT ATTRIBUTES	×	=	=	=	=	=	=	=
SEND (any valid form)	EIBERR + EIBSYNRB	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND (any valid form)	EIBERR + EIBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND (any valid form)	EIBERR	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND INVITE WAIT	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND INVITE CONFIRM	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND INVITE	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND LAST WAIT	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND LAST CONFIRM	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND LAST	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND WAIT	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND CONFIRM	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBERR + EIBSYNRB	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBERR + EIBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend

Table 20. States 7 - 13 (continued)

Command issued	**EIB flag returned**	**CONF-SEND**	**CONF-FREE**	**SYNC-RECEIVE**	**SYNC-SEND**	**SYNC-FREE**	**FREE**	**ROLL-BACK**
		State 7	**State 8**	**State 9**	**State 10**	**State 11**	**State 12**	**State 13**
RECEIVE	EIBERR	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBSYNC + EIBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBSYNC + EIBRECV	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBSYNC	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBCONF + EIBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBCONF + EIBRECV	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBCONF	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBRECV	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE NOTRUNCATE	EIBCOMPL	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
CONVERSE (equivalent to SEND INVITE WAIT followed by RECEIVE)		As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE
ISSUE CONFIRMATION	×	2	12	Abend	Abend	Abend	Abend	Abend
ISSUE ERROR	EIBFREE	12	12	12	12	12	Abend	Abend
ISSUE ERROR	×	2	2	2	2	2	Abend	Abend
ISSUE ABEND	×	12	12	12	12	12	Abend	Abend
ISSUE SIGNAL	×	=	=	=	=	=	Abend	Abend
ISSUE PREPARE	EIBERR + EIBSYNRB	Abend	Abend	Abend	Abend	Abend	Abend	Abend
ISSUE PREPARE	EIBERR + EIBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend
ISSUE PREPARE	EIBERR	Abend	Abend	Abend	Abend	Abend	Abend	Abend
ISSUE PREPARE	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SYNCPOINT	EIBRLDBK	Abend ASP2	Abend ASP2	2 or 5	2 or 5	2 or 5	=	Abend ASP2
SYNCPOINT	×	Abend ASP2	Abend ASP2	5	2	12	=	Abend ASP2
SYNCPOINT ROLLBACK	×	2 or 5	2 or 5	2 or 5	2 or 5	2 or 5	=	2 or 5
WAIT CONVID	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
FREE	×	Abend	Abend	Abend	Abend	Abend	End	Abend

Chapter 3. Writing programs for MRO conversations

These topics describe the CICS APIs available for DTP programming using MRO conversations.

MRO conversation flow

These topics introduce some of the MRO DTP commands.

Conversation initiation

The front-end transaction is responsible for acquiring a session, specifying the conversation characteristics and requesting the startup of the back-end transaction in the partner system.

Allocating a session to the conversation

Initially, there is no conversation, and therefore no conversation state. By issuing an ALLOCATE command, the front-end transaction acquires a session to start a new conversation.

The RESP value returned should be checked to ensure that a session has been allocated. If successfully allocated, DFHRESP(NORMAL), the conversation is in **allocated state** (state 1) and the session identifier (**convid**) from EIBRSRCE must be saved immediately.

The convid must be used in subsequent commands for this conversation. Figure 10 on page 48 shows a program fragment containing an example of the ALLOCATE command. You will notice that the PROFILE option has been omitted from the command.

If the PROFILE option is specified for an MRO link, CICS ignores it at execution time. So none of the facilities selected through use of a profile (for example, RTIMEOUT and JOURNALING) are available. The front-end transaction has no control over its session processing options when an MRO session is being used.

A back-end transaction with an MRO session as its principal facility will be sent the INBFMH parameter by CICS, regardless of the what the front-end transaction specifies on the PROFILE option of the ALLOCATE command.

Using ATI to allocate a session

Front-end transactions are often initiated from terminals. But it is also possible to use the EXEC CICS START command to initiate a front-end transaction on an MRO session.

When the front-end transaction is successfully started in this way, a conversation can continue as if an ALLOCATE command had been issued. The only difference is that an automatically-initiated front-end transaction has the MRO session as its *principal facility*.

Connecting the partner transaction

When a session has been acquired, the next step is to cause the partner transaction to be initiated.

The state table shows that, in **allocated state** (state 1), one of the commands available is SEND. Using this command, the back-end transaction's identifier can be specified in the first four bytes of the data which, when transferred to the partner system, will be used to attach the required back-end transaction. The send buffer containing the transaction identifier together with any other data, will be flushed immediately and the front-end transaction will wait until a response is received from the back end. Figure 10 on page 48 shows an example in which a transaction identifier is sent.

Alternatively, when a session has been acquired, the front-end transaction can build and send an attach header with the first transmission of data. The attach header can be built using the BUILD ATTACH command.

When using the BUILD ATTACH command, an eight-character name must be given to the built attach header which can then be used in the ATTACHID option of the first SEND (or CONVERSE) command. The back-end transaction identifier should also be specified.

```
 * ...
DATA DIVISION.
WORKING-STORAGE SECTION.
* ...
01 FILLER.
02 WS-CONVID PIC X(4).
02 WS-RESP PIC S9(8) COMP.
02 WS-STATE PIC S9(8) COMP.
02 WS-SYSID PIC X(4) VALUE 'SYSB'.
02 WS-PROC PIC X(4) VALUE 'BBBB'.
02 WS-LEN-PROCN PIC S9(5) COMP VALUE +4.
* ...
PROCEDURE DIVISION.
* ...
EXEC CICS ALLOCATE SYSID(WS-SYSID) RESP(WS-RESP) END-EXEC.
IF WS-RESP = DFHRESP(NORMAL)
THEN MOVE EIBRSRCE TO WS-CONVID
ELSE
* ... No session allocated. Examine EIBRCODE.
END-IF.
* ...
EXEC CICS SEND CONVID(WS-CONV)
RESP(WS-RESP) STATE(WS-STATE)
FROM(WS-PROC) LENGTH(WS-LEN-PROCN)
END-EXEC.
IF WS-RESP = DFHRESP(NORMAL)
THEN
* ... No errors, conversation started.
ELSE
* ... Conversation not started. Examine EIBRCODE.
END-IF.
```

Figure 10. Starting an MRO conversation

Back-end transaction initiation

The back-end transaction is initiated either by an attach header received from the partner system or by a transaction identifier included in the incoming data, and is started with the session as its principal facility.

Initially, the back-end transaction should determine the convid from EIBTRMID. This is not strictly necessary because the session is the back-end transaction's principal facility making the CONVID parameter optional for DTP commands on this conversation. However, the convid is very useful for audit trails. Also, if the back-end transaction is involved in more than one conversation, then always specifying the convid will improve program readability and problem determination. Figure 11 on page 49 shows a back-end transaction that does obtain the convid.

When the back-end transaction receives data, the presence of an attach header is indicated by either EIBATT or RESP(INBFMH). One of these is normally set after the back-end transaction issues its first RECEIVE command. The EXTRACT ATTACH command can be used to access session-related information from the attach header (for example, the back-end transaction identifier) if required, but it is not mandatory.

```
 * ...
DATA DIVISION.
WORKING-STORAGE SECTION.
* ...
01 FILLER.
02 WS-CONVID PIC X(4).
02 WS-STATE PIC S9(7) COMP.
* ...
01 FILLER.
02 WS-RECORD PIC X(100).
02 WS-MAX-LEN PIC S9(5) COMP VALUE +100.
02 WS-RCVD-LEN PIC S9(5) COMP VALUE +0.
* ...
PROCEDURE DIVISION.
* ...
EXEC CICS ASSIGN FACILITY(WS-CONVID) END-EXEC.
* ...
* Receive data from the front-end transaction.
*
EXEC CICS RECEIVE CONVID(WS-CONVID) STATE(WS-STATE)
INTO(WS-RECORD) MAXLENGTH(WS-MAX-LEN)
NOTRUNCATE LENGTH(WS-RCVD-LEN)
END-EXEC.
*
* ... Check outcome of RECEIVE.
* ...
```

Figure 11. Startup of a back-end MRO transaction

It is possible that the back-end transaction may fail to start. This will result in the front-end transaction abending. When this happens, message DFHIR3783 contains the reason for the error.

Transferring data on the conversation

These topics discuss how to pass data between the front-end and back-end transactions.

Sending data to the partner transaction

The SEND command is used to send data to the connected partner.

This command is valid in **allocated state** (state 1) or **send state** (state 2). Because a successful simple SEND completes in **send state** (state 2), it is possible to issue a number of successive sends.

An example of a simple SEND command can be seen in Figure 12 on page 50.

```
 * ...
DATA DIVISION.
WORKING-STORAGE SECTION.
* ...
01 FILLER.
02 WS-CONVID PIC X(4).
02 WS-RESP PIC S9(8) COMP.
02 WS-STATE PIC S9(8) COMP.
* ...
01 FILLER.
02 WS-SEND-AREA PIC X(70).
02 WS-SEND-LEN PIC S9(5) COMP VALUE +70.
* ...
01 FILLER.
02 WS-RCVD-AREA PIC X(100).
02 WS-MAX-LEN PIC S9(5) COMP VALUE +100.
02 WS-RCVD-LEN PIC S9(5) COMP VALUE +0.
* ...
PROCEDURE DIVISION.
* ...
EXEC CICS SEND CONVID(WS-CONVID) RESP(WS-RESP)
STATE(WS-STATE)
FROM(WS-SEND-AREA) LENGTH (WS-SEND-LEN)
END-EXEC.
* ... Check outcome of SEND.
* ...
*
EXEC CICS SEND INVITE CONVID(WS-CONVID)
RESP(WS-RESP) STATE(WS-STATE)
END-EXEC.
* ...
* Receive data from the partner transaction.
*
EXEC CICS RECEIVE CONVID(WS-CONVID)
RESP(WS-RESP) STATE(WS-STATE)
INTO(WS-RCVD-AREA) MAXLENGTH(WS-MAX-LEN)
NOTRUNCATE LENGTH(WS-RCVD-LEN)
END-EXEC.
*
* ... Check outcome of RECEIVE.
* ...
```

Figure 12. Transferring data on an MRO conversation

Switching from sending to receiving data

To switch from sending to receiving, use a **SEND INVITE** command with or without the WAIT option.

The state table in "State transitions in MRO conversations" on page 55 shows that after both SEND INVITE and SEND INVITE WAIT, the conversation switches the current state to **receive state** (state 5).

An example of a SEND INVITE command can be seen in Figure 12 on page 50.

```
 * ...
DATA DIVISION.
WORKING-STORAGE SECTION.
* ...
01 FILLER.
02 WS-RESP PIC S9(8) COMP.
02 WS-STATE PIC S9(8) COMP.
* ...
PROCEDURE DIVISION.
* ...
* Check return code from SEND INVITE
IF WS-RESP = DFHRESP(NORMAL)
THEN
* ... Request successful, check state
IF WS-STATE = DFHVALUE(RECEIVE)
THEN
* ... SEND OK, continue processing
ELSE
* ... Logic error, should never happen
END-IF
ELSE
* ... Examine EIBRCODE for source of error
END-IF.
* ...
```

Figure 13. Checking the outcome of a SEND INVITE command

Receiving data from the partner transaction

The RECEIVE command is used to receive data from the connected partner.

T he rows in the state tables for the RECEIVE command show the EIB fields that should be tested after issuing a RECEIVE command. As well as showing which field should be tested, the state table also shows the order in which the tests should be made. Instead of testing some of the EIB fields, you can test the resulting conversation state; this is shown in Figure 14 on page 52. Note that you should always test the value returned by the RESP option.

```
 * ...
DATA DIVISION.
WORKING-STORAGE SECTION.
* ...
01 FILLER.
02 WS-RESP PIC S9(8) COMP.
02 WS-STATE PIC S9(8) COMP.
* ...
PROCEDURE DIVISION.
* ...
* Check return code from RECEIVE
IF WS-RESP = DFHRESP(NORMAL)
THEN
* ... Request successful, check state
EVALUATE WS-STATE
WHEN DFHVALUE(ROLLBACK)
* ... Partner issued SYNCPOINT ROLLBACK
WHEN DFHVALUE(SYNCFREE)
* ... Partner issued SYNCPOINT and LAST
WHEN DFHVALUE(SYNCRECEIVE)
* ... Partner issued SYNCPOINT
WHEN DFHVALUE(FREE)
* ... Partner issued LAST
WHEN DFHVALUE(SEND)
* ... Partner issued INVITE
WHEN DFHVALUE(RECEIVE)
* ... Processing for receipt of data
* (including EIBCOMPL for incomplete data)
WHEN OTHER
* ... Logic error, should never happen
END-EVALUATE.
ELSE
* ... Examine EIBRCODE for source of error
END-IF.
* ...
```

Figure 14. Checking the outcome of a RECEIVE command

Note: In the same way as it is possible to send the INVITE and LAST indicators with data, it is also possible to receive them with data. Syncpoint requests may also be received with data. However, indications of conversation failure are never received with data.

The CONVERSE command
The CONVERSE command combines the functions SEND INVITE and RECEIVE.

T his command is useful when one transaction needs a response from the partner transaction to continue processing.

Safeguarding data integrity

If it is important to safeguard data integrity across connected transactions, then synchronization commands are available.

The commands are:

SYNCPOINT
SYNCPOINT ROLLBACK
SRRCMIT (SAA verb for SYNCPOINT)
SRRBACK (SAA verb for SYNCPOINT ROLLBACK)

The use of these commands in DTP is described in “Syncpointing a distributed process” on page 97.

Ending the conversation

These topics the different ways a conversation can end, either unexpectedly or under transaction control.

The following sections describe To end a transaction, one transaction issues a request for termination and the other receives this request. Once this has happened the conversation is unusable and **both** transactions must issue a FREE command to release the session.

Ending a conversation normally

The SEND LAST command is used to terminate a conversation. It should be used in conjunction with either the WAIT option or the SYNCPOINT command, and followed by the FREE command.

However, SEND LAST WAIT causes the conversation to end before any subsequent syncpoint can be propagated to the partner transaction. This may mean that the protected resources in one system could be committed while those in the other system could be backed out.

From the state table it can be seen that it is possible to end a conversation by issuing the FREE command provided the conversation is in **send state** (state 2). This generates an implicit SEND LAST WAIT command before the FREE is executed and therefore is not recommended.

Note: A distributed transaction should not end a conversation by issuing an EXEC CICS RETURN command, but instead follow the sequence of commands as described. The issue of an EXEC CICS RETURN could lead to one or both transactions ending abnormally.

Unexpected termination of a conversation

If a partner systems fails, or a session goes out of service in the middle of a DTP conversation, the transaction is terminated abnormally.

Checking the outcome of a DTP command

Checking the response from a DTP command can be separated into three stages.

The stages are:

1. Testing for request failure
2. Testing for indicators received on the conversation
3. Testing the conversation state.

Testing for request failure is the same as for other EXEC CICS commands in that conditions are raised and may be handled using HANDLE CONDITION or RESP. EIBRCODE will also contain an error code.

If the request has not failed, it is possible to test for indicators received on the conversation. These are returned to the application in the EIB. The following EIB fields are relevant to all MRO DTP commands. (See EIB fields for programming information on the contents and format of EIB fields.)

EIBFREE
when set to X'FF' indicates that the partner transaction has ended the conversation. It should be tested in conjunction with EIBSYNC to determine exactly how to end the conversation.

EIBSYNC
when set to X'FF' indicates the partner transaction has requested a syncpoint.

EIBSYNRB
when set to X'FF' indicates the partner transaction has issued a SYNCPOINT ROLLBACK command.

Table 21 on page 53 shows how these EIB fields interact.

Table 21. Interaction of some EIB fields

EIB- FREE	EIB- SYNRB	EIB- SYNC	Description
X'00'	X'FF'	X'00'	The partner transaction or system has issued SYNCPOINT ROLLBACK.
X'FF'	X'00'	X'00'	The partner transaction or system has issued SEND LAST followed by a FREE command.
X'FF'	X'00'	X'FF'	The partner transaction or system has issued SEND LAST followed by SYNCPOINT. The local program should reply with a SYNCPOINT command followed by a FREE command.

Table 21. Interaction of some EIB fields (continued)

EIB- FREE	EIB- SYNRB	EIB- SYNC	Description
X'00'	X'00'	X'FF'	The partner transaction or system has issued a SYNCPOINT.

In addition the following EIB fields are relevant only to the RECEIVE and CONVERSE commands:

EIBATT
when set to X'FF' indicates that the data received contained an attach header. The attach header is not passed to the application; however, EIBATT indicates that an EXTRACT ATTACH command is appropriate.

EIBCOMPL
when set to X'FF' indicates that all the data sent at one time has been received. This field is used in conjunction with the RECEIVE NOTRUNCATE command.

EIBFMH
when set to X'FF' indicates that the data passed to the application contains a concatenated Function Management Header (FMH). This happens only when the partner CICS transaction builds an FMH in the data and the FMH option on the SEND command is specified.

EIBRECV
when set to X'00' indicates the partner transaction used the INVITE or LAST option on its last SEND command. When set on (X'FF'), EIBRECV indicates that another RECEIVE is required.

After the EIB fields have been analyzed, it is possible to test the conversation state to determine which DTP commands may be issued next. See "State transitions in MRO conversations" on page 55.

Note: CICS ignores the profile you specify on the PROFILE option of the ALLOCATE for an MRO link and instead uses the default profile. This enables FMHs to be sent and received and EIBATT or EIBFMH to be set appropriately. The default profile DFHCICSA, used for the session allocated by the front-end transaction, has INBFMH (ALL) specified. The default principal facility profile DFHCICST used for the back-end transaction does not have INBFMH (ALL) specified.

Checking EIB fields and the conversation state

Most of the information supplied by the EIB indicator fields can be obtained from the conversation state. However, there are some EIB fields that you cannot ignore.

F or example, when the conversation remains in **receive state** (state 5) after a RECEIVE command has been issued, only EIBFMH indicates that the partner transaction has sent an FMH.

Note that the state table provided in "State transitions in MRO conversations" on page 55 contains not only states and commands issued, but also relevant EIB fields settings. The order in which the EIB fields are shown provides a sensible sequence for checking them in an application.

Summary of commands for MRO conversations

The CICS application programming interface provides a set of commands for use in MRO conversations.

Table 22 on page 54 shows the commands used in MRO conversations.

Table 22. Summary of CICS commands used in MRO conversations

Use to ...	Command	More information
Acquire a session.	ALLOCATE	"Allocating a session to the conversation" on page 47
Build an attach header.	BUILD ATTACH	"Connecting the partner transaction" on page 47
Access session-related information.	EXTRACT ATTACH	"Back-end transaction initiation" on page 48

Table 22. Summary of CICS commands used in MRO conversations (continued)

Use to ...	Command	More information
Send data and control information to the conversation partner.	SEND	"Sending data to the partner transaction" on page 49
Receive data from the conversation partner.	RECEIVE	"Receiving data from the partner transaction" on page 51
Send and receive data on the conversation.	CONVERSE	"The CONVERSE command" on page 52
Inform all conversation partners of readiness to commit recoverable resources.	SYNCPOINT	"Syncpointing a distributed process" on page 97
Inform conversation partners of the need to back out changes to recoverable resources.	SYNCPOINT ROLLBACK	"The SYNCPOINT ROLLBACK command" on page 98
Free the session.	FREE	"Ending a conversation normally" on page 53

For programming information about CICS commands, see CICS command summary.

State transitions in MRO conversations

These topics shows the state transitions that occur when transactions engage in MRO conversations.

The state transitions are presented in the form of a state table. The state table shows which commands a transaction can issue while the conversation is in any given state. It also shows how the conversation state changes as a result of any command.

How to use the state table

The state tables show the commands you can issue, the EIB flags that can be set when the command is issued, and the conversation states.

The commands you can issue, coupled with the EIB flags that can be set after execution, are shown down the first column of the table. These commands correspond to the rows of the table. The possible conversation states are shown across the top of the table. The states correspond to the columns of the table. The intersection of row (command and EIB flag) and column (state) represents the state transition, if any, that occurs when that command returning a particular EIB flag is issued in that state. The order in which EIB flags are shown with a command is the order in which you should test the EIB flags in your program.

A number at an intersection indicates the state number of the next state. Other symbols represent other conditions, as follows:

Symbol	Meaning
N/A	Cannot occur.
×	The EIB flag is any one that has not been covered in earlier rows, or it is irrelevant.
Abend *code*	The command is not valid in this state. Issuing a command in a state in which it is not valid usually causes an AZI1 abend. When a different abend code applies, this is shown in the tables.
=	Remains in current state.
End	End of conversation.

Initial conversation states

Before a session is allocated, there is no conversation, and therefore no conversation state.

The **EXEC CICS ALLOCATE** command gets a session to start a new conversation and does not affect any conversation that is already in progress, hence the **ALLOCATE** command does not appear in the tables. After the **ALLOCATE** command is successfully issued, the new conversation in the front-end transaction is in **ALLOCATED** state.

The back-end transaction starts in **RECEIVE** state after the front-end transaction has successfully initiated the partner transaction.

Testing the conversation state

There are two ways for an application to inquire on the current conversation state.

The first is to use the **EXEC CICS EXTRACT ATTRIBUTES STATE** command and the second is to use the STATE parameter on the DTP commands. In both cases the current state is returned to the application in a CICS-value data area (cvda). Table 23 on page 56 shows how the cvda codes relate to the conversation state. It also shows the symbolic names defined for the cvda values.

Table 23. The conversation states

States used in this book		**States used in DTP programs**	
State name	**State number**	**Symbolic name**	**cvda code**
Allocated	1	DFHVALUE(ALLOCATED)	81
Send	2	DFHVALUE(SEND)	90
Pendfree	4	DFHVALUE(PENDFREE)	86
Receive	5	DFHVALUE(RECEIVE)	88
Syncreceive	9	DFHVALUE(SYNCRECEIVE)	92
Syncfree	11	DFHVALUE(SYNCFREE)	91
Free	12	DFHVALUE(FREE)	85
Rollback	13	DFHVALUE(ROLLBACK)	89

State tables for MRO conversations

Tables showing the state transitions that occur when transactions engage in MRO conversations, under the EXEC CICS API.

The `ISSUE SIGNAL` command and the EIBSIG flag

In the tables, the EIBSIG flag is not mentioned. This is because its use is optional and is entirely a matter of agreement between the two conversation partners. In the worst case, it can occur at any time after every command that affects the EIB flags. However, used for the purpose for which it was intended, it usually occurs after a SEND command. Its priority in the order of testing depends on the role you give it in the application.

The EIBSIG flag is set when the partner issues the **ISSUE SIGNAL** command.

The `RECEIVE NOTRUNCATE` command

The **RECEIVE NOTRUNCATE** command returns a zero value in EIBCOMPL to indicate that the user buffer was too small to contain all the data received from the partner transaction. Normally, you would continue to issue **RECEIVE NOTRUNCATE** commands until the last section of data is passed to you, which is indicated by EIBCOMPL = X'FF' . If NOTRUNCATE is not specified, and the data area specified by the RECEIVE command is too small to contain all the data received, CICS truncates the data and sets the LENGERR condition.

State changes for the SYNCPOINT and SYNCPOINT ROLLBACK commands

When the SYNCPOINT and SYNCPOINT ROLLBACK commands are issued, they are propagated on, and affect the state of, all the conversations that are currently active for the task, including APPC conversations.

Following rollback, the conversation can be in **SEND** or **RECEIVE** state, depending on the conversation state at the start of the current distributed unit of work.

After a syncpoint or rollback, it is advisable to determine the conversation state before issuing any further commands against the conversation. To do this, use the EXTRACT ATTRIBUTES STATE command or the STATE option on the EXEC CICS commands to determine the conversation state.

State tables

Table 24. States 1 - 6

		Command returns	**ALLO-CATED**	**SEND**	**PEND-RECEIVE**	**PEND-FREE**	**RECEIVE**	**CONF-RECEIVE**
Command issued	**EIB flag returned**		**State 1**	**State 2**	**State 3**	**State 4**	**State 5**	**State 6**
BUILD ATTACH	×	Immediately	=	=	N/A	=	Abend	N/A
EXTRACT ATTACH	×	Immediately	=	=	N/A	=	=	N/A
EXTRACT ATTRIBUTES	×	Immediately	=	=	N/A	=	=	N/A
SEND INVITE WAIT	×	After data and CD flows	5	5	N/A	Abend	Abend	N/A
SEND INVITE	×	After data and CD flows	5	5	N/A	Abend	Abend	N/A
SEND LAST WAIT	×	After data and EB flows	12	12	N/A	Abend	Abend	N/A
SEND LAST	×	After data flows	4	4	N/A	Abend	Abend	N/A
SEND	×	After data flows	2	=	N/A	Abend	Abend	N/A
RECEIVE	EIBSYNC + EIBFREE + EIBCOMPL	After sync flow detected	Abend	Abend	N/A	Abend	11	N/A
RECEIVE	EIBSYNC + EIBRECV + EIBCOMPL	After sync flow detected	Abend	Abend	N/A	Abend	9	N/A
RECEIVE	EIBSYNRB + EIBCOMPL	After rollback flow detected	Abend	Abend	N/A	Abend	13	N/A
RECEIVE	EIBFREE	After EB detected	Abend	Abend	N/A	Abend	12	N/A
RECEIVE	EIBRECV	When data available	Abend	Abend	N/A	Abend	=	N/A
RECEIVE NOTRUNCATE	EIBCOMPL	When data available	Abend	Abend	N/A	Abend	=	N/A
RECEIVE	×	When data available	Abend	Abend	N/A	Abend	2	N/A
CONVERSE	As for RECEIVE but allowed in send state		As for RECEIVE but allowed in send state	As for RECEIVE but allowed in send state	As for RECEIVE but allowed in send state	As for RECEIVE but allowed in send state	As for RECEIVE but allowed in send state	As for RECEIVE but allowed in send state
SYNCPOINT	EIBRLDBK	After response from partner	=	2 or 5	N/A	2 or 5	Abend ASP1	N/A
SYNCPOINT	×	After response from partner	=	=	N/A	12	Abend ASP1	N/A
SYNCPOINT ROLLBACK	×	After rollback across UOW	=	2 or 5	N/A	2 or 5	2 or 5	N/A
FREE	×	Immediately	End	End	N/A	End	Abend	N/A

Table 25. States 7 - 13

Command issued	**EIB flag returned**	**CONF-SEND**	**CONF-FREE**	**SYNC-RECEIVE**	**SYNC-SEND**	**SYNC-FREE**	**FREE**	**ROLL-BACK**
		State 7	**State 8**	**State 9**	**State 10**	**State 11**	**State 12**	**State 13**
BUILD ATTACH	×	N/A	N/A	=	N/A	=	=	=
EXTRACT ATTACH	×	N/A	N/A	=	N/A	=	=	=
EXTRACT ATTRIBUTES	×	N/A	N/A	=	N/A	=	=	=
SEND INVITE WAIT	×	N/A	N/A	Abend	N/A	Abend	Abend	Abend
SEND INVITE	×	N/A	N/A	Abend	N/A	Abend	Abend	Abend
SEND LAST WAIT	×	N/A	N/A	Abend	N/A	Abend	Abend	Abend
SEND LAST	×	N/A	N/A	Abend	N/A	Abend	Abend	Abend
SEND	×	N/A	N/A	Abend	N/A	Abend	Abend	Abend
RECEIVE	EIBSYNC + EIBFREE + EIBCOMPL	N/A	N/A	Abend	N/A	Abend	Abend	Abend
RECEIVE	EIBSYNC + EIBRECV + EIBCOMPL	N/A	N/A	Abend	N/A	Abend	Abend	Abend
RECEIVE	EIBSYNRB + EIBCOMPL	N/A	N/A	Abend	N/A	Abend	Abend	Abend
RECEIVE	EIBFREE	N/A	N/A	Abend	N/A	Abend	Abend	Abend
RECEIVE	EIBRECV	N/A	N/A	Abend	N/A	Abend	Abend	Abend
RECEIVE NOTRUNCATE	EIBCOMPL	N/A	N/A	Abend	N/A	Abend	Abend	Abend
RECEIVE	×	N/A	N/A	Abend	N/A	Abend	Abend	Abend
CONVERSE		As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE
SYNCPOINT	EIBRLDBK	N/A	N/A	2 or 5	N/A	2 or 5	=	Abend
SYNCPOINT	×	N/A	N/A	5	N/A	12	=	Abend
SYNCPOINT ROLLBACK	×	N/A	N/A	2 or 5	N/A	2 or 5	=	2 or 5
FREE	×	N/A	N/A	Abend	N/A	Abend	End	Abend

Chapter 4. Writing programs for APPC basic conversations

These topics describe the CICS APIs available for DTP programming using APPC basic conversations.

Conversation initiation

The front-end transaction is responsible for acquiring a session, specifying the conversation characteristics, and requesting the startup of the back-end transaction in the partner system.

Allocating a session to the conversation

Initially, there is no conversation, and therefore no conversation state. By issuing a GDS ALLOCATE command, the front-end transaction acquires a session to start a new conversation.

RETCODE should be checked to ensure that a session has really been allocated. If successfully allocated (RETCODE = X'00'), the conversation is in **allocated state** (state 1) and the session identifier (**convid**) is placed in the data area specified on the CONVID parameter.

The convid must be used in subsequent commands for this conversation. Figure 15 on page 60 shows an example of a GDS ALLOCATE command.

Note: If the remote system is using z/OS Communications Server persistent session support, you may need to code a timeout value on the GDS ALLOCATE command. See Effect of z/OS Communications Server persistent sessions support for DTP conversations on APPC sessions.

Using ATI to allocate a session

Front-end transactions are often initiated from terminals. But it is also possible to use the **EXEC CICS START** command to initiate a front-end transaction on an APPC session.

W hen this is done, and the front-end transaction is successfully started, a conversation can continue as if a GDS ALLOCATE command had been issued. The only difference is that, when ATI is used, the APPC session is the front-end transaction’s principal facility.

```
* ...
EXEC CICS GDS ALLOCATE SYSID(WSYSID) CONVID(WCONVID) *
STATE(WSTATE) RETCODE(WRETC)
*
* Check outcome of GDS ALLOCATE
*
NC WRETC,WRETC
BNZ ALLOCERR No session allocated, check RETCODE
* ...
EXEC CICS GDS CONNECT PROCESS CONVID(WCONVID) *
STATE(WSTATE) *
PROCNAME(WPROC) *
PROCLENGTH(WLENPROC) *
SYNCLEVEL(WSYNCLVL) *
CONVDATA(WCDB) RETCODE(WRETC)
NC WRETC,WRETC
BNZ CONNERR Request failed, analyze RETCODE
* ... No errors, conversation started.
NC CDBERR,CDBERR
BNZ SESSERR Session failed, examine RETCODE.
* ... Start sending data.
* ...
WSTATE DS F
WRETC DS XL6
WCDB DS 0CL24
COPY DFHCDBLK
WCONVID DS CL4
WSYSID DC CL4'SYSB'
WPROC DC CL4'BBBB'
WLENPROC DC F'4'
WSYNCLVL DC F'2'
* ...
```

Figure 15. Starting an APPC basic conversation at sync level 2

Connecting the partner transaction

When the front-end transaction has acquired a session, the next step is to initiate the partner transaction.

The state tables show that, in the **allocated state** (state 1), one of the commands available is GDS CONNECT PROCESS. This command is used to attach the required back-end transaction. It should be noted that the results of the GDS CONNECT PROCESS are placed in the send buffer and are not sent immediately to the partner system. Transmission occurs when the send buffer is flushed, either by sending more data than fits in the send buffer or by issuing a GDS WAIT command.

A successful GDS CONNECT PROCESS causes the conversation state to switch to **send state** (state 2). Figure 15 on page 60 is a program fragment showing an example of a GDS CONNECT PROCESS.

Note: For clarity, the **`EXEC CICS GDS ALLOCATE`** and **`GDS CONNECT PROCESS`** commands shown in Figure 15 on page 60 identify the partner LU and transaction explicitly. To avoid doing this, you could use the PARTNER option of these commands. This specifies a set of definitions that include the names of the partner LU, the communication profile to be used on the session, and the partner transaction. Thus, in Figure 15 on page 60 , the PARTNER option could be used instead of SYSID on the **`EXEC CICS GDS ALLOCATE`** command, and instead of PROCNAME and PROCLENGTH on the **`EXEC CICS GDS CONNECT PROCESS`** command. The advantage of using PARTNER is that it makes your DTP programs more maintainable: the details of each partner program can be held in a single definition.

Initial data for the back-end transaction

While connecting the back-end transaction, the front-end transaction can send initial data to it.

T his kind of data, called **program initialization parameters** (PIPs), is placed in specially formatted structures and specified on the GDS CONNECT PROCESS command. The PIPLIST (along with PIPLENGTH) option of the GDS CONNECT PROCESS command is used to send PIPs to the back-end transaction.

To examine any PIPs received, the back-end transaction uses the GDS EXTRACT PROCESS command.

PIP data is used only by the two connected transactions and not by the CICS systems. APPC systems other than CICS may not support PIP, or may support it differently.

The PIP data must be formatted into one or more subfields according to the SNA-architected rules. The content of each subfield is defined by the application developer. You should format PIP data as follows:

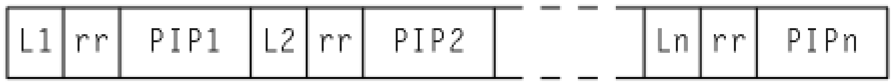

Figure 16. Format of PIP data

CICS inserts information in the reserved fields so that the PIP is architecturally correct. The PIPLENGTH option must specify the total length of the PIP list and must be between 4 and 32763.

Back-end transaction initiation

A back-end transaction is initiated as a result of the front end's **GDS CONNECT PROCESS** command.

Initially the back-end transaction should determine the convid. Figure 17 on page 61 shows a fragment of a back-end transaction that uses the EXEC CICS GDS ASSIGN command to obtain the convid. The back-end transaction can also obtain the transaction identifier and sync level used to start the conversation. The GDS EXTRACT PROCESS command is used to obtain this information.

The back-end transaction starts in **receive state** (state 5). So, after obtaining the convid, the back-end transaction can issue a GDS RECEIVE command.

```
* ...
EXEC CICS GDS ASSIGN PRINCONVID(WCONVID) RETCODE(WRETC)
*
* ...
*
EXEC CICS GDS EXTRACT PROCESS CONVID(WCONVID) *
PROCNAME(WPROC) *
RETCODE(WRETC) *
PROCLENGTH(WLENPROC) *
SYNCLEVEL(WSYNCLVL)
* ...
* Receive first data from front-end transaction.
* ...
*
WSTATE DS F
WRETC DS XL6
WCDB DS 0CL24
COPY DFHCDBLK
WCONVID DS CL4
WPROC DS CL4
WLENPROC DS F
WSYNCLVL DS F
* ...
```

Figure 17. Startup of a back-end transaction

What happens if the back-end transaction fails to start

It is possible that the back-end transaction fails to start. However, because of the transmission delay mechanism in APPC, the front-end transaction is not informed of this fact until the conversation has been active long enough for responses from the back-end system to be received.

The front-end transaction is informed of this via CDBERR and CDBFREE. In addition, CDBERRCD is set as shown in Table 26 on page 62.

Table 26. Some indications of back-end failure

CDBERRCD value	Reason
10086032	The PIP data sent with the GDS CONNECT PROCESS was incorrectly specified.
10086034	The partner system does not support basic conversations.
080F6051	The partner transaction failed security check.
10086041	The partner transaction does not support the sync level requested on the GDS CONNECT PROCESS.
10086021	The partner system does not recognize the requested transaction identifier.
084C0000	The partner system cannot start the partner transaction.
084B6031	The partner system is temporarily unable to start the partner transaction.

Before sending data, the front-end transaction should find out whether the back end transaction has started successfully. One way of doing this is to issue a GDS SEND CONFIRM command directly after the GDS CONNECT PROCESS. This causes the front-end transaction to be suspended until the back end transaction has responded or the back-end system has sent the failure notification as described.

Sending data to the partner transaction

To send data on an APPC basic conversation, an application must format the data into **generalized data stream** (GDS) records.

A GDS record contains a 16-bit (2-byte) header followed by the application data. The 16 bits of the header consist of the following fields:

Concatenation bit
This is the high-order bit of the first byte of the header. An application program can use it to group records together logically. This bit does not affect the way CICS processes the records.

LL
This is the rest of the header (15 bits). It specifies the overall length of the data (including the length of the header).

Figure 18 on page 62 shows the format of GDS records.

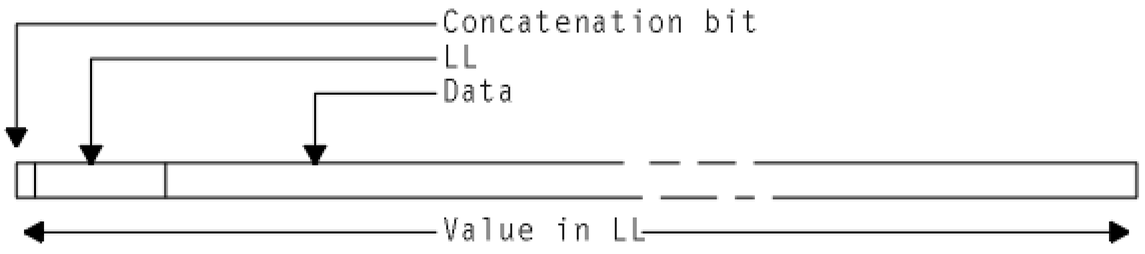

Figure 18. Format of GDS records

Up to 32 765 bytes of application data can be accommodated in one GDS record.

Data formatted into GDS records can be transmitted by the GDS SEND command. This command is valid only in **send state** (state 2).

Because a simple GDS SEND keeps the conversation in **send state** (state 2), you can issue a number of successive sends. You need not issue a GDS SEND for every record to be sent; you can send partial or multiple records at a time. However, make sure that the last logical record is complete when you use the INVITE, LAST, or CONFIRM options, and before you issue a syncpoint request.

Figure 19 on page 63 is an example of the use of GDS SEND commands.

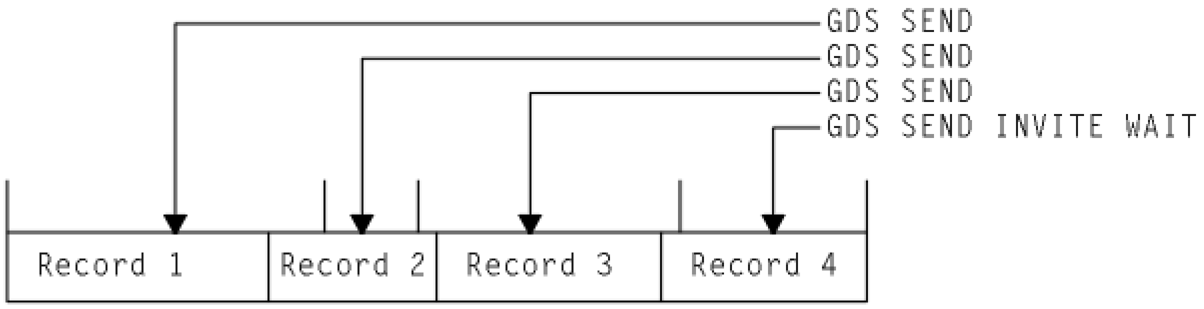

Figure 19. An example of the use of GDS SEND commands

This flexibility also allows you to use separate GDS SEND commands for the GDS header and the application data—a useful technique to avoid shifting data into storage contiguous with its GDS header. The program fragment in Figure 20 on page 63 uses this technique.

```
* ...
LA R5,L'SENDHDR+LEN'SENDDATA Compute LL value
STH R5,SENDHDR Place length in LL
LA R5,L'SENDHDR Length of GDS header
ST R5,SENDLEN into send length field
EXEC CICS GDS SEND FROM(SENDHDR) FLENGTH(SENDLEN) *
CONVID(WCONVID) RETCODE(WRETC) *
STATE(WSTATE) CONVDATA(WCDB)
*
* ... Check outcome of the SEND
* ...
LA R5,L'SENDDATA Length of application data
ST R5,SENDLEN into send length field
EXEC CICS GDS SEND FROM(SENDDATA) FLENGTH(SENDLEN) *
CONVID(WCONVID) RETCODE(WRETC) *
STATE(WSTATE) CONVDATA(WCDB)
*
* ... Check outcome of the SEND
* ...
EXEC CICS GDS SEND INVITE WAIT *
CONVID(WCONVID) RETCODE(WRETC) *
STATE(WSTATE) CONVDATA(WCDB)
*
* ... Check outcome of last command
* ...
*
WSTATE DS F
WRETC DS XL6
WCDB DS 0CL24
COPY DFHCDBLK
WCONVID DS CL4
SENDDATA DS CL100
SENDLEN DS F
SENDHDR DS H
* ...
```

Figure 20. Sending data on an APPC basic conversation

The records from a simple GDS SEND command are initially stored in a local CICS buffer which is “flushed” either when this buffer is full or when the transaction requests transmission. The transaction can request transmission either by using a GDS WAIT command or by using the WAIT option on the GDS SEND command. The reason transmission is deferred is to reduce the number of calls to the network. However, the application should use GDS WAIT if the partner transaction requires the data to continue processing.

Switching from sending to receiving data

To switch from sending to receiving records, use a GDS SEND INVITE command with the WAIT or CONFIRM option.

This command switches the conversation from **send state** (state 2) to **receive state** (state 5). An example of a GDS SEND INVITE WAIT command can be seen in Figure 20 on page 63. Figure 26 on page 76 illustrates the response-testing sequence.

For further information on the CONFIRM option, see "How to synchronize conversations using CONFIRM commands" on page 67.

Receiving data from the partner transaction

The GDS RECEIVE command is used to receive data from the connected partner transaction.

T he rows in the state tables for the GDS RECEIVE command show the CONVDATA fields that should be tested after issuing a GDS RECEIVE command. As well as showing which fields should be tested, the state tables also show the order in which the tests should be made. As an alternative to testing some of the CONVDATA fields it is possible to test the resulting conversation state. This is shown in Figure 25 on page 75. Note that both RETCODE and CDBERR should always be tested.

The amount of data received is determined by:

- How much the conversation partner sent
- The value supplied on the MAXFLENGTH option
- Whether the LLID or BUFFER option is used.

The first factor is obvious: the application cannot receive more than is sent. The value of MAXFLENGTH is an upper limit; CICS never returns more bytes than this value specifies. The LLID and BUFFER options enable the application to specify how CICS is to treat the data. This is described in "Receiving data by the record" on page 65 and "Receiving data by the buffer" on page 66.

In the same way as it is possible to send GDS records with the INVITE, LAST, or CONFIRM option, it is also possible to receive them together. Syncpoint requests can also be received with GDS records. However, GDS ISSUE ERROR, GDS ISSUE ABEND, and indications of conversation failure are received by themselves —never with GDS records.

An example of a GDS RECEIVE command can be seen in Figure 21 on page 65. Figure 25 on page 75 illustrates the response testing sequence.

```
* ...
RECVLOOP DS 0H
LA R5,L'RECVHDR Length of GDS header
ST R5,RECVMAX as maximum receive length
* Receive GDS header from partner transaction
EXEC CICS GDS RECEIVE INTO(RECVHDR) MAXFLENGTH(RECVMAX) *
LLID FLENGTH(RECVLEN) *
CONVID(WCONVID) RETCODE(WRETC) *
STATE(WSTATE) CONVDATA(WCDB)
*
* ... Check outcome of the GDS RECEIVE
* ...
LA R5,L'RECVAREA Length of application buffer
ST R5,RECVMAX as maximum receive length
* Receive application data from partner transaction
EXEC CICS GDS RECEIVE INTO(RECVAREA) MAXFLENGTH(RECVMAX) *
LLID FLENGTH(RECVLEN) *
CONVID(WCONVID) RETCODE(WRETC) *
STATE(WSTATE) CONVDATA(WCDB)
* ...
* ... Check outcome of the GDS RECEIVE
* ... (including CDBCOMPL).
B RECVLOOP Loop while in receive state
* ...
*
WSTATE DS F
WRETC DS XL6
WCDB DS 0CL24
COPY DFHCDBLK
WCONVID DS CL4
RECVAREA DS CL100
RECVMAX DS F
RECVLEN DS F
RECVHDR DS H
* ...
```

Figure 21. Receiving data on an APPC basic conversation

Receiving data by the record

If you specify the LLID option on a GDS RECEIVE command, the data is considered as a series of GDS records. On each GDS RECEIVE request, data is received from not more than one record.

If the record is longer than the value specified in the MAXFLENGTH option, two or more RECEIVE commands are required to recover the whole record. CDBCOMPL is set on when the end of a GDS record has been received. Consider the example shown in Figure 22 on page 65.

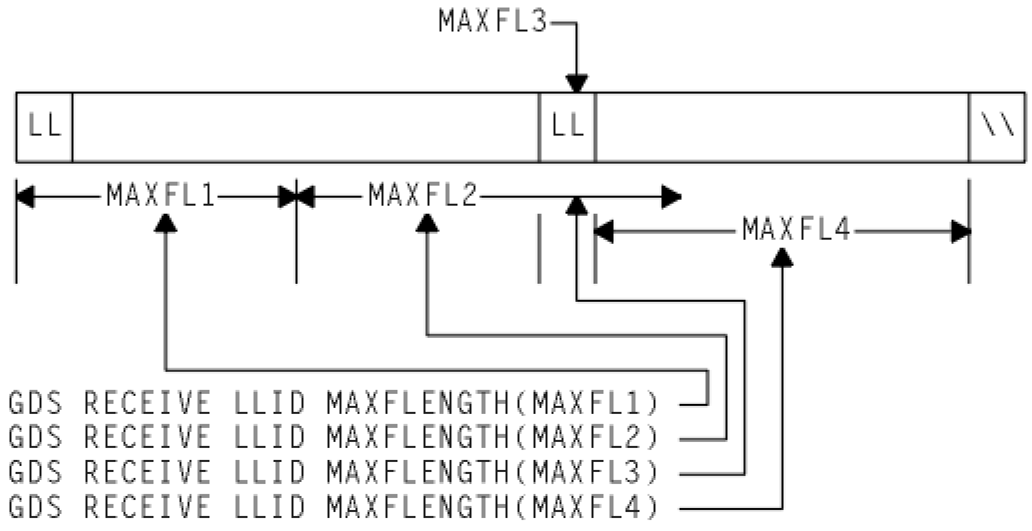
MAXFL3
LL
LL
\\
MAXFL1
MAXFL2
MAXFL4
GDS RECEIVE LLID MAXFLENGTH(MAXFL1)
GDS RECEIVE LLID MAXFLENGTH(MAXFL2)
GDS RECEIVE LLID MAXFLENGTH(MAXFL3)
GDS RECEIVE LLID MAXFLENGTH(MAXFL4)

Figure 22. An example of the effect of the LLID option

The first RECEIVE command receives the front portion of the first record. The length received is restricted by the MAXFLENGTH value (MAXFL1). The second RECEIVE command receives the rest of the first logical record. Even though the MAXFLENGTH value (MAXFL2) allows more data to be received, this cannot be done without breaking the LL boundary rule. The third RECEIVE command is for two bytes of data (the LL field). The fourth RECEIVE command receives the rest of the second record.

The application can tell if a complete record has been received, because CDBCOMPL is set (X'FF'). In the example given, CDBCOMPL is set on after the second and fourth RECEIVE commands. CDBCOMPL is set off (X'00') after the first and third RECEIVE commands.

Receiving data by the buffer

Unlike the LLID option, the BUFFER option does not respect GDS record boundaries.

If the MAXFLENGTH value allows, bytes will be received for more than one record. A GDS RECEIVE command with the BUFFER option recovers the length of data specified in the MAXFLENGTH option, ignoring GDS record boundaries. CICS does not return control to the application program until this length of data has been received or the partner transaction sends the INVITE or LAST option.

Figure 23 on page 66 shows the effect of the BUFFER option on the same four RECEIVE commands discussed in "Receiving data by the record" on page 65.

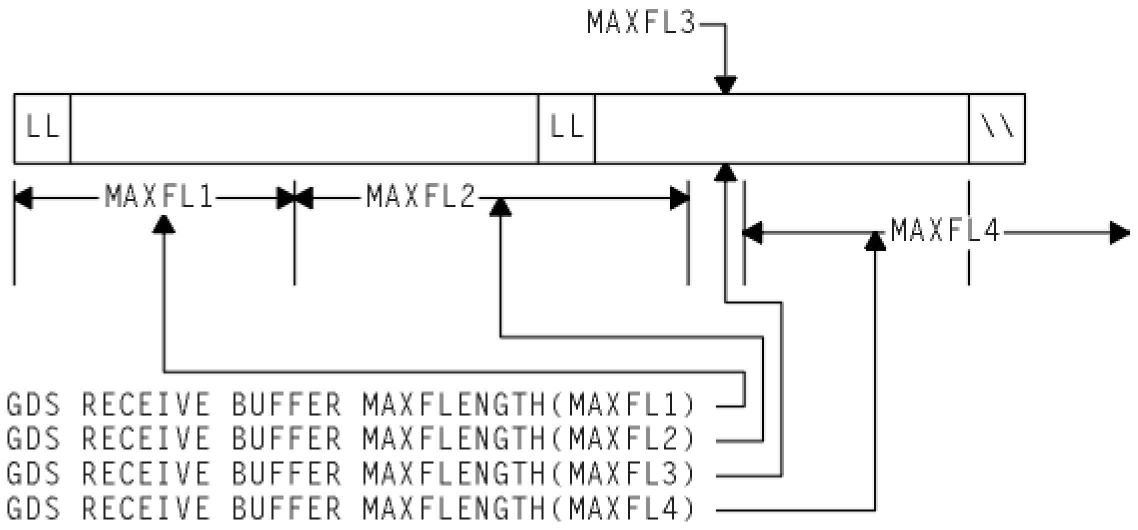
MAXFL3
LL
LL
\\
MAXFL1
MAXFL2
MAXFL4
GDS RECEIVE BUFFER MAXFLENGTH(MAXFL1)
GDS RECEIVE BUFFER MAXFLENGTH(MAXFL2)
GDS RECEIVE BUFFER MAXFLENGTH(MAXFL3)
GDS RECEIVE BUFFER MAXFLENGTH(MAXFL4)

Figure 23. An example of the effect of the BUFFER option

Communicating errors across a conversation

The APPC basic API provides commands to enable transactions to pass error notification across a conversation. There are three commands depending on the severity of the error.

The most severe, GDS ISSUE ABEND, causes the conversation to terminate abnormally and is described in "Emergency termination of a conversation" on page 70 . The other two commands are described in the following section.

Requesting INVITE from the partner transaction

If a transaction is receiving data on a conversation and wants to send, it can use the GDS ISSUE SIGNAL command to request that the partner transaction does a GDS SEND INVITE.

W hen the GDS ISSUE SIGNAL request is received, CDBSIG is set (X'FF'). Note that on receipt of a signal, a transaction is *not* obliged to issue GDS SEND INVITE.

Demanding INVITE from the partner transaction

If a transaction wants to send an immediate error notification to the partner transaction it can use the **`GDS ISSUE ERROR`** command.

This command is also one of the preferred negative responses to GDS SEND CONFIRM. However it should **not** be used to reject GDS ISSUE PREPARE, SYNCPOINT or SYNCPOINT ROLLBACK. When the GDS ISSUE ERROR is received, CDBERR is set (X'FF') and the first two bytes of CDBERRCD are X'0889'.

If a GDS ISSUE ERROR command is used in **receive state** (state 5), all incoming data is purged until an INVITE, SYNCPOINT or LAST is received. If LAST is received, no error indication is sent to the partner transaction, CDBFREE is set (X'FF') and the conversation is switched to **free state** (state 12).

If LAST is not received, the conversation is switched to **send state** (state 2). It is normal to communicate the reason for the error to the partner transaction. The GDS SEND INVITE WAIT command could be used to send an appropriate error message and then a GDS RECEIVE could be used to receive a reply.

Because GDS ISSUE ERROR is allowed in both **send state** (state 2) and **receive state** (state 5), it is possible for both communicating transactions to use GDS ISSUE ERROR at the same time. When this happens, only one of the GDS ISSUE ERROR commands is effective. The other is purged with incoming data. However, both commands will appear to have completed successfully and the transaction whose GDS ISSUE ERROR was purged will pick up CDBERR (= X'FF') on a subsequent command.

Safeguarding data integrity

If it is important to safeguard data integrity across connected transactions, then the CICS synchronization commands are available.

Table 27. Synchronization commands for APPC basic applications

Conversation sync level	Commands
0	None
1	GDS SEND CONFIRM GDS ISSUE CONFIRMATION
2	GDS SEND CONFIRM GDS ISSUE CONFIRMATION SYNCPOINT GDS ISSUE PREPARE SYNCPOINT ROLLBACK SRRCMIT SRRBACK

The SRRCMIT and SRRBACK commands are defined in the following sections. SAA verbs for SYNCPOINT and SYNCPOINT ROLLBACK respectively.

How to synchronize conversations using CONFIRM commands

A confirmation exchange affects a single, specified, conversation and involves two commands.

1. The transaction that is in **send state** (state 2) issues a GDS SEND CONFIRM command causing a request for confirmation to be sent to the partner transaction. The transaction is suspended awaiting a response.
2. The partner transaction receives a request for confirmation. It can then respond positively by issuing a GDS ISSUE CONFIRMATION command. Alternatively, it can respond negatively by using the GDS ISSUE ERROR or GDS ISSUE ABEND commands.

Requesting confirmation
The CONFIRM option on the GDS SEND command flushes the conversation send buffer; that is, it causes a real transmission to occur.

Data can be sent with the GDS SEND CONFIRM command. Either the INVITE or the LAST option can also be specified.

The **send state** (state 2) column of the state table for APPC basic conversations at sync level 1 (see "State tables for APPC basic conversations at sync level 1" on page 80) shows what happens for the possible combinations of the CONFIRM, INVITE, and LAST options. After a GDS SEND CONFIRM command, without the INVITE or LAST options, the conversation remains in **send state** (state 2). If the INVITE option is used, the conversation switches to **receive state** (state 5). If the LAST option is used, the conversation switches to **free state** (state 12).

A similar effect to GDS SEND LAST CONFIRM can by achieved by using the command sequence:

```
 GDS SEND LAST
GDS SEND CONFIRM
```

Note from the state tables that the GDS SEND LAST puts the conversation into **pendfree state** (state 4), so data cannot be sent with a GDS SEND CONFIRM command used in this way.

The form of command used depends on how the conversation is to continue if the required confirmation is received. Whichever is used, the response from GDS SEND CONFIRM *must* always be checked. (See "Checking the response to GDS SEND CONFIRM" on page 68 .)

Receiving and replying to a confirmation request
On receipt of a confirmation request, the CONVDATA and conversation state will be set depending on the request issued by the partner transaction.

The CONVDATA and conversation state, together with the contents of the CDBCONF, CDBRECV, and CDBFREE fields are shown in Table 28 on page 68.

Table 28. How confirmation requests affect the state and flags

Command issued by partner transaction	Conversation state on receipt of request	CDB-CONF on receipt of request	CDB-RECV on receipt of request	CDB-FREE on receipt of request
GDS SEND CONFIRM	confreceive (state 6)	X'FF'	X'FF'	X'00'
GDS SEND INVITE CONFIRM	confsend (state 7)	X'FF'	X'00'	X'00'
GDS SEND LAST CONFIRM	conffree (state 8)	X'FF'	X'00'	X'FF'

There are three ways of replying:

1. Reply positively with a GDS ISSUE CONFIRMATION command.
2. Reply negatively with a GDS ISSUE ERROR command. This reply puts the conversation into **send state** (state 2) regardless of the partner transaction request.
3. Abnormally end the conversation with a GDS ISSUE ABEND command. This makes the conversation unusable and a GDS FREE command must be issued immediately.

Checking the response to GDS SEND CONFIRM
After issuing GDS SEND [INVITE|LAST] CONFIRM, it is important to test CDBERR to determine the partner transaction's response.

Table 29 on page 69 shows the response received when the partner transaction issues different commands.

Table 29. Indicators of the partner transaction's response

Command issued in reply by partner transaction	Conversation state	CDBERR	CDBFREE
GDS ISSUE CONFIRMATION	Dependent on original GDS SEND [INVITE|LAST] CONFIRM request	X'00'	X'00'
GDS ISSUE ERROR	Receive (state 5)	X'FF'	X'00'
GDS ISSUE ABEND	Free (state 12)	X'FF'	X'FF'

If CDBERR= X'00' , the partner transaction has replied GDS ISSUE CONFIRMATION.

If the partner transaction replies GDS ISSUE ERROR, this is indicated by CDBERR (= X'FF') and the first two bytes of CDBERRCD= X'0889'. When the partner transaction replies GDS ISSUE ERROR in response to GDS SEND LAST CONFIRM, the LAST option is ignored and the conversation is *not* terminated. The conversation is switched to **receive state** (state 5).

If the partner transaction replies GDS ISSUE ABEND, both CDBERR and CDBFREE are both set (X'FF'), and the first two bytes of CDBERRCD contain X'0864'. The conversation is switched to **free state** (state 12).

Ending the conversation

These topics describe the different ways a conversation can end, either unexpectedly or under transaction control.

To end a transaction, one transaction issues a request for termination and the other receives this request. Once this has happened the conversation is unusable and **both** transactions must issue a GDS FREE command to release the session.

Normal termination of a conversation

The GDS SEND LAST command is used to terminate a conversation. It should be used in conjunction with either the WAIT or CONFIRM options or the SYNCPOINT command (depending on the conversation sync level).

A distributed transaction should not end a conversation by issuing an **EXEC CICS RETURN** command, but instead follow the sequence of commands shown. The issue of an **EXEC CICS RETURN** could lead to one or both transactions ending abnormally.

Table 30. Terminating commands for different sync levels

Sync level	Command sequence
0	GDS SEND LAST WAIT GDS FREE
1	GDS SEND LAST CONFIRM GDS FREE
2	GDS SEND LAST SYNCPOINT GDS FREE

Note: It is important that the GDS SEND LAST command for sync level 2 is **not** accompanied by WAIT or CONFIRM because either of these options will cause the conversation to end before the subsequent syncpoint has propagated to the partner transaction. This may mean that protected resources of one transaction could be committed while those in the partner transaction could be backed out. The resulting state errors may also lead to the session being unbound.

Emergency termination of a conversation

The GDS ISSUE ABEND command provides a means of abnormally ending the conversation. It is valid for all levels of synchronization, but should be avoided at sync level 2, because its use at the wrong time can lead to a loss of data integrity.

GDS ISSUE ABEND can be issued by either transaction, whether it is in send or receive state, at any time after the conversation has started. For a transaction in **send state** (state 2), any deferred data that is waiting for transmission is flushed before the GDS ISSUE ABEND command is transmitted.

The transaction that issues the GDS ISSUE ABEND command is not itself abended. It must, however, issue a FREE command for the conversation unless it is designed to terminate immediately.

If a GDS ISSUE ABEND command is issued in **receive state** (state 5), CICS purges all incoming data until an INVITE, syncpoint request, or LAST indicator is received. If LAST is received, no abend indication is sent to the partner transaction.

If a GDS ISSUE ABEND is received, both CDBERR and CDBFREE set (X'FF'), the first two bytes of CDBERRCD contain X'0864' . The only command that can be subsequently issued for the conversation is GDS FREE.

Unexpected termination of a conversation

If a partner systems fails or a session goes out of service in the middle of a DTP conversation, the conversation is terminated abnormally and the application informed the next time a command accesses the session.

In addition, both CDBERR and CDBFREE are set on (X'FF'), and CDBERRCD contains one of the following values representing the reason for the error.

X'08640001'
partner system with persistent session support has failed and restarted

X'1008600B'
session has failed due to a protocol error

X'A0000100'
temporary session failure

X'A0010100'

Checking the outcome of GDS commands

The CICS exec interface block (EIB) is not affected by EXEC CICS GDS commands, and no CICS conditions can be raised when EXEC CICS GDS commands are executed. Instead, you must provide data areas in your application to receive return codes and session status information.

The data areas required are:

- A 6-byte area to receive RETCODE information
- A 24-byte area to receive CONVDATA information.

Within the bounds of the programming language you are using, you can give these areas any identifiers you like. They must be named explicitly in most EXEC CICS GDS commands.

Checking the response from a GDS command can be separated into three stages:

1. Testing for request failure; this involves testing RETCODE.
2. Testing for indicators received on the conversation. These indicators are found in CONVDATA.
3. Testing the conversation state.

Testing for request failure

The RETCODE area is used to detect any errors that occur when an EXEC CICS GDS command is executed. These errors correspond to CICS exception conditions, such as NOTALLOC, that can be raised when EXEC CICS commands are executed.

These errors usually reflect failure of the request. Figure 24 on page 71 shows the possible hexadecimal values for the first three bytes of RETCODE. These values are structured so that the first byte indicates the general error description and subsequent bytes provide the detail.

```
00 .. .. Normal return code
01 .. .. ALLOCATE failure (applicable only to GDS ALLOCATE)



01 04 .. SYSBUSY, unknown modename, task cancelled
01 04 04 No bound contention winner available (SYSBUSY)
01 04 08 Modename not known on this system
01 04 0C Attempt to use reserved modename SNASVCMG, or no COS
table in z/OS Communications Server for the modename
01 04 10 Task cancelled during queuing of ALLOCATE
01 04 14 The requested modegroup is closed
01 04 18 The requested modegroup is draining
01 08 .. SYSID is out of service
01 08 00 Connection out of service or in quiesce state, no
free sessions in requested modegroup, or z/OS
Communications Server ACB is closed
01 08 04 Maximum number of queued ALLOCATE requests specified
on QUEUELIMIT CONNECTION parameter exceeded
01 08 08 ALLOCATE queue purged because MAXQTIME would be
exceeded
01 0C .. SYSID is not known in TCT
01 0C 00 SYSID name is not known
01 0C 04 SYSID name is not that of an APPC connection
01 0C 14 NETNAME specified in PARTNER definition is not known
02 0C 00 PARTNER is not known
03 .. .. INVREQ error
03 00 .. Session is either not defined as APPC, in use by
CPI Communications, or (for EXTRACT PROCESS) not
the principal facility
03 04 .. GDS command issued on a conversation that is not basic
03 08 .. Command issued in wrong state
03 0C .. Sync level cannot be supported or cannot support the
command issued
03 10 .. LL error on a GDS SEND
03 14 .. SEND CONFIRM or ISSUE CONFIRMATION used at sync level 0
03 24 .. GDS ISSUE PREPARE used in wrong state
04 .. .. NOTALLOC error (CONVID specifies an unallocated session)
05 .. .. LENGERR error (FLENGTH, MAXFLENGTH, PROCLENGTH, PIPLENGTH,
or MAXPROCLEN error)
06 00 00 PROFILE specified in PARTNER definition is not known
```

Figure 24. RETCODE values

Testing indicators

When RETCODE shows a normal return code from a GDS command, the CONVDATA area (where applicable) contains information on the indicators received on the conversation. These indicators can be used to find out why the conversation state is what it is.

The structure of the CONVDATA area is shown in Table 31 on page 71.

Table 31. Structure of the conversation data block

Field name	Length (bytes)	Meaning
CDBCOMPL	1	X'FF' = data complete
CDBSYNC	1	X'FF' = SYNCPOINT required

Table 31. Structure of the conversation data block (continued)

Field name	Length (bytes)	Meaning
CDBFREE	1	X'FF' = FREE required
CDBRECV	1	X'FF' = RECEIVE required
CDBSIG	1	X'FF' = SIGNAL received
CDBCONF	1	X'FF' = CONFIRM received
CDBERR	1	X'FF' = ERROR received
CDBERRCD	4	Error code (when CDBERR set)
CDBSYNRB	1	X'FF' = SYNCPOINT ROLLBACK required
CDBRSVD	12	Reserved

These definitions are provided in copybook DFHCDBLK. There is one copybook for C, which defines a *typedef* for the structure, and another copybook for assembler. To provide the flexibility to enable your application to manage more than one conversation at the same time, the assembler version does not contain a DSECT statement.

The meanings of the CONVDATA fields are as follows:

CDBERR
when set to X'FF' indicates an error has occurred on the conversation. The reason is in CDBERRCD. This could be as a result of a GDS ISSUE ERROR, GDS ISSUE ABEND, or SYNCPOINT ROLLBACK command issued by the partner transaction. CDBERR can be set as a result of any command that can be issued while the conversation is in **receive state** (state 5), or following any command that causes a transmission to the partner system. It is safest to test CDBERR in conjunction with CDBFREE and CDBSYNRB after every GDS command.

CDBERRCD
contains the reason for CDBERR. If CDBERR is not set, this field is not used.

CDBFREE
when set to X'FF' indicates that the partner transaction had ended the conversation. It should be tested along with CDBERR and CDBSYNC to find out exactly how to end the conversation.

CDBSIG
when set to X'FF' indicates the partner transaction or system has issued and GDS ISSUE SIGNAL command.

CDBSYNRB
when set to X'FF' indicates the partner transaction or system has issued a SYNCPOINT ROLLBACK command. (This is relevant only for conversations at sync level 2.)

Table 32 on page 72 shows how these CDB fields interact.

Table 32. Interaction between some CDB fields—all DTP commands

CDB- ERR	CDB- FREE	CDB- SYNRB	CDBERRCD	Description
X'FF'	X'00'	X'00'	X'08890000' X'08890001'	The partner transaction has sent GDS ISSUE ERROR
X'FF'	X'00'	X'00'	X'08890100' X'08890101'	The partner system has sent GDS ISSUE ERROR
X'FF'	X'00'	X'00'	X'A0020000'	Error in data received from partner

Table 32. Interaction between some CDB fields—all DTP commands (continued)				
CDB- ERR	**CDB-FREE**	**CDB-SYNRB**	**CDBERRCD**	**Description**
X'FF'	X'FF'	X'00'	X'08640000'	The partner transaction has sent GDS ISSUE ABEND
X'FF'	X'FF'	X'00'	X'08640001'	The partner system has sent GDS ISSUE ABEND
X'FF'	X'FF'	X'00'	X'08640002'	A partner resource has timed out
X'FF'	X'FF'	X'00'	X'1008600B'	The session has failed due to a protocol error
X'FF'	X'FF'	X'00'	X'A0000100'	A temporary session failure
X'FF'	X'FF'	X'00'	X'A0010100'	RTIMOUT has triggered
X'FF'	X'FF'	X'00'	X'10086032'	The PIP data sent with the GDS CONNECT PROCESS was incorrectly specified
X'FF'	X'FF'	X'00'	X'10086034'	The partner system does not support basic conversations
X'FF'	X'FF'	X'00'	X'080F6051'	The partner transaction failed security check
X'FF'	X'FF'	X'00'	X'10086041'	The partner transaction does not support the sync level requested on the GDS CONNECT PROCESS
X'FF'	X'FF'	X'00'	X'10086021'	The partner transactions name is not recognized by the partner system
X'FF'	X'FF'	X'00'	X'084C0000'	The partner system cannot start partner transaction
X'FF'	X'FF'	X'00'	X'084B6031'	The partner system is temporarily unable to start the partner transaction
X'FF'	X'00'	X'FF'	X'08240000'	The partner transaction or system has issued SYNCPOINT ROLLBACK
X'00'	X'00'	—	—	The command completed successfully

In addition, the following CONVDATA fields are relevant only to GDS RECEIVE commands:

CDBCOMPL
: when set to X'FF' indicates that all the data sent at one time has been received. This field is used in conjunction with the GDS RECEIVE LLID command.

CDBCONF
: when set to X'FF' indicates that the partner transaction has issued a GDS SEND CONFIRM command and requires a response.

CDBRECV
: is only used when CDBERR is not set. When CDRECV is on (X'FF'), another GDS RECEIVE is required.

CDBSYNC
: when set to X'FF' indicates that the partner transaction or system has requested a syncpoint. (This is relevant only for conversations at sync level 2.)

Table 33 on page 73 shows how some of these CDB fields interact for RECEIVE commands.

Table 33. Interaction between some CDB fields—RECEIVE commands only					
CDB-ERR	**CDB-FREE**	**CDB-RECV**	**CDB-SYNC**	**CDB-CONF**	**Description**
X'00'	X'00'	X'00'	X'00'	X'00'	The partner transaction or system has issued GDS SEND INVITE WAIT. The local program is now in send state.

Table 33. Interaction between some CDB fields—RECEIVE commands only (continued)

CDB-ERR	CDB-FREE	CDB-RECV	CDB-SYNC	CDB-CONF	Description
X'00'	X'00'	X'00'	X'FF'	X'00'	The partner transaction or system has issued GDS SEND INVITE, followed by a SYNCPOINT. The local program is now in syncsend state.
X'00'	X'00'	X'00'	X'00'	X'FF'	The partner transaction or system has issued GDS SEND INVITE CONFIRM. The local program is now in confsend state.
X'00'	X'00'	X'FF'	X'00'	X'00'	The partner transaction or system has issued GDS SEND or GDS SEND WAIT. The local program is in receive state.
X'00'	X'00'	X'FF'	X'FF'	X'00'	The partner transaction or system has issued a SYNCPOINT. The local program is in syncreceive state.
X'00'	X'00'	X'FF'	X'00'	X'FF'	The partner transaction or system has issued a GDS SEND CONFIRM. The local program is in confreceive state.
X'00'	X'FF'	X'00'	X'00'	X'00'	The partner transaction or system has issued a GDS SEND LAST WAIT. The local program is in free state.
X'00'	X'FF'	X'00'	X'FF'	X'00'	The partner transaction or system has issued a GDS SEND LAST followed by a SYNCPOINT. The local program is in syncfree state.
X'00'	X'FF'	X'00'	X'00'	X'FF'	The partner transaction or system has issued a GDS SEND LAST CONFIRM. The local program is in conffree state.

After analyzing the CONVDATA fields, you can test the conversation state to find out which GDS commands you can issue next. See "State transitions in APPC basic conversations" on page 77.

Checking CONVDATA fields and the conversation state

Most of the information supplied by the CONVDATA fields can also be obtained from the conversation state. However, you must also check CDBERR and CDBERRCD.

For example, if after a GDS SEND INVITE WAIT or a GDS RECEIVE command has been issued, the conversation is in **receive state** (state 5), only CDBERR indicates that the partner transaction has sent a GDS ISSUE ERROR. This is illustrated in Figure 25 on page 75 and Figure 26 on page 76.

It should be noted that the state tables provided contain not only conversation states and commands issued, but also relevant CONVDATA field settings. The order in which these fields are shown provides a sensible sequence of checks for an application.

```
* ...
* Check return code from RECEIVE
NC WRETC,WRETC
BNZ BADRET Request-related error, analyze
* ... Request successful
NC CDBERR,CDBERR
BNZ ERROR Error indicated, analyze
* ... No errors, check state
CLC WSTATE,DFHVALUE(SYNCFREE)
BE OKSYNFR Partner issued SYNCPOINT and LAST
CLC WSTATE,DFHVALUE(SYNCRECEIVE)
BE OKSYNRC Partner issued SYNCPOINT
CLC WSTATE,DFHVALUE(SYNCSEND)
BE OKSYNSE Partner issued SYNCPOINT and INVITE
CLC WSTATE,DFHVALUE(CONFFREE)
BE OKCONFR Partner issued CONFIRM and LAST
CLC WSTATE,DFHVALUE(CONFRECEIVE)
BE OKCONRC Partner issued CONFIRM
CLC WSTATE,DFHVALUE(CONFSEND)
BE OKCONSE Partner issued CONFIRM and INVITE
CLC WSTATE,DFHVALUE(FREE)
BE OKFREE Partner issued LAST
CLC WSTATE,DFHVALUE(SEND)
BE OKSEND Partner issued INVITE
CLC WSTATE,DFHVALUE(RECEIVE)
BE OKRECV Processing for receipt of data
* (including CDBCOMPL for incomplete data)
B LOGICERR Logic error, should never happen
* ...
ERROR DS 0H
* Error indicated
CLC WSTATE,DFHVALUE(ROLLBACK)
BE ERRRLBK ROLLBACK received
CLC WSTATE,DFHVALUE(FREE)
BE ERRFREE ISSUE ABEND & TERMERR received,
* reason in CDBERRCD
CLC WSTATE,DFHVALUE(RECEIVE)
BE ERRRECV ISSUE ERROR received,
* reason in CDBERRCD
B LOGICERR Logic error, should never happen
* ...
BADRET DS 0H
* ... Examine RETCODE for source of error
* ...
WSTATE DS F
WRETC DS XL6
WCDB DS 0CL24
COPY DFHCDBLK
* ...
```

Figure 25. Checking the outcome of a GDS RECEIVE command

```
* ...
* Check return code from SEND INVITE WAIT
NC WRETC,WRETC
BNZ BADRET Request-related error, analyze RETCODE
* ... Request successful
NC CDBERR,CDBERR
BNZ ERROR Error indicated, analyze state
* ... No errors, check state
CLC WSTATE,DFHVALUE(RECEIVE)
BE OKRECV Processing for receipt of data
* (incl. CDBCOMPL for incomplete data)
B LOGICERR Logic error, should never happen
* ...
ERROR DS 0H
* Error indicated
CLC WSTATE,DFHVALUE(ROLLBACK)
BE ERRRLBK ROLLBACK received
CLC WSTATE,DFHVALUE(FREE)
BE ERRFREE ISSUE ABEND & TERMERR received,
* reason in CDBERRCD
CLC WSTATE,DFHVALUE(RECEIVE)
BE ERRRECV ISSUE ERROR received,
* reason in CDBERRCD
B LOGICERR Logic error, should never happen
* ...
BADRET ... Examine RETCODE for source of error
* ...
*
WSTATE DS F
WRETC DS XL6
WCDB DS 0CL24
COPY DFHCDBLK
* ...
```

Figure 26. Checking the outcome of a GDS SEND INVITE WAIT command

Summary of commands for APPC basic conversations

The CICS application programming interface provides a set of commands for use in APPC basic conversations.

Table 34. Summary of commands used in basic conversations

Use to ...	**Sync levels**	**Command**	**More information**
Acquire a session to the partner system.	0,1,2	GDS ALLOCATE	"Allocating a session to the conversation" on page 59
Initiate a conversation with a named process on the partner system.	0,1,2	GDS CONNECT PROCESS	"Connecting the partner transaction" on page 60
Obtain the session and connection identifiers of the transaction's principal facility.	0,1,2	GDS ASSIGN	"Back-end transaction initiation" on page 61
Access session-related information in the attach header that initiated the transaction.	0,1,2	GDS EXTRACT PROCESS	"Back-end transaction initiation" on page 61
Send data and control information to the conversation partner.	0,1,2	GDS SEND	"Sending data to the partner transaction" on page 62

Table 34. Summary of commands used in basic conversations (continued)

Use to ...	Sync levels	Command	More information
Receive data from the conversation partner.	0,1,2	GDS RECEIVE	"Receiving data from the partner transaction" on page 64
Transmit any deferred data or control indicators.	0,1,2	GDS WAIT	"Sending data to the partner transaction" on page 62
Reply positively to GDS SEND CONFIRM.	1,2	GDS ISSUE CONFIRMATION	"Receiving and replying to a confirmation request" on page 68
Prepare a conversation partner for syncpointing.	2	GDS ISSUE PREPARE	"The ISSUE PREPARE command" on page 97
Inform the conversation partner of a program-detected error.	0,1,2	GDS ISSUE ERROR	"Receiving and replying to a confirmation request" on page 68
Signal an unusual condition to the conversation partner, usually against the flow of data.	0,1,2	GDS ISSUE SIGNAL	"Communicating errors across a conversation" on page 66
Inform the conversation partner that the conversation should be abandoned.	0,1,2	GDS ISSUE ABEND	"Emergency termination of a conversation" on page 70
Free the session.	0,1,2	GDS FREE	"Ending the conversation" on page 69
Inform all a transaction's conversation partners that it is ready to commit its recoverable resources.	2	SYNCPOINT	"Syncpointing a distributed process" on page 97
Inform all a transaction's conversation partners that it wants to back out changes to recoverable resources.	2	SYNCPOINT ROLLBACK	"The SYNCPOINT ROLLBACK command" on page 98

State transitions in APPC basic conversations

These topics show how the state changes when GDS commands are issued in APPC basic conversations.

T he state transitions are presented in the form of state tables showing which commands can be issued while the conversation is in any given state. The tables also show how the conversation state changes as a result of a command.

How to use the state tables

The state tables show the commands you can issue, the CDB flags that can be set when the command is issued, and the conversation states.

The commands you can issue, coupled with the CDB flags that can be set after execution, are shown in the first column of the table. The possible conversation states are shown across the top of the table. The states correspond to the columns of the table. The intersection of a row (command and CDB flag) and a column (state) represents the state transition, if any, that occurs when a particular command, issued in a particular state, returns a particular CDB flag. The order in which the CDB flags appear with a command also shows the order in which you test the CDB flags in your program.

A number at an intersection indicates the next state. Other symbols represent other conditions, as follows:

Symbol	Meaning
N/A	Cannot occur.
×	The CDB flag is any one that has not been covered in earlier rows, or it is irrelevant (but see the note on CDBSIG if you want to use GDS ISSUE SIGNAL).
Abend	The command is not valid in this state. Issuing a command in a state in which it is not valid causes a bad response to be returned.
=	Remains in current state.
End	End of conversation.

Initial conversation states

Before a session is allocated, there is no conversation, and therefore no conversation state.

T he **`EXEC CICS GDS ALLOCATE`** command gets a session to start a new conversation and does not affect any conversation that is already in progress, hence the **`GDS ALLOCATE`** command does not appear in the tables. After the **`GDS ALLOCATE`** command is successfully issued, the new conversation in the front-end transaction is in **`ALLOCATED`** state.

The back-end transaction starts in **`RECEIVE`** state after the front-end transaction has successfully issued the **`GDS CONNECT PROCESS`** command.

Testing the conversation state

There are two ways for an application to inquire on the current conversation state. The first is to use the EXEC CICS GDS EXTRACT ATTRIBUTES STATE command and the second is to use the STATE parameter on the GDS commands.

In both cases the current state is returned to the application in a CICS value data area (cvda). Table 35 on page 78 shows how the cvda codes relate to the conversation state. The table also shows the symbolic names defined for the cvda values.

Table 35. The conversation states

States used in this book		States used in DTP programs	
State name	**State number**	**Symbolic name**	**cvda code**
Allocated	1	DFHVALUE(ALLOCATED)	81
Send	2	DFHVALUE(SEND)	90
Pendreceive	3	DFHVALUE(PENDRECEIVE)	87
Pendfree	4	DFHVALUE(PENDFREE)	86
Receive	5	DFHVALUE(RECEIVE)	88
Confreceive	6	DFHVALUE(CONFRECEIVE)	83
Confsend	7	DFHVALUE(CONFSEND)	84
Conffree	8	DFHVALUE(CONFFREE)	82
Syncreceive	9	DFHVALUE(SYNCRECEIVE)	92
Syncsend	10	DFHVALUE(SYNCSEND)	93
Syncfree	11	DFHVALUE(SYNCFREE)	91
Free	12	DFHVALUE(FREE)	85
Rollback	13	DFHVALUE(ROLLBACK)	89

State tables for APPC basic conversations at sync level 0

Tables showing the state transitions that occur when transactions engage in APPC basic (or *unmapped*) conversations at sync level 0, under the EXEC CICS API.

The GDS ISSUE SIGNAL command and the CDBSIG flag

In the tables, the CDBSIG flag is not mentioned. This is because its use is optional and is entirely a matter of agreement between the two conversation partners. In the worst case, it can occur at any time after every command that affects the CDB flags. However, used for the purpose for which it was intended, it usually occurs after a GDS SEND command. Its priority in the order of testing depends on the role you give it in the application.

The CDBSIG flag is set when the partner issues the **`GDS ISSUE SIGNAL`** command.

State tables

Table 36. States 1 - 6

		Command returns	ALLO-CATED	SEND	PEND-RECEIVE	PEND-FREE	RECEIVE	CONF-RECEIVE
Command issued	**CDB flag returned**		**State 1**	**State 2**	**State 3**	**State 4**	**State 5**	**State 6**
GDS CONNECT PROCESS	EIBERR + EIBFREE	Immediately	12	Abend	Abend	Abend	Abend	N/A
GDS CONNECT PROCESS	×	Immediately	2	Abend	Abend	Abend	Abend	N/A
GDS EXTRACT PROCESS (back-end transaction only)	×	Immediately	=	=	=	=	=	N/A
GDS EXTRACT ATTRIBUTES	×	Immediately	=	=	=	=	=	N/A
GDS SEND (any valid form)	CDBERR + CDBFREE	After error detected	Abend	12	Abend	Abend	Abend	N/A
GDS SEND (any valid form)	CDBERR	After error detected	Abend	5	Abend	Abend	Abend	N/A
GDS SEND INVITE WAIT	×	After data flows	Abend	5	Abend	Abend	Abend	N/A
GDS SEND INVITE	×	After data buffered	Abend	3	Abend	Abend	Abend	N/A
GDS SEND LAST WAIT	×	After data flows	Abend	12	Abend	Abend	Abend	N/A
GDS SEND LAST	×	After data buffered	Abend	4	Abend	Abend	Abend	N/A
GDS SEND WAIT	×	After data flows	Abend	=	Abend	Abend	Abend	N/A
GDS SEND	×	After data buffered	Abend	=	Abend	Abend	Abend	N/A
GDS RECEIVE	CDBERR + CDBFREE	After error detected	Abend	Abend	Abend	Abend	12	N/A
GDS RECEIVE	CDBERR	After error detected	Abend	Abend	Abend	Abend	=	N/A
GDS RECEIVE	CDBFREE	After error detected	Abend	Abend	Abend	Abend	12	N/A
GDS RECEIVE	CDBRECV	When data available	Abend	Abend	Abend	Abend	=	N/A
GDS RECEIVE LLID	CDBCOMPL	When data available	Abend	Abend	Abend	Abend	=	N/A
GDS RECEIVE	×	When data available	Abend	Abend	Abend	Abend	2	N/A
GDS ISSUE ERROR	CDBFREE	After response from partner	Abend	12	12	Abend	12	N/A
GDS ISSUE ERROR	×	After response from partner	Abend	=	2	Abend	2	N/A
GDS ISSUE ABEND	×	Immediately	Abend	12	12	12	12	N/A
GDS ISSUE SIGNAL	×	Immediately	Abend	=	=	Abend	=	N/A

Table 36. States 1 - 6 (continued)

Command issued	CDB flag returned	Command returns	ALLO-CATED	SEND	PEND-RECEIVE	PEND-FREE	RECEIVE	CONF-RECEIVE
			State 1	**State 2**	**State 3**	**State 4**	**State 5**	**State 6**
GDS WAIT	×	Immediately	Abend	=	5	12	Abend	N/A
GDS FREE	×	Immediately	End	Abend	Abend	End	Abend	N/A

Table 37. States 7 -13

Command issued	CDB flag returned	CONF- SEND	CONF-FREE	SYNC-RECEIVE	SYNC-SEND	SYNC-FREE	FREE	ROLL-BACK
		State 7	**State 8**	**State 9**	**State 10**	**State 11**	**State 12**	**State 13**
GDS CONNECT PROCESS	EIBERR + EIBFREE	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS CONNECT PROCESS	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS EXTRACT PROCESS (back-end transaction only)	×	N/A	N/A	N/A	N/A	N/A	=	N/A
GDS EXTRACT ATTRIBUTES	×	N/A	N/A	N/A	N/A	N/A	=	N/A
GDS SEND (any valid form)	CDBERR + CDBFREE	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS SEND (any valid form)	CDBERR	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS SEND INVITE WAIT	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS SEND INVITE	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS SEND LAST WAIT	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS SEND LAST	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS SEND WAIT	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS SEND	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS RECEIVE	CDBERR + CDBFREE	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS RECEIVE	CDBERR	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS RECEIVE	CDBFREE	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS RECEIVE	CDBRECV	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS RECEIVE LLID	CDBCOMPL	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS RECEIVE	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS ISSUE ERROR	CDBFREE	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS ISSUE ERROR	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS ISSUE ABEND	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS ISSUE SIGNAL	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS WAIT	×	N/A	N/A	N/A	N/A	N/A	Abend	N/A
GDS FREE	×	N/A	N/A	N/A	N/A	N/A	End	N/A

State tables for APPC basic conversations at sync level 1

Tables showing the state transitions that occur when transactions engage in APPC basic (or *unmapped*) conversations at sync level 1, under the EXEC CICS API.

The GDS ISSUE SIGNAL command and the CDBSIG flag

In the tables, the CDBSIG flag is not mentioned. This is because its use is optional and is entirely a matter of agreement between the two conversation partners. In the worst case, it can occur at any time after every command that affects the CDB flags. However, used for the purpose for which it was intended, it usually occurs after a GDS SEND command. Its priority in the order of testing depends on the role you give it in the application.

The CDBSIG flag is set when the partner issues the **GDS ISSUE SIGNAL** command.

State tables

Table 38. States 1 - 6

Command issued	CDB flag returned	Command returns	ALLO-CATED	SEND	PEND-RECEIVE	PEND-FREE	RECEIVE	CONF-RECEIVE
			State 1	State 2	State 3	State 4	State 5	State 6
GDS CONNECT PROCESS	EIBERR + EIBFREE	Immediately	12	Abend	Abend	Abend	Abend	Abend
GDS CONNECT PROCESS	×	Immediately	2	Abend	Abend	Abend	Abend	Abend
GDS EXTRACT PROCESS	×	Immediately	=	=	=	=	=	=
GDS EXTRACT ATTRIBUTES	×	Immediately	=	=	=	=	=	=
GDS SEND (any valid form)	CDBERR + CDBFREE	After error flow detected	Abend	12	Abend	12	Abend	Abend
GDS SEND (any valid form)	CDBFREE	After error flow detected	Abend	12	Abend	Abend	Abend	Abend
GDS SEND INVITE WAIT	×	After data flows	Abend	5	Abend	Abend	Abend	Abend
GDS SEND INVITE CONFIRM	×	After response from partner	Abend	5	Abend	Abend	Abend	Abend
GDS SEND INVITE	×	After data buffered	Abend	3	Abend	Abend	Abend	Abend
GDS SEND LAST WAIT	×	After data flows	Abend	12	Abend	Abend	Abend	Abend
GDS SEND LAST CONFIRM	×	After response from partner	Abend	12	Abend	Abend	Abend	Abend
GDS SEND LAST	×	After data buffered	Abend	4	Abend	Abend	Abend	Abend
GDS SEND WAIT	×	After data flows	Abend	=	Abend	Abend	Abend	Abend
GDS SEND CONFIRM	×	After response from partner	Abend	=	5	12	Abend	Abend
GDS SEND	×	After data buffered	Abend	=	Abend	Abend	Abend	Abend
GDS RECEIVE	CDBERR + CDBFREE	After error detected	Abend	Abend	Abend	Abend	12	Abend
GDS RECEIVE	CDBERR	After error detected	Abend	Abend	Abend	Abend	=	Abend
GDS RECEIVE	CDBCONF + CDBFREE	After confirm flow detected	Abend	Abend	Abend	Abend	8	Abend
GDS RECEIVE	CDBCONF + CDBRECV	After confirm flow detected	Abend	Abend	Abend	Abend	6	Abend
GDS RECEIVE	CDBCONF	After confirm flow detected	Abend	Abend	Abend	Abend	7	Abend
GDS RECEIVE	CDBFREE	After error detected	Abend	Abend	Abend	Abend	12	Abend
GDS RECEIVE	CDBRECV	When data available	Abend	Abend	Abend	Abend	=	Abend
GDS RECEIVE LLID	CDBCOMPL	When data available	Abend	Abend	Abend	Abend	=	Abend
GDS RECEIVE	×	When data available	Abend	Abend	Abend	Abend	2	Abend
GDS ISSUE CONFIRMATION	×	Immediately	Abend	Abend	Abend	Abend	Abend	5
GDS ISSUE ERROR	CDBFREE	After response from partner	Abend	12	12	Abend	12	12
GDS ISSUE ERROR	×	After response from partner	Abend	=	2	Abend	2	2
GDS ISSUE ABEND	×	Immediately	Abend	12	12	12	12	12

Table 38. States 1 - 6 (continued)

Command issued	CDB flag returned	Command returns	ALLO-CATED	SEND	PEND-RECEIVE	PEND-FREE	RECEIVE	CONF-RECEIVE
			State 1	**State 2**	**State 3**	**State 4**	**State 5**	**State 6**
GDS ISSUE SIGNAL	×	Immediately	Abend	=	=	Abend	=	=
GDS WAIT	×	Immediately	Abend	=	5	12	Abend	Abend
GDS FREE	×	Immediately	End	Abend	Abend	End	Abend	Abend

Table 39. States 7 - 13

Command issued	CDB flag returned	CONF- SEND	CONF-FREE	SYNC-RECEIVE	SYNC-SEND	SYNC-FREE	FREE	ROLL-BACK
		State 7	**State 8**	**State 9**	**State 10**	**State 11**	**State 12**	**State 13**
GDS CONNECT PROCESS	EIBERR + EIBFREE	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS CONNECT PROCESS	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS EXTRACT PROCESS	×	=	=	N/A	N/A	N/A	=	N/A
GDS EXTRACT ATTRIBUTES	×	=	=	N/A	N/A	N/A	=	N/A
GDS SEND (any valid form)	CDBERR + CDBFREE	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS SEND (any valid form)	CDBFREE	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS SEND INVITE WAIT	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS SEND INVITE CONFIRM	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS SEND INVITE	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS SEND LAST WAIT	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS SEND LAST CONFIRM	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS SEND LAST	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS SEND WAIT	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS SEND CONFIRM	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS SEND	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS RECEIVE	CDBERR + CDBFREE	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS RECEIVE	CDBERR	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS RECEIVE	CDBCONF + CDBFREE	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS RECEIVE	CDBCONF + CDBRECV	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS RECEIVE	CDBCONF	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS RECEIVE	CDBFREE	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS RECEIVE	CDBRECV	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS RECEIVE LLID	CDBCOMPL	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS RECEIVE	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS ISSUE CONFIRMATION	×	2	12	N/A	N/A	N/A	Abend	N/A
GDS ISSUE ERROR	CDBFREE	12	12	N/A	N/A	N/A	Abend	N/A
GDS ISSUE ERROR	×	2	2	N/A	N/A	N/A	Abend	N/A
GDS ISSUE ABEND	×	12	12	N/A	N/A	N/A	Abend	N/A
GDS ISSUE SIGNAL	×	=	=	N/A	N/A	N/A	Abend	N/A
GDS WAIT	×	Abend	Abend	N/A	N/A	N/A	Abend	N/A
GDS FREE	×	Abend	Abend	N/A	N/A	N/A	End	N/A

State tables for APPC mapped conversations at sync level 2

Tables showing the state transitions that occur when transactions engage in APPC mapped conversations at sync level 2, under the EXEC CICS API.

The GDS ISSUE SIGNAL command and the CDBSIG flag

In the tables, the CDBSIG flag is not mentioned. This is because its use is optional and is entirely a matter of agreement between the two conversation partners. In the worst case, it can occur at any time after every command that affects the CDB flags. However, used for the purpose for which it was intended, it usually occurs after a GDS SEND command. Its priority in the order of testing depends on the role you give it in the application.

The CDBSIG flag is set when the partner issues the **`GDS ISSUE SIGNAL`** command.

State changes for the SYNCPOINT and SYNCPOINT ROLLBACK commands

When the SYNCPOINT and SYNCPOINT ROLLBACK commands are issued, they are propagated on, and affect the state of, all the conversations that are currently active for the task, including MRO conversations.

Following rollback, the conversation can be in **SEND** or **RECEIVE** state, depending on the conversation state at the start of the current distributed unit of work. The conversation can be in **FREE** state if it ended abnormally due to session failure or due to deallocate abend being received, or if the partner transaction issued a SEND LAST WAIT or FREE command.

After a syncpoint or rollback, it is advisable to determine the conversation state before issuing any further commands against the conversation.

State changes following the ISSUE PREPARE command

Although ISSUE PREPARE can return with the conversation in either **SYNCSEND** state, **SYNCRECEIVE** state, or **SYNCFREE** state, the only commands allowed on that conversation following an ISSUE PREPARE are SYNCPOINT and SYNCPOINT ROLLBACK. All other commands abend.

State tables

Table 40. States 1 - 6

		Command returns	**ALLO-CATED**	**SEND**	**PEND-RECEIVE**	**PEND-FREE**	**RECEIVE**	**CONF-RECEIVE**
Command issued	**CDB flag returned**		**State 1**	**State 2**	**State 3**	**State 4**	**State 5**	**State 6**
GDS CONNECT PROCESS	EIBERR + EIBFREE	Immediately	12	Abend	Abend	Abend	Abend	Abend
GDS CONNECT PROCESS	×	Immediately	2	Abend	Abend	Abend	Abend	Abend
GDS EXTRACT PROCESS (back-end transaction only)	×	Immediately	=	=	=	=	=	=
GDS EXTRACT ATTRIBUTES	×	Immediately	=	=	=	=	=	=
GDS SEND (any valid form)	CDBERR + CDBFREE	After error flow detected	Abend	12	Abend	12	Abend	Abend
GDS SEND (any valid form)	CDBERR	After error flow detected	Abend	5	Abend	12	Abend	Abend
GDS SEND INVITE WAIT	×	After data flows	Abend	5	Abend	Abend	Abend	Abend
GDS SEND INVITE CONFIRM	×	After response from partner	Abend	5	Abend	Abend	Abend	Abend
GDS SEND INVITE	×	After data buffered	Abend	3	Abend	Abend	Abend	Abend
GDS SEND LAST WAIT	×	After data flows	Abend	12	Abend	Abend	Abend	Abend
GDS SEND LAST CONFIRM	×	After response from partner	Abend	12	Abend	Abend	Abend	Abend

Table 40. States 1 - 6 (continued)

Command issued	CDB flag returned	Command returns	ALLO-CATED	SEND	PEND-RECEIVE	PEND-FREE	RECEIVE	CONF-RECEIVE
			State 1	**State 2**	**State 3**	**State 4**	**State 5**	**State 6**
GDS SEND LAST	×	After data buffered	Abend	4	Abend	Abend	Abend	Abend
GDS SEND WAIT	×	After data flows	Abend	=	Abend	Abend	Abend	Abend
GDS SEND CONFIRM	×	After response from partner	Abend	=	5	12	Abend	Abend
GDS SEND	×	After data buffered	Abend	=	Abend	Abend	Abend	Abend
GDS RECEIVE	CDBERR + CDBSYNRB	After rollback flow detected	Abend	Abend	Abend	Abend	13	Abend
GDS RECEIVE	CDBERR + CDBFREE	After error detected	Abend	Abend	Abend	Abend	12	Abend
GDS RECEIVE	CDBERR	After error detected	Abend	Abend	Abend	Abend	=	Abend
GDS RECEIVE	CDBSYNC + CDBFREE	After sync flow detected	Abend	Abend	Abend	Abend	11	Abend
GDS RECEIVE	CDBSYNC + CDBRECV	After sync flow detected	Abend	Abend	Abend	Abend	9	Abend
GDS RECEIVE	CDBSYNC	After sync flow detected	Abend	Abend	Abend	Abend	10	Abend
GDS RECEIVE	CDBCONF + CDBFREE	After confirm flow detected	Abend	Abend	Abend	Abend	8	Abend
GDS RECEIVE	CDBCONF + CDBRECV	After confirm flow detected	Abend	Abend	Abend	Abend	6	Abend
GDS RECEIVE	CDBCONF	After confirm flow detected	Abend	Abend	Abend	Abend	7	Abend
GDS RECEIVE	CDBFREE	After error flow detected	Abend	Abend	Abend	Abend	12	Abend
GDS RECEIVE	CDBRECV	When data available	Abend	Abend	Abend	Abend	=	Abend
GDS RECEIVE LLID	CDBCOMPL	When data available	Abend	Abend	Abend	Abend	=	Abend
GDS RECEIVE	×	When data available	Abend	Abend	Abend	Abend	2	Abend
GDS ISSUE CONFIRMATION	×	Immediately	Abend	Abend	Abend	Abend	Abend	5
GDS ISSUE ERROR	CDBFREE	After response from partner	Abend	12	12	Abend	12	12
GDS ISSUE ERROR	×	After response from partner	Abend	=	2	Abend	2	2
GDS ISSUE ABEND	×	Immediately	Abend	12	12	12	12	12
GDS ISSUE SIGNAL	×	Immediately	Abend	=	=	Abend	=	=
GDS ISSUE PREPARE	CDBERR + CDBSYNRB	After response from partner	Abend	13	13	13	Abend	Abend
GDS ISSUE PREPARE	CDBERR + CDBFREE	After error detected	Abend	12	12	12	Abend	Abend
GDS ISSUE PREPARE	CDBERR	After error detected	Abend	5	5	5	Abend	Abend
GDS ISSUE PREPARE	×	After response from partner	Abend	10	9	11	Abend	Abend
SYNCPOINT	EIBRLDBK	After response from partner	=	2 or 5	2 or 5	2 or 5	Abend	Abend
SYNCPOINT	×	After response from partner	=	=	5	12	Abend	Abend

Table 40. States 1 - 6 (continued)

Command issued	CDB flag returned	Command returns	ALLO-CATED	SEND	PEND-RECEIVE	PEND-FREE	RECEIVE	CONF-RECEIVE
			State 1	State 2	State 3	State 4	State 5	State 6
SYNCPOINT ROLLBACK	×	After rollback across UOW	=	2 or 5	2 or 5	2 or 5	2 or 5	2 or 5
GDS WAIT	×	Immediately	Abend	=	5	12	Abend	Abend
GDS FREE	×	Immediately	End	Abend	Abend	End	Abend	Abend

Table 41. States 7 -13

Command issued	CDB flag returned	CONF- SEND	CONF-FREE	SYNC-RECEIVE	SYNC-SEND	SYNC-FREE	FREE	ROLL-BACK
		State 7	State 8	State 9	State 10	State 11	State 12	State 13
GDS CONNECT PROCESS	EIBERR + EIBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS CONNECT PROCESS	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS EXTRACT PROCESS (back-end transaction only)	×	=	=	=	=	=	=	=
GDS EXTRACT ATTRIBUTES	×	=	=	=	=	=	=	=
GDS SEND (any valid form)	CDBERR + CDBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS SEND (any valid form)	CDBERR	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS SEND INVITE WAIT	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS SEND INVITE CONFIRM	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS SEND INVITE	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS SEND LAST WAIT	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS SEND LAST CONFIRM	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS SEND LAST	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS SEND WAIT	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS SEND CONFIRM	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS SEND	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS RECEIVE	CDBERR + CDBSYNRB	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS RECEIVE	CDBERR + CDBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS RECEIVE	CDBERR	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS RECEIVE	CDBSYNC + CDBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS RECEIVE	CDBSYNC + CDBRECV	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS RECEIVE	CDBSYNC	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS RECEIVE	CDBCONF + CDBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS RECEIVE	CDBCONF + CDBRECV	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS RECEIVE	CDBCONF	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS RECEIVE	CDBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS RECEIVE	CDBRECV	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS RECEIVE LLID	CDBCOMPL	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS RECEIVE	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS ISSUE CONFIRMATION	×	2	12	Abend	Abend	Abend	Abend	Abend
GDS ISSUE ERROR	CDBFREE	12	12	12	12	12	Abend	Abend
GDS ISSUE ERROR	×	2	2	2	2	2	Abend	Abend

Table 41. States 7 -13 (continued)

Command issued	**CDB flag returned**	**CONF- SEND**	**CONF- FREE**	**SYNC- RECEIVE**	**SYNC- SEND**	**SYNC- FREE**	**FREE**	**ROLL- BACK**
		State 7	**State 8**	**State 9**	**State 10**	**State 11**	**State 12**	**State 13**
GDS ISSUE ABEND	×	12	12	12	12	12	Abend	Abend
GDS ISSUE SIGNAL	×	=	=	=	=	=	Abend	Abend
GDS ISSUE PREPARE	CDBERR + CDBSYNRB	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS ISSUE PREPARE	CDBERR + CDBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS ISSUE PREPARE	CDBERR	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS ISSUE PREPARE	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SYNCPOINT	EIBRLDBK	Abend	Abend	2 or 5	2 or 5	2 or 5	=	Abend
SYNCPOINT	×	Abend	Abend	5	2	12	=	Abend
SYNCPOINT ROLLBACK	×	2 or 5	2 or 5	2 or 5	2 or 5	2 or 5	Abend	2 or 5
GDS WAIT	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
GDS FREE	×	Abend	Abend	Abend	Abend	Abend	End	Abend

Chapter 5. Writing programs for LUTYPE6.1 conversations

These topics describe the CICS APIs available for DTP programming using LUTYPE6.1 conversations.

Conversation initiation

The front-end transaction is responsible for acquiring a session, specifying the conversation characteristics, and requesting the startup of the back-end transaction in the partner system.

Allocating a session to the conversation

Initially, there is no conversation, and therefore no conversation state.

The front-end transaction acquires a session to start a new conversation by issuing an ALLOCATE command.

The RESP value should be checked to ensure that a session has been allocated. If successful, the RESP value is DFHRESP(NORMAL), the conversation is in **allocated state** (state 1) and the session identifier (**convid**) from EIBRSRCE must be saved immediately. The convid must be used in subsequent commands for this conversation.

If the front-end transaction is started by ATI in the local region, and is required to hold a conversation with an LUTYPE6.1 session as its principal facility, the session has already been allocated when the transaction starts. You can omit the SESSION option from commands relating to the principal facility. If, however, you want to name the session explicitly in these commands, you should obtain its name from EIBTRMID.

Connecting the partner transaction

When a session has been acquired, the next step is to cause the partner transaction to be initiated.

The state table shows that, in **allocated state** (state 1), one of the commands available is SEND. Using this command, the back-end transaction identifiers can be specified in the first four bytes of the data which, when transferred to the partner system, will attach the required back-end transaction. The send buffer containing the transaction name together with any other data, will be flushed immediately and the front-end transaction will wait until a response is received from the back-end transaction.

Alternatively, when a session has been acquired, the front-end transaction can build and send an attach header with the first transmission of data. The attach header can be built using the BUILD ATTACH command.

When using the BUILD ATTACH command, you must give a name to the built attach header which can then be used in the ATTACHID option of the first SEND (or converse) command. The back-end transaction name should also be specified.

Back-end transaction initiation

The back-end transaction is initiated either by an attach header received from the partner system or by a transaction name included in the incoming data, and is started with the session as its principal facility.

Initially, the back-end transaction should determine the convid from EIBTRMID. This is not strictly necessary because the session is the back-end transaction's principal facility making the CONVID parameter optional for DTP commands on this conversation. However, the convid is very useful for audit trails. Also, if the back-end transaction is involved in more than one conversation, then always specifying the convid improves program readability and problem determination.

A CICS transaction can be the back-end transaction in CICS-to- IMS communication only in the special case of SEND/RECEIVE asynchronous processing. The transaction is initiated by an LUTYPE6.1 attach FMH received from the remote IMS system, and is allowed to issue a single RECEIVE command only, possibly followed by an EXTRACT ATTACH command.

It is possible that the back-end transaction might fail to start. This will result in the front-end transaction abending.

Transferring data on the conversation

These topics discuss how to pass data between the front-end and back-end transactions.

Sending data to the partner transaction

The SEND command is used to send data to the connected partner.

This command is valid in allocated state (state 1) or **send state** (state 2). Because a successful simple SEND completes in **send state** (state 2), it is possible to issue a number of successive sends.

Switching from sending to receiving data

There is more than one way of switching from **send state** to **receive state** .

One possibility is to use a SEND INVITE command. The state table shows that after SEND INVITE the conversation switches to **pendreceive state** (state 3). As the column for state 3 shows, a WAIT TERMINAL command switches the conversation to **receive state** (state 5).

Another possibility is to specify INVITE and WAIT on the SEND command. As the state table shows, SEND INVITE WAIT switches the conversation to **receive state** (state 5).

Receiving data from the partner transaction

The RECEIVE command is used to receive data from the connected partner.

T he rows in the state tables for the RECEIVE command show the EIB fields that should be tested after issuing a RECEIVE command. As well as showing which field should be tested, the state tables also shows the order in which the tests should be made. Note that you should always test for RESP values.

The transaction whose side of the conversation is in **receive state** cannot change to **send state** , but can request a change of direction by using the ISSUE SIGNAL command. This causes the SIGNAL condition to be raised in the partner transaction the next time it issues a SEND, RECEIVE, or CONVERSE command. The application is responsible for determining the purpose of the SIGNAL condition and responding appropriately.

Waiting for a signal

A transaction can wait for its partner to send a signal. This is done by issuing the WAIT SIGNAL command and testing for the SIGNAL condition.

T he WAIT SIGNAL command suspends the transaction until its partner responds with an ISSUE SIGNAL command. This response activates the suspended transaction and raises the SIGNAL condition.

Combining sending and receiving

The CONVERSE command combines the functions SEND INVITE and RECEIVE.

T his command is useful when one transaction needs a response from the partner transaction to continue processing.

Communicating errors across a conversation

If a transaction is receiving data on a conversation and needs to notify its partner of an error, it can use the ISSUE SIGNAL command to request that the partner does a SEND INVITE.

When the ISSUE SIGNAL request is received, EIBSIG is set to X'FF' and the SIGNAL condition is raised. Note that when a *signal* is received, the transaction is not obliged to issue SEND INVITE.

Safeguarding data integrity

If it is important to safeguard data integrity across connected transactions, then synchronization commands are available.

The c ommands are:

SYNCPOINT
SRRCMIT (SAA verb for SYNCPOINT)

The use of these commands in DTP is described in "Syncpointing a distributed process" on page 97.

Ending the conversation

These topics describe the different ways a conversation can end, either unexpectedly or under transaction control.

W hen under transaction control, one transaction will issue a request for termination and the other will receive this request. Once this has happened the conversation is unusable and **both** transactions must issue a FREE command to release the session.

Ending a conversation normally

The **`SEND LAST`** command is used to terminate a conversation.

It should be used in conjunction with either the WAIT option or the SYNCPOINT command, and followed by the FREE command. However, SEND LAST WAIT will cause the conversation to end before the subsequent syncpoint can be propagated to the partner transaction. This may mean that the protected resources in one system could be committed while those in the other system could be backed out.

From the state table it can be seen that it is possible to end a conversation by issuing the FREE command provided the conversation is in **send state** (state 2). This will generate an implicit SEND LAST WAIT command before the FREE is executed and is therefore not recommended.

Note: A distributed transaction should not end a conversation by issuing an **`EXEC CICS RETURN`** command, but instead follow the sequence of commands as described. The issue of an **`EXEC CICS RETURN`** could lead to one or both transactions ending abnormally.

Unexpected termination of a conversation

From time to time, partner systems do fail and sessions go out of service.

I f this happens in the middle of a DTP conversation, the transaction will be terminated abnormally.

Checking the outcome of a DTP command

Checking the response from a DTP command can be separated into two stages.

The stages are:

1. Testing for request failure
2. Testing for indicators received on the conversation.

Testing for request failure is the same as for other EXEC CICS commands in that conditions are raised and may be handled using HANDLE CONDITION or RESP. EIBRCODE will also contain an error code.

If the request has not failed, it is then possible to test for indicators received on the conversation. These are returned to the application in the EIB. The following EIB fields are relevant to all DTP commands. (See EIB fields for programming information on the contents and format of EIB fields.)

EIBFREE
when set to X'FF' indicates that the partner transaction has ended the conversation. It should be tested in conjunction with EIBSYNC to determine exactly how to end the conversation.

EIBSYNC
when set to X'FF' indicates the partner transaction/system has requested a syncpoint.

Table 42 on page 90 shows how these EIB fields interact.

Table 42. Interaction of some EIB fields

EIB- FREE	**EIB- SYNC**	**Description**
X'FF'	X'00'	The partner transaction or system has sent SEND LAST followed by a FREE command.
X'FF'	X'FF'	The partner transaction or system has issued SEND LAST followed by SYNCPOINT. The local program should reply with a SYNCPOINT command followed by a FREE command.
X'00'	X'FF'	The partner transactions or system has issued a SYNCPOINT.

In addition, there is a group of EIB fields that are relevant only to the RECEIVE and CONVERSE commands. These are:

EIBCOMPL
when set to X'FF' indicates that all the data sent at one time has been received. This field is used in conjunction with the RECEIVE NOTRUNCATE command.

EIBRECV
when set to X'FF' indicates the partner transaction did not use the INVITE option on its last SEND command.

EIBATT
when set to X'FF' indicates that the data received contained an attach header. The attach header is not passed to the application; however, EIBATT indicates that an EXTRACT ATTACH command is appropriate.

EIBFMH
when set to X'FF' indicates that the data passed to the application contains a concentrated FMH. This happens only when the partner CICS transaction builds an FMH in the data and the FMH option on the SEND command is specified.

Note: Profiles specifying INBFMH (ALL) must be used in the ALLOCATE commands if FMHs are to be sent and received and EIBATT or EIBFMH to be sent appropriately. The default profile DFHCICSA used for the session allocated by the front-end transaction, has INBFMH (ALL) specified. However, the default principal facility profile DFHCICST used for the back-end transaction does not have INBFMH (ALL) specified.

Considerations for the front-end transaction

Several special considerations apply to the front-end transaction in an LUTYPE6.1 conversation.

Except in the special case of the receiving transaction in SEND/RECEIVE asynchronous processing, the CICS transaction is always the front-end transaction in CICS-to- IMS DTP.

The front-end transaction is responsible for acquiring a session to the remote IMS system and initiating the partner transaction.

Thereafter, the two transactions become equals. However, the front-end transaction is usually designed as the client, or driving, transaction.

Session allocation

You acquire an LUTYPE6.1 session to a remote IMS system by means of the ALLOCATE command.

The command has the following format:

```
 ALLOCATE {SYSID(name)|SESSION(name)}
[PROFILE(name)]
[NOQUEUE]
```

You can use the SESSION option to request the use of a specific session to the remote IMS system, or you can use the SYSID option to name the partner system and allow CICS to select an available session. The use of the SESSION option is not normally recommended, because it can result in an application program queuing on a specific session when others are available. In most cases, therefore, you use the SYSID option to name the system with which the session is required.

If CICS cannot find the named system, or all sessions to that system are out of service, it raises the SYSIDERR condition. If CICS cannot find the named session, or that session is out of service, it raises the SESSIONERR condition.

The PROFILE option allows you to select a specified communication profile for an LUTYPE6.1 session. The profile, which is set up during resource definition, contains a set of terminal control processing options that are to be used for the session.

If you omit the PROFILE option, CICS uses the default profile DFHCICSA. This profile specifies INBFMH(ALL), which means that incoming function management headers are passed to your program and cause the INBFMH condition to be raised.

The NOQUEUE option allows you to specify explicitly that you do not want your request for a session to be queued if a session is not available immediately. A session is "not immediately available" in any of the following situations:

- All the sessions to the specified system are in use.
- The only available sessions are not bound (in which case CICS would have to bind a session).
- The only available sessions are contention losers (in which case CICS would have to bid to begin a bracket).

The action taken by CICS if a session is not immediately available depends on whether you specify NOQUEUE and also on whether your application has executed a HANDLE command for the SYSBUSY condition. These are the possible combinations:

- HANDLE for SYSBUSY condition
  - Control is returned immediately to the label specified in the HANDLE command, whether or not you have specified NOQUEUE.
- No HANDLE for SYSBUSY condition
  - If you have specified NOQUEUE, control is returned immediately to your application program. A RESP value of DFHRESP(SYSBUSY) is returned. You should test this field immediately after issuing the ALLOCATE command.
  - If you have omitted the NOQUEUE option, CICS queues the request until a session is available.

Whether a delay in acquiring a session is acceptable is dependent on your application.

Similar considerations apply to an ALLOCATE command that specifies SESSION rather than SYSID. The associated condition is SESSBUSY.

The session identifier

When a session has been allocated, the name by which it is known is available in the EIBRSRCE field in the EIB.

Because EIBRSRCE will probably be overwritten by the next EXEC CICS command, your application must capture the session name immediately. It is the name that you must use in the SESSION option of all subsequent commands that relate to this session.

Summary of commands for LUTYPE6.1 conversations

The CICS application programming interface provides a set of commands for use in LUTYPE6.1 conversations.

Table 43 on page 92 shows the commands used in LUTYPE6.1 conversations.

Table 43. Summary of commands used in LUTYPE6.1 conversations

Use to ...	Command	More information
Acquire a session.	ALLOCATE	"Allocating a session to the conversation" on page 87
Build an attach header.	BUILD ATTACH	"Connecting the partner transaction" on page 87
Access session-related information.	EXTRACT ATTACH	"Back-end transaction initiation" on page 87
Send data and control information to the conversation partner.	SEND	"Sending data to the partner transaction" on page 88
Receive data from the conversation partner.	RECEIVE	"Receiving data from the partner transaction" on page 88
Send and receive data on the conversation.	CONVERSE	"Combining sending and receiving" on page 88
Inform all partners of readiness to commit recoverable resources.	SYNCPOINT	"Syncpointing a distributed process" on page 97
Signal an unusual condition to the conversation partner, usually against the flow of data.	ISSUE SIGNAL	"Communicating errors across a conversation" on page 89
Suspend processing until the SIGNAL condition is raised.	WAIT SIGNAL	"Waiting for a signal" on page 88
Ensure that CICS has transmitted any accumulated data or data flow control indicators before further processing.	WAIT TERMINAL	"Switching from sending to receiving data" on page 88
Free the session.	FREE	"Ending a conversation normally" on page 89

State transitions in LUTYPE6.1 conversations

These topics show the state transitions that occur when transactions engage in LUTYPE6.1 conversations. The state transitions are presented in the form of a state table. The state table shows which commands a transaction can issue while the conversation is in any given state. It also shows how the conversation state changes as a result of any command.

How to use the state table

The state tables show the commands you can issue, the EIB flags that can be set when the command is issued, and the conversation states.

The commands you can issue, coupled with the EIB flags that can be set after execution, are shown on the first column of the table. The possible conversation states are shown across the top of the table. The states correspond to the columns of the table. The intersection of row (command and EIB flag) and column (state) represents the state transition, if any, that occurs when that command returning a particular EIB flag is issued in that state.

A number at an intersection indicates the state number of the next state. Other symbols represent other conditions, as follows:

Symbol	Meaning
N/A	Cannot occur.
×	The EIB flag is any one that has not been covered in earlier rows, or it is irrelevant.
Abend	The command is not valid in this state. Issuing a command in a state in which it is not valid usually causes an ATCV abend.
=	Remains in current state.
End	End of conversation.

Initial states

A front-end transaction can be initiated either from a transaction or by automatic transaction initiation (ATI).

A terminal-initiated front-end transaction must issue an ALLOCATE command to acquire a session. If the session is successfully allocated, the front-end transaction's side of the conversation goes into **allocated state** (state 1).

A front-end transaction started by ATI in the local region, with an LUTYPE6.1 session as its principal facility, already has a session allocated. Such a transaction does not issue an ALLOCATE command, and its side of the conversation starts in **send state** (state 2).

A back-end transaction is initially in **receive state** (state 5).

Testing the conversation state

An application cannot check the state of an LUTYPE6.1 conversation directly.

T he application must instead check RESP and the EIB fields after each command, and must follow the rules shown in the state table.

State tables for LUTYPE6.1 conversations

Tables showing the state transitions that occur when transactions engage in LUTYPE6.1 conversations, under the EXEC CICS API.

The `ISSUE SIGNAL` command and the EIBSIG flag

In the tables, the EIBSIG flag is not mentioned. This is because its use is optional and is entirely a matter of agreement between the two conversation partners. In the worst case, it can occur at any time after every command that affects the EIB flags. However, used for the purpose for which it was intended, it usually occurs after a SEND command. Its priority in the order of testing depends on the role you give it in the application.

The EIBSIG flag is set when the partner issues the **`ISSUE SIGNAL`** command.

The `RECEIVE NOTRUNCATE` command

The **`RECEIVE NOTRUNCATE`** command returns a zero value in EIBCOMPL to indicate that the user buffer was too small to contain all the data received from the partner transaction. Normally, you would continue to issue **`RECEIVE NOTRUNCATE`** commands until the last section of data is passed to you, which is indicated by EIBCOMPL = X'FF' . If NOTRUNCATE is not specified, and the data area specified by the RECEIVE command is too small to contain all the data received, CICS truncates the data and sets the LENGERR condition.

State changes for the SYNCPOINT and SYNCPOINT ROLLBACK commands

When the SYNCPOINT and SYNCPOINT ROLLBACK commands are issued, they are propagated on, and affect the state of, all the conversations that are currently active for the task, including APPC and MRO conversations.

State tables

Table 44. States 1 - 6

		Command returns	ALLO-CATED	SEND	PEND-RECEIVE	PEND-FREE	RECEIVE	CONF-RECEIVE
Command issued	**EIB flag returned**		**State 1**	**State 2**	**State 3**	**State 4**	**State 5**	**State 6**
BUILD ATTACH	×	Immediately	=	=	=	=	=	N/A
EXTRACT ATTACH	×	Immediately	=	=	=	=	=	N/A
SEND INVITE WAIT	×	After data and CD flows	5	5	Abend	Abend	Abend	N/A
SEND INVITE	×	After data buffered	3	3	Abend	Abend	Abend	N/A
SEND LAST WAIT	×	After data and EB flows	12	12	Abend	Abend	Abend	N/A
SEND LAST	×	After data buffered	4	4	Abend	Abend	Abend	N/A
SEND	×	After data buffered	=	=	Abend	Abend	Abend	N/A
RECEIVE	EIBSYNC + EIBFREE	After sync flow detected	Abend	11	11	Abend	11	N/A
RECEIVE	EIBSYNC + EIBRECV	After sync flow detected	Abend	9	9	Abend	9	N/A
RECEIVE	EIBSYNC	After sync flow detected	Abend	10	10	Abend	10	N/A
RECEIVE	EIBFREE	After EB detected	Abend	12	12	Abend	12	N/A
RECEIVE	EIBRECV	When data available	Abend	5	5	Abend	=	N/A
RECEIVE NOTRUNCATE	EIBCOMPL	When data available	Abend	5	5	Abend	=	N/A
RECEIVE	×	When data available	Abend	2	2	Abend	2	N/A
CONVERSE	As for RECEIVE but allowed in send state		As for RECEIVE but allowed in send state	As for RECEIVE but allowed in send state	As for RECEIVE but allowed in send state	As for RECEIVE but allowed in send state	As for RECEIVE but allowed in send state	As for RECEIVE but allowed in send state
ISSUE SIGNAL	×	Immediately	Abend	=	=	=	=	N/A
WAIT SIGNAL	×	After response from partner	Abend	=	=	=	=	N/A
SYNCPOINT	×	After response from partner	=	=	5	12	Abend	N/A
WAIT TERMINAL	×	Immediately	=	=	5	12	=	N/A
FREE	×	Immediately	End	End	Abend	End	Abend	N/A

Table 45. States 7 -13

Command issued	EIB flag returned	CONF- SEND	CONF-FREE	SYNC-RECEIVE	SYNC-SEND	SYNC-FREE	FREE	ROLL-BACK
		State 7	**State 8**	**State 9**	**State 10**	**State 11**	**State 12**	**State 13**
BUILD ATTACH	×	N/A	N/A	N/A	=	=	=	=
EXTRACT ATTACH	×	N/A	N/A	=	=	=	=	N/A
SEND INVITE WAIT	×	N/A	N/A	Abend	Abend	Abend	Abend	N/A

Table 45. States 7 -13 (continued)

Command issued	**EIB flag returned**	**CONF- SEND**	**CONF- FREE**	**SYNC- RECEIVE**	**SYNC- SEND**	**SYNC- FREE**	**FREE**	**ROLL- BACK**
		State 7	**State 8**	**State 9**	**State 10**	**State 11**	**State 12**	**State 13**
SEND INVITE	×	N/A	N/A	Abend	Abend	Abend	Abend	N/A
SEND LAST WAIT	×	N/A	N/A	Abend	Abend	Abend	Abend	N/A
SEND LAST	×	N/A	N/A	Abend	Abend	Abend	Abend	N/A
SEND	×	N/A	N/A	Abend	Abend	Abend	Abend	N/A
RECEIVE	EIBSYNC + EIBFREE	N/A	N/A	Abend	Abend	Abend	Abend	N/A
RECEIVE	EIBSYNC + EIBRECV	N/A	N/A	Abend	Abend	Abend	Abend	N/A
RECEIVE	EIBSYNC	N/A	N/A	Abend	Abend	Abend	Abend	N/A
RECEIVE	EIBFREE	N/A	N/A	Abend	Abend	Abend	Abend	N/A
RECEIVE	EIBRECV	N/A	N/A	Abend	Abend	Abend	Abend	N/A
RECEIVE NOTRUNCATE	EIBCOMPL	N/A	N/A	Abend	Abend	Abend	Abend	N/A
RECEIVE	×	N/A	N/A	Abend	Abend	Abend	Abend	N/A
CONVERSE		As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE
ISSUE SIGNAL	×	N/A	N/A	=	=	=	Abend	N/A
WAIT SIGNAL	×	N/A	N/A	=	=	=	Abend	N/A
SYNCPOINT	×	N/A	N/A	5	2	12	=	N/A
WAIT TERMINAL	×	N/A	N/A	Abend	Abend	Abend	Abend	N/A
FREE	×	N/A	N/A	Abend	Abend	Abend	End	N/A

Chapter 6. Syncpointing a distributed process

You can use the **SYNCPOINT** and **SYNCPOINT ROLLBACK** commands to commit and roll back a distributed process.

This topic concentrates on the programming aspects of using the **SYNCPOINT** and **SYNCPOINT ROLLBACK** commands across APPC conversations at sync level 2 and MRO conversations.

Syncpointing a distributed process

You can use the **SYNCPOINT** and **SYNCPOINT ROLLBACK** commands to commit and roll back a distributed process.

This topic concentrates on the programming aspects of using the **SYNCPOINT** and **SYNCPOINT ROLLBACK** commands across APPC conversations at sync level 2 and MRO conversations. This includes issuing syncpoint requests and receiving them, because they are transmitted to all partners connected on conversations at sync level 2. The section also describes how these partners are given the opportunity to back out even though they have been requested to commit.

The SAA equivalent commands (SRRCMIT and SRRBACK) are described in Systems Application Architecture Common Programming Interface Resource Recovery Reference.

The SYNCPOINT command

The SYNCPOINT command is used to commit recoverable resources. In a DTP environment, the effect of the SYNCPOINT command is propagated across all conversations using sync level 2 or MRO.

N o matter how many DTP transactions are connected by conversations at sync level 2, the distributed process should be designed such that only one of the transactions initiates syncpoint activity for the distributed unit of work. When issuing the SYNCPOINT command, this transaction, known as the **syncpoint initiator** must be in **send state** (state 2), **pendreceive state** (state 3), or **pendfree state** (state 4) on all its conversations at sync level 2. Any transaction that receives the syncpoint request becomes a **syncpoint agent**.

A syncpoint agent is in **receive state** on its conversation with the syncpoint initiator and becomes aware of the syncpoint request by testing EIBSYNC (CDBSYNC in the APPC basic interface) after issuing a RECEIVE command. If it decides to respond positively by issuing SYNCPOINT, it must be in an appropriate state on all the conversations with its own agents, for which it has become syncpoint initiator. If an agent transaction responds negatively to a syncpoint request by issuing SYNCPOINT ROLLBACK, the initiator sees EIBRLDBK set (X'FF'), which must be tested on return from the SYNCPOINT command. (This is also true for APPC basic conversations.)

Your transaction design should ensure that all participating transactions are in the correct conversation state before a SYNCPOINT command is issued.

When a syncpoint agent receives the syncpoint request, it is given the opportunity to respond positively (to commit recoverable resources) with a SYNCPOINT command or negatively (to back out recoverable resources) with a SYNCPOINT ROLLBACK command. For information on backing out recoverable resources, see “The SYNCPOINT ROLLBACK command” on page 98.

Examples of these commands are given in “Synchronizing two CICS systems” on page 99 and “Synchronizing three or more CICS systems” on page 106.

The ISSUE PREPARE command

The ISSUE PREPARE (GDS ISSUE PREPARE for the APPC basic interface) command is used to send the initial syncpoint flow to a selected partner on an APPC conversation at sync level 2. Depending on the partner's response, this command can then be followed by a SYNCPOINT or SYNCPOINT ROLLBACK command.

The reasons for using ISSUE PREPARE are as follows:

1. In complex DTP involving several conversing transactions, an ISSUE ERROR command from one of the transactions may not reach the syncpoint initiator in time to prevent it from issuing a SYNCPOINT command. This can lead to complex backout procedures for the distributed unit of work.

   Use ISSUE PREPARE as a way of flushing any error responses from the network.
2. If one or more syncpoint agents are not completely "reliable", use ISSUE PREPARE to check the status of these agents before proceeding with a general distributed syncpoint.

   Receiving ISSUE PREPARE is exactly the same as receiving SYNCPOINT. The partner program cannot detect any difference.

The SYNCPOINT ROLLBACK command

The SYNCPOINT ROLLBACK command is used to back out changes to recoverable resources. In a DTP environment, the effect of the SYNCPOINT command is propagated across all conversations using MRO or sync level 2.

A SYNCPOINT ROLLBACK command can be issued in any conversation state. If the command is issued when a conversation is in **receive state** (state 5), incoming data on that conversation is purged as described for the ISSUE ERROR and ISSUE ABEND commands.

When a transaction receives a SYNCPOINT ROLLBACK in response to a syncpoint request, the EIBRLDBK indicator is set. If SYNCPOINT ROLLBACK is received in response to any other request, the EIBERR and EIBSYNRB indicators (CDBERR and CDBSYNRB in the basic interface) are set.

The conversation state of each partner is restored to the state at the beginning of the distributed unit of work after a SYNCPOINT ROLLBACK command.

If a session failure or notification of a deallocate abend occurs during SYNCPOINT ROLLBACK processing, the command still completes successfully. If the same thing happens during SYNCPOINT processing, the command might complete successfully with EIBRLDBK set. In such circumstances, the conversation on which the failure or abend occurred will be in **free state** (state 12).

To avoid potential state problems, you can check the conversation state by using the STATE option on the command following SYNCPOINT ROLLBACK. However, to avoid the possibility of an abend, you are recommended to follow each SYNCPOINT ROLLBACK command with an EXTRACT ATTRIBUTES STATE command instead.

When a backout is required

In some situations, a transaction must back out in response to a request received from a partner.

A backout is required in the following circumstances:

- When SYNCPOINT ROLLBACK is received
- After ISSUE ABEND is sent
- After EIBERR and EIBFREE (CDBERR and CDBFREE in the basic interface) are returned together.

The conversation state does not always reflect the requirement to back out. However, CICS is aware of this requirement and converts the next SYNCPOINT request to a SYNCPOINT ROLLBACK request. If no SYNCPOINT or SYNCPOINT ROLLBACK request is issued before the end of the task, the task is abended (ASPN), and all recoverable resources are backed out.

Synchronizing two CICS systems

This section gives examples of how to commit and back out changes to recoverable resources made by two DTP transactions connected on a conversation using MRO or sync level 2.

SYNCPOINT in response to SYNCPOINT

In an APPC mapped conversation , a transaction issues a **SYNCPOINT** command and its partner responds with a **SYNCPOINT** command.

Figure 27 on page 99 , Figure 28 on page 99 , and Figure 29 on page 100 illustrate the effect of SEND, SEND INVITE, or SEND LAST preceding SYNCPOINT on an APPC mapped conversation. The figures also show the conversation state before each command and the state and EIB fields set after each command.

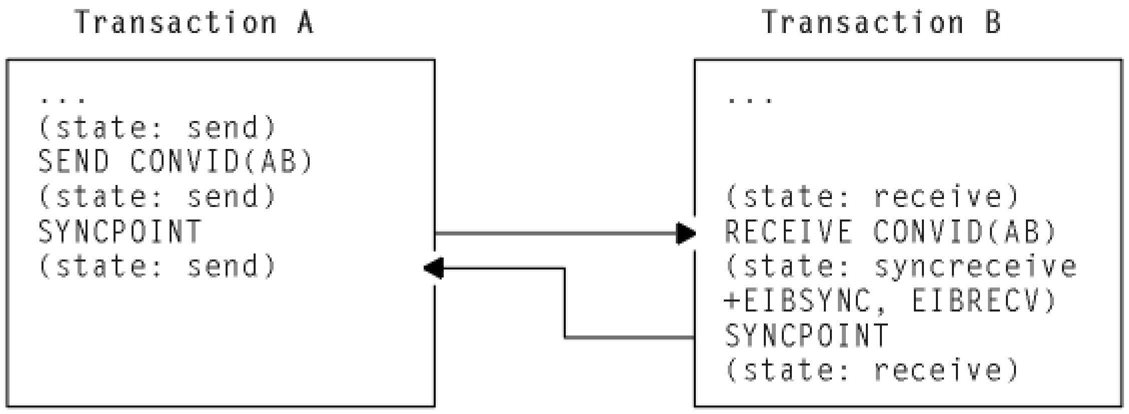

Figure 27. SYNCPOINT (in response to SEND followed by SYNCPOINT) on an APPC mapped conversation

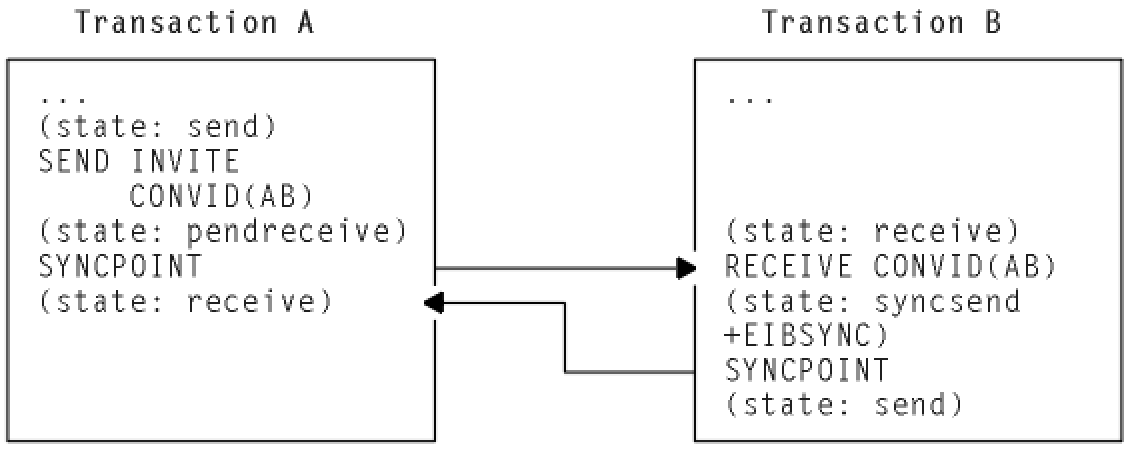

Figure 28. SYNCPOINT (in response to SEND INVITE followed by SYNCPOINT) on an APPC mapped conversation

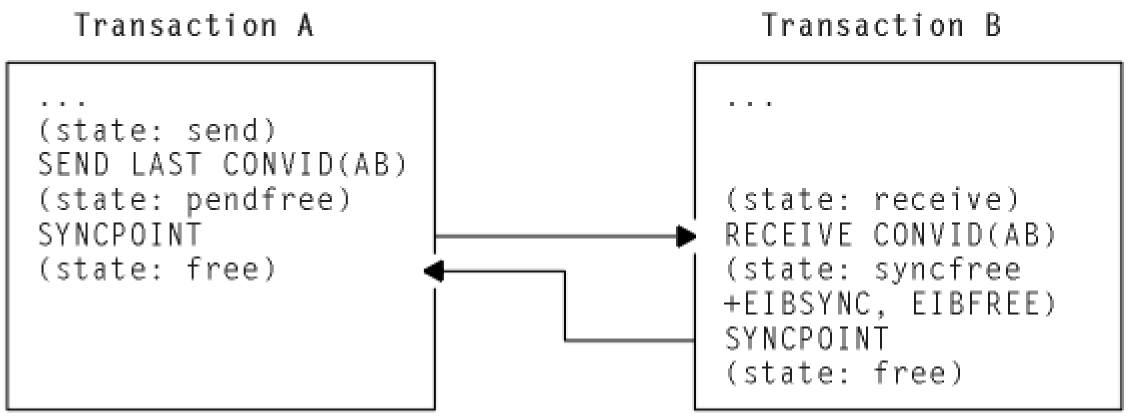

Figure 29. SYNCPOINT (in response to SEND LAST followed by SYNCPOINT) on an APPC mapped conversation

SYNCPOINT in response to ISSUE PREPARE
In an APPC mapped conversation , a transaction issues an **ISSUE PREPARE** command and its partner responds with a **SYNCPOINT** command.

Figure 30 on page 100 illustrates a SYNCPOINT command being used in response to ISSUE PREPARE on an APPC mapped conversation. The figure also shows the conversation state before each command and the state and EIB fields set after each command.

Note that it is also possible to use an ISSUE PREPARE command in **pendreceive state** (state 3) and **pendfree state** (state 4).

Note also that, although the ISSUE PREPARE command in Figure 30 on page 100 returns with the conversation in **syncsend state** (state 10), the only commands available for use on that conversation are SYNCPOINT and SYNCPOINT ROLLBACK. All other commands abend ATCV.

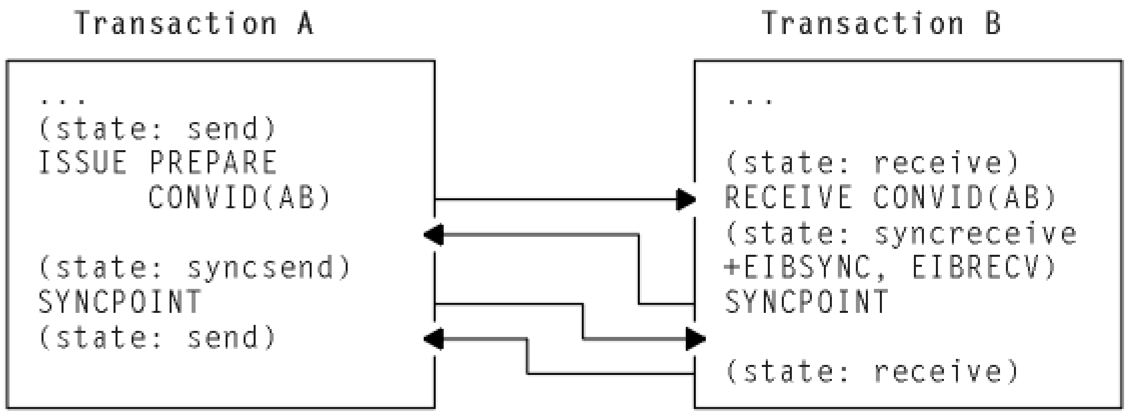

Figure 30. SYNCPOINT in response to ISSUE PREPARE on an APPC mapped conversation

SYNCPOINT ROLLBACK in response to SYNCPOINT ROLLBACK
In an APPC mapped conversation , a transaction issues a **SYNCPOINT ROLLBACK** command and its partner responds with a **SYNCPOINT ROLLBACK** command.

Figure 31 on page 101 illustrates a SYNCPOINT ROLLBACK command being used in response to SYNCPOINT ROLLBACK on an APPC mapped conversation. The figure also shows the conversation state before each command and the state and EIB fields set after each command.

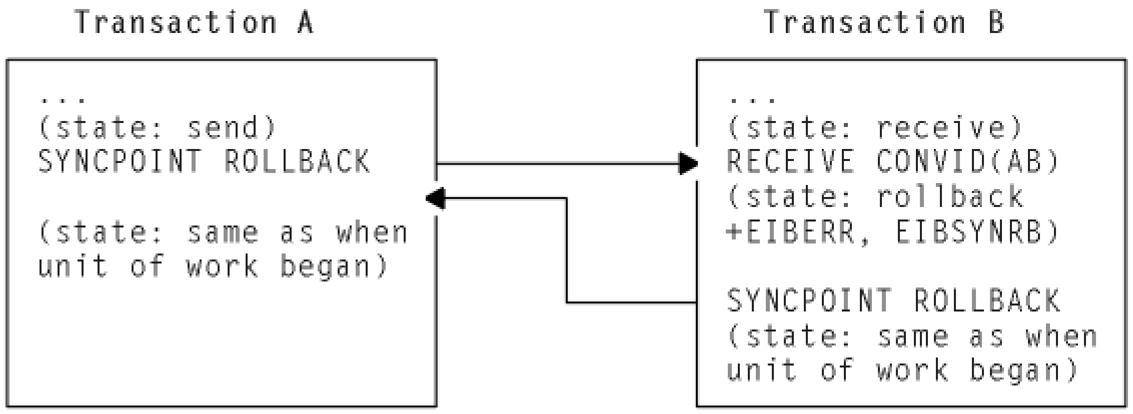
Transaction A
...
(state: send)
SYNCPOINT ROLLBACK
(state: same as when unit of work began)
Transaction B
...
(state: receive)
RECEIVE CONVID(AB)
(state: rollback +EIBERR, EIBSYNRB)
SYNCPOINT ROLLBACK
(state: same as when unit of work began)

Figure 31. SYNCPOINT ROLLBACK in response to SYNCPOINT ROLLBACK on an APPC mapped conversation

SYNCPOINT ROLLBACK in response to SYNCPOINT
In an APPC mapped conversation , a transaction issues a **SYNCPOINT** command and its partner responds with a **SYNCPOINT ROLLBACK** command.

Figure 32 on page 101 illustrates a SYNCPOINT ROLLBACK command being used in response to SYNCPOINT on an APPC mapped conversation. The figure also shows the conversation state before each command and the state and EIB fields set after each command.

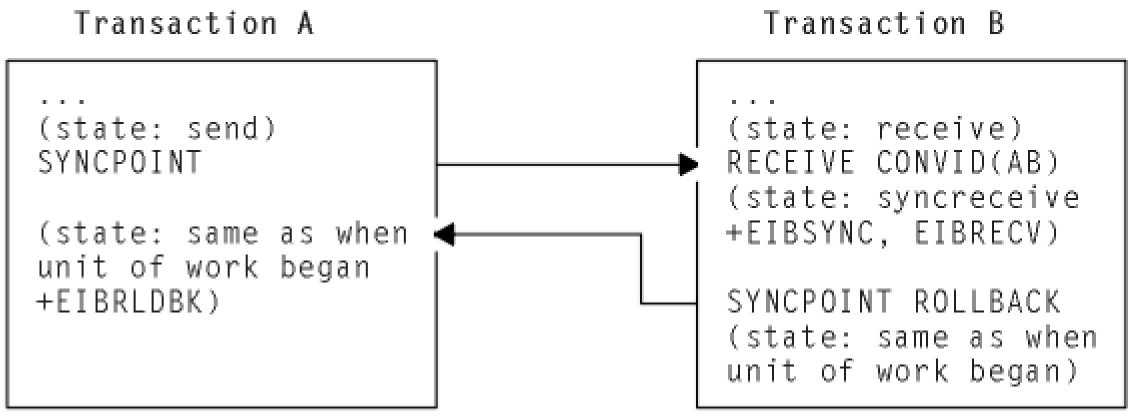
Transaction A
...
(state: send)
SYNCPOINT
(state: same as when unit of work began +EIBRLDBK)
Transaction B
...
(state: receive)
RECEIVE CONVID(AB)
(state: syncreceive +EIBSYNC, EIBRECV)
SYNCPOINT ROLLBACK
(state: same as when unit of work began)

Figure 32. SYNCPOINT ROLLBACK in response to SYNCPOINT on an APPC mapped conversation

SYNCPOINT ROLLBACK in response to ISSUE PREPARE
In an APPC mapped conversation , a transaction issues an **ISSUE PREPARE** command and its partner responds with a **SYNCPOINT ROLLBACK** command.

Figure 33 on page 102 illustrates a SYNCPOINT ROLLBACK command being used in response to ISSUE PREPARE on an APPC mapped conversation. The figure also shows the conversation state before each command and the state and EIB fields set after each command.

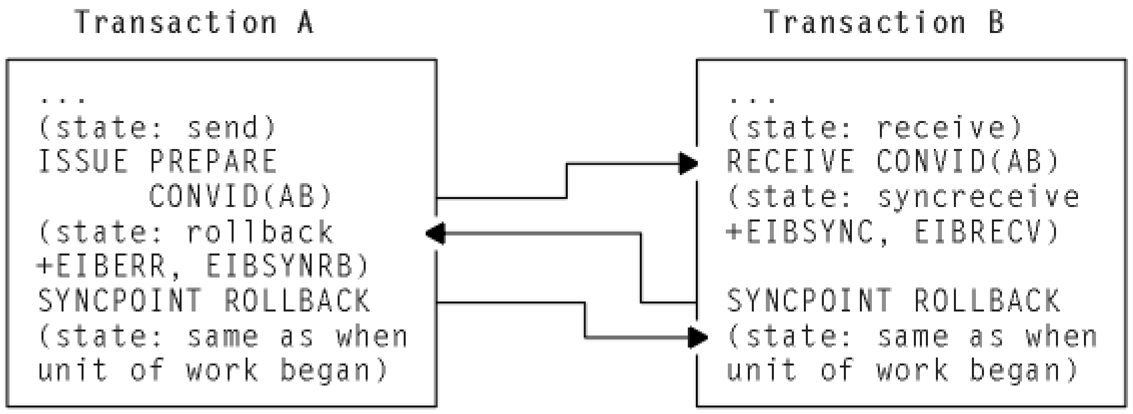

Figure 33. SYNCPOINT ROLLBACK in response to ISSUE PREPARE on an APPC mapped conversation

ISSUE ERROR in response to SYNCPOINT
In an APPC mapped conversation , a transaction issues a **SYNCPOINT** command and its partner responds with an **ISSUE ERROR** command.

Figure 34 on page 102 illustrates an ISSUE ERROR command being used in response to SYNCPOINT on an APPC mapped conversation. The figure also shows the conversation state before each command and the state and EIB fields set after each command. You can also send ISSUE ERROR before receiving SYNCPOINT; but this is not shown, because the results are the same.

It is pointless to use ISSUE ERROR as a response to SYNCPOINT, because this causes the syncpoint initiator to discard all data transmitted with the ISSUE ERROR by the syncpoint agent. To safeguard integrity, the syncpoint agent has to issue a SYNCPOINT ROLLBACK command.

Note that if transaction A were running on a CICS release earlier than 3.2, the results would be different.

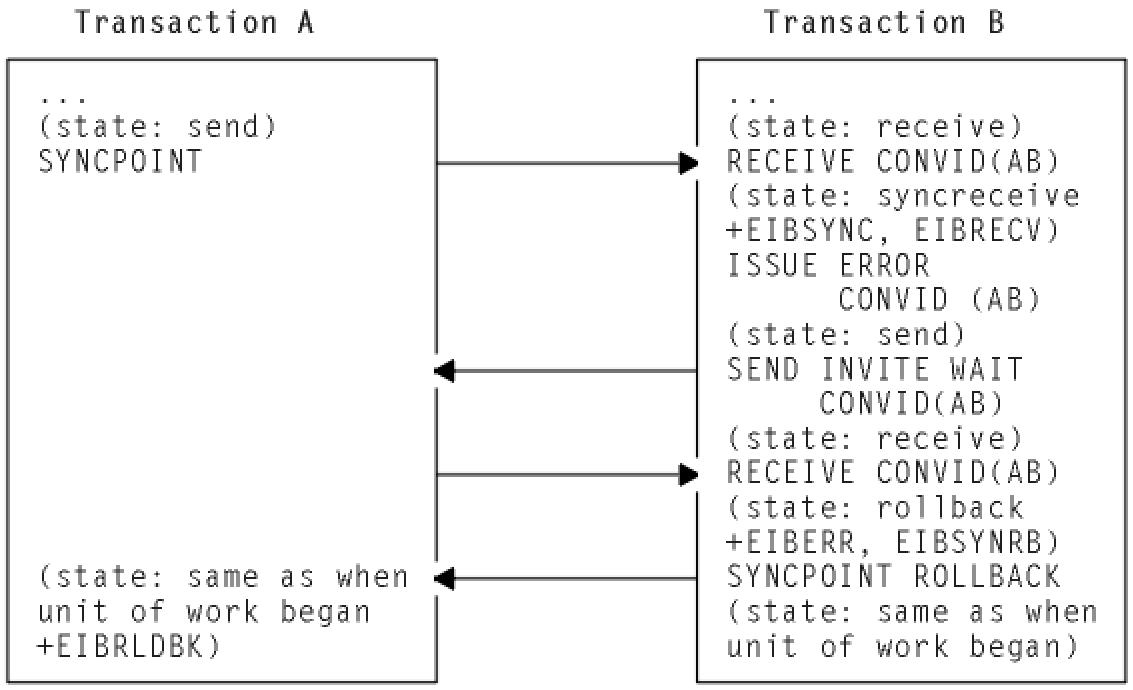

Figure 34. ISSUE ERROR in response to SYNCPOINT on an APPC mapped conversation

ISSUE ERROR in response to ISSUE PREPARE
In an APPC mapped conversation, a transaction issues a **SYNCPOINT** command and its partner responds with a **SYNCPOINT** command.

Figure 35 on page 103 illustrates an ISSUE ERROR command being used in response to ISSUE PREPARE on an APPC mapped conversation. The figure also shows the conversation state before each command

and the state and EIB fields set after each command. You can also send ISSUE ERROR before receiving ISSUE PREPARE; but this is not shown, because the results are the same.

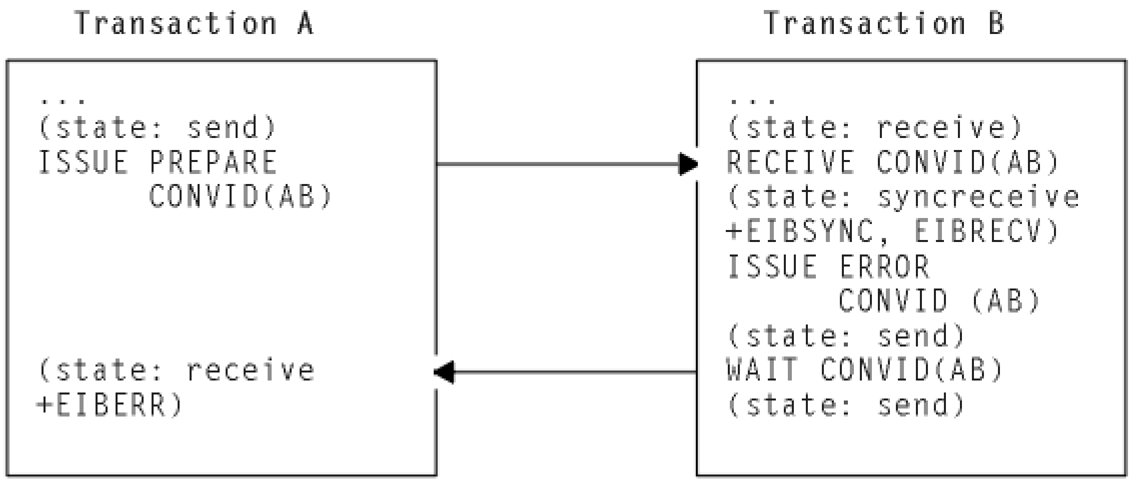

Figure 35. ISSUE ERROR in response to ISSUE PREPARE on an APPC mapped conversation

ISSUE ABEND in response to SYNCPOINT

In an APPC mapped conversation, a transaction issues a **SYNCPOINT** command and its partner responds with an **ISSUE ABEND** command.

Figure 36 on page 103 illustrates an ISSUE ABEND command being used in response to SYNCPOINT on an APPC mapped conversation. The figure also shows the conversation state before each command and the state and EIB fields set after each command. You can also send ISSUE ABEND before receiving SYNCPOINT; but this is not shown, because the results are the same.

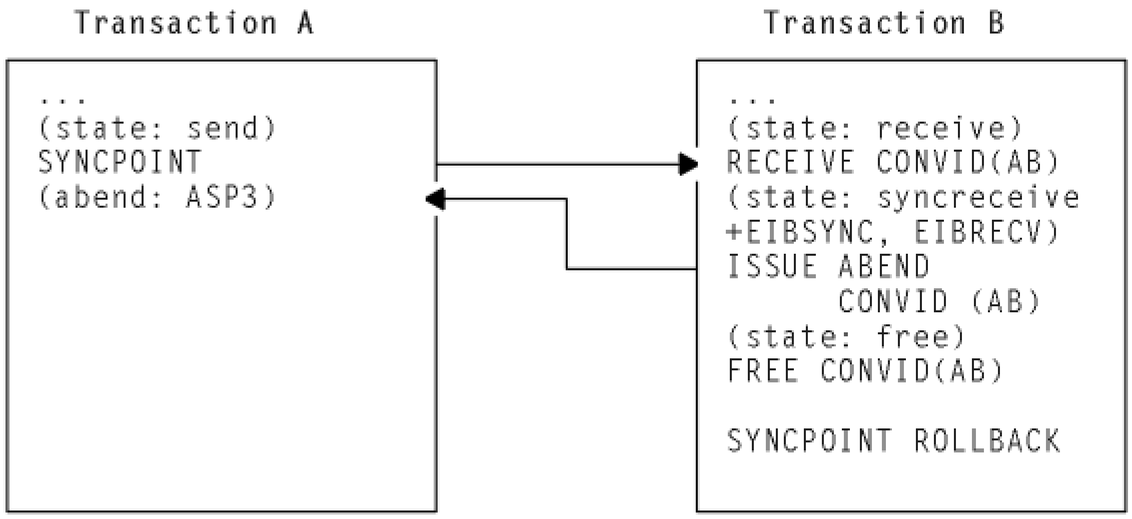

Figure 36. ISSUE ABEND in response to SYNCPOINT on an APPC mapped conversation

ISSUE ABEND in response to ISSUE PREPARE

In an APPC mapped conversation, a transaction issues an **ISSUE PREPARE** command and its partner responds with an **I SSUE ABEND** command.

Figure 37 on page 104 illustrates an ISSUE ABEND command being used in response to ISSUE PREPARE on an APPC mapped conversation. The figure also shows the conversation state before each command and the state and EIB fields set after each command. You can also send ISSUE ABEND before receiving ISSUE PREPARE; but this is not shown, because the results are the same.

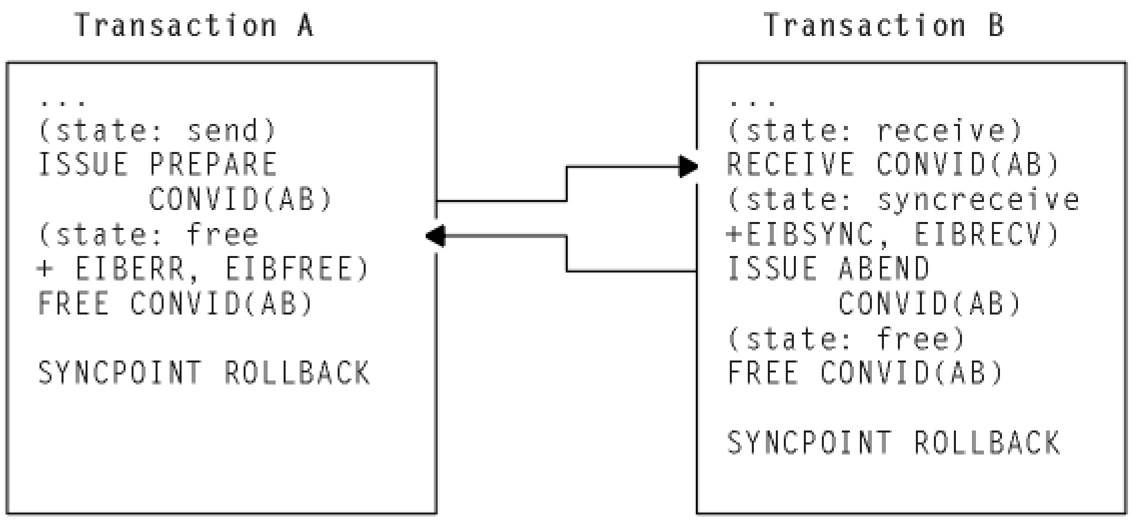

Figure 37. ISSUE ABEND in response to ISSUE PREPARE on an APPC mapped conversation

Session failure in response to SYNCPOINT
In an APPC mapped conversation, a transaction issues a **SYNCPOINT** command, its partner responds with a **SYNCPOINT** command, but the session fails before the first transaction receives the response.

Figure 38 on page 104 and Figure 39 on page 105 illustrate what happens if the session fails before or after a SYNCPOINT command issued in response to SYNCPOINT on an APPC mapped conversation. The figures also show the conversation state before each command and the state and EIB fields set after each command.

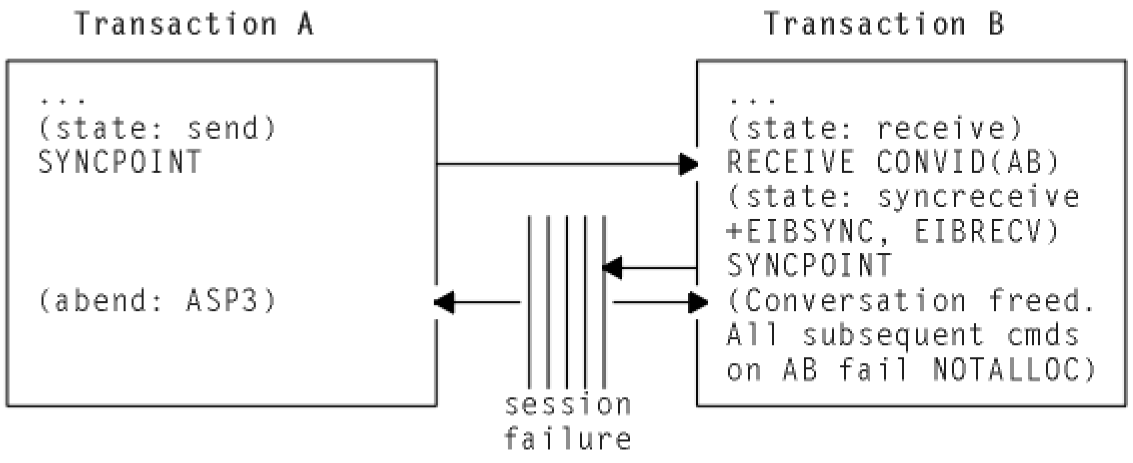

Figure 38. Session failure before SYNCPOINT in response to SYNCPOINT on an APPC mapped conversation

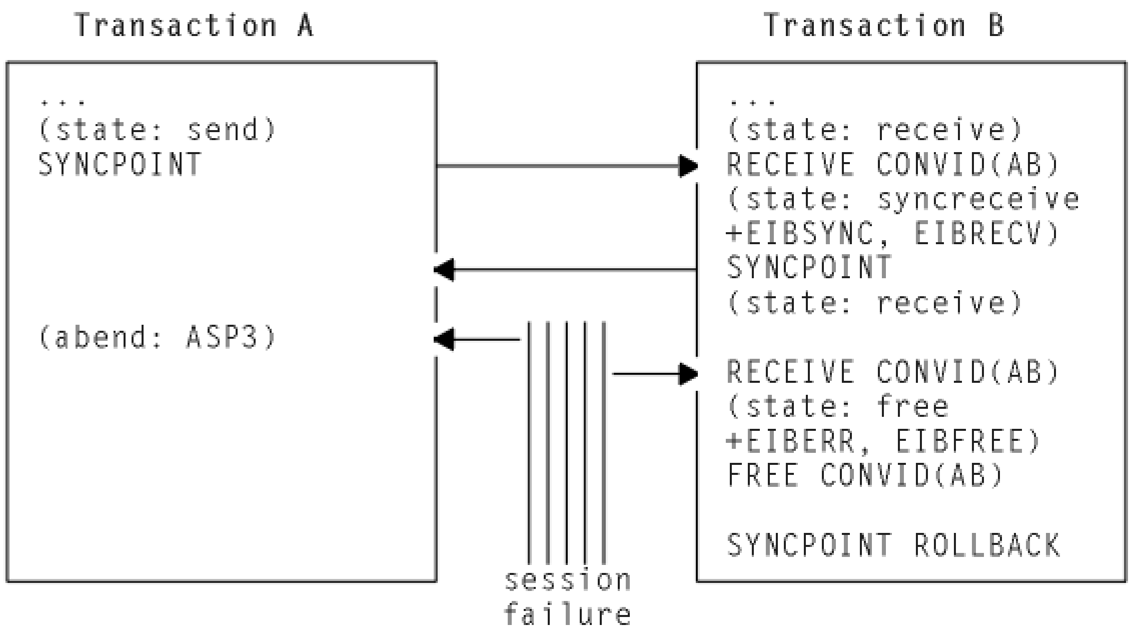

Figure 39. Session failure after SYNCPOINT in response to SYNCPOINT on an APPC mapped conversation

Session failure in response to ISSUE PREPARE

In an APPC mapped conversation, a transaction issues an **ISSUE PREPARE** command, its partner responds with a **SYNCPOINT** command, but the session fails before the first transaction receives the response.

Figure 40 on page 105 illustrates what happens if the session fails after ISSUE PREPARE is received by transaction B and before the SYNCPOINT response is received by transaction A on an APPC mapped conversation. The figure also shows the conversation state before each command and the state and EIB fields set after each command.

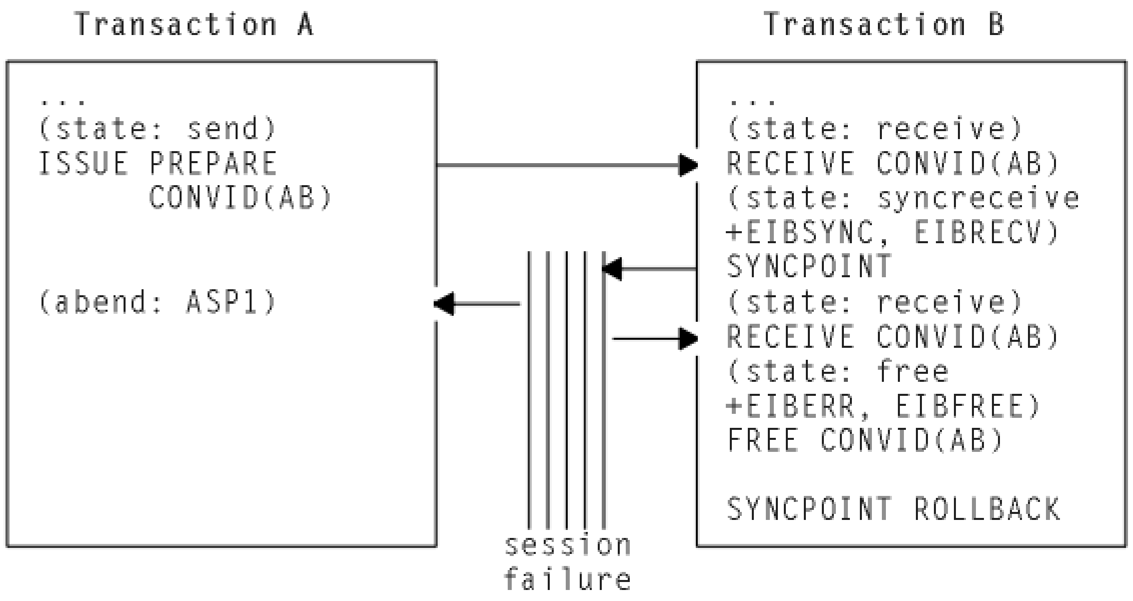

Figure 40. Session failure during SYNCPOINT in response to ISSUE PREPARE on an APPC mapped conversation

Session failure in response to SYNCPOINT ROLLBACK

In an APPC mapped conversation, a transaction issues a **SYNCPOINT ROLLBACK** command, its partner responds with a **SYNCPOINT ROLLBACK** command, but the session fails before the first transaction receives the response.

Figure 41 on page 106 illustrates what happens if the session fails after SYNCPOINT ROLLBACK is received and before the response is issued on an APPC mapped conversation. The figure also shows the conversation state before each command and the state and EIB fields set after each command.

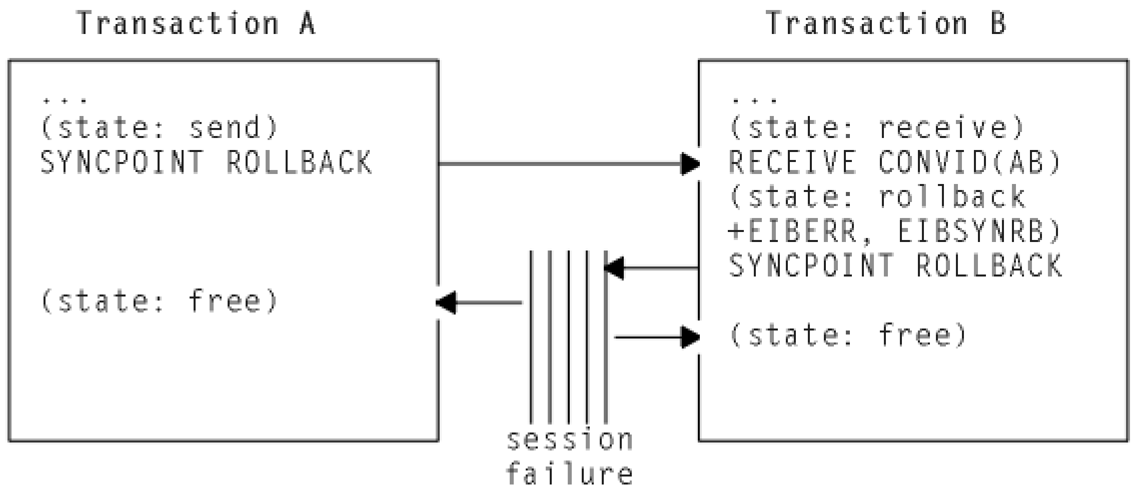

Figure 41. Session failure during SYNCPOINT ROLLBACK in response to SYNCPOINT ROLLBACK on an APPC mapped conversation

Synchronizing three or more CICS systems

This section gives examples of how to commit and back out recoverable resources affected by three or more DTP transactions connected on conversations at sync level 2.

SYNCPOINT in response to SYNCPOINT

In a complex distributed transaction, all the agents agree that the transaction should be committed.

Figure 42 on page 107 shows the sequence of events for a successful syncpoint involving six conversing transactions:

Transaction A

- is in conversation with transactions B and D. Before the syncpoint, its conversations with B and D are in send state.
- is the syncpoint initiator with respect to transactions B and D.

Transaction B

- is in conversation with transactions A, C, and E. Before the syncpoint, its conversation with A is in receive state, and its conversations with C and E are in send state.
- is a syncpoint agent of transaction A, and the syncpoint initiator with respect to transactions C and E.

Transaction C

- is in conversation with transaction B. Before the syncpoint, its conversation with B is in receive state.
- is a syncpoint agent of transaction B.

Transaction D

- is in conversation with transactions A and F. Before the syncpoint, its conversation with A is in receive state, and its conversation F is in send state.
- is a syncpoint agent of transaction A, and the syncpoint initiator with respect to transaction F.

Transaction E

- is in conversation with transaction B. Before the syncpoint, its conversation with B is in receive state.
- is a syncpoint agent with respect to transaction B.

Transaction F

- is in conversation with transaction D. Before the syncpoint, its conversation with D is in receive state.
- is the only syncpoint agent of transaction D.

It illustrates the states and actions that occur when transactions issue SYNCPOINT requests. To write successful distributed applications you do not need to understand all the data flows that take place during a distributed syncpoint. In this example, the programmer is concerned only with issuing SYNCPOINT in response to finding a conversation in **syncreceive state** (state 9).

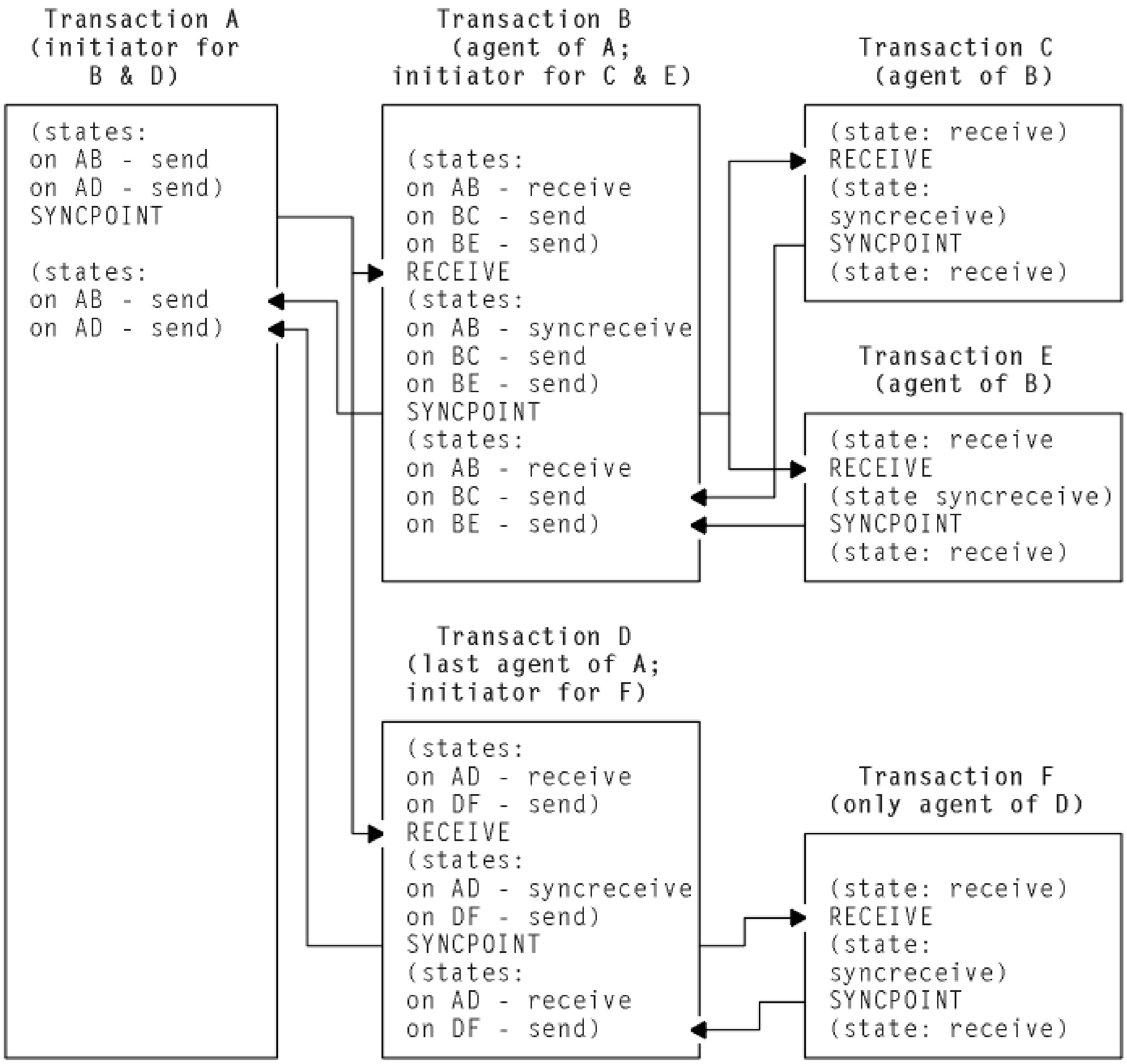

Figure 42. A distributed syncpoint with all partners running on CICS Transaction Server for z/OS, Version 5 Release 4

1. Transaction A, which is in **send state** (state 2) on its conversations with transactions B and D, decides to end the distributed unit of work, and therefore issues a SYNCPOINT command.

2. Transaction B sees that its half of its conversation with transaction A is in **syncreceive state** (state 9), so it issues a SYNCPOINT command. Transaction B is responding to a request from transaction A, but it also becomes the syncpoint initiator for transactions C and E, and must ensure that its conversations with these transactions are in a valid state for issuing a SYNCPOINT command. In this example, they are both in **send state** (state 2).
3. Transaction C sees that its half of its conversation with transaction B is in **syncreceive state** (state 9), so it issues a SYNCPOINT command.
4. Transaction E sees that its half of its conversation with transaction B is in **syncreceive state** (state 9), so it issues a SYNCPOINT command.
5. Transaction D sees that its half of its conversation with transaction A is in **syncreceive state** (state 9), so it issues a SYNCPOINT command. Transaction D is responding to a request from transaction A, but it also becomes the syncpoint initiator for transaction F, and must ensure that its conversation with this transaction is in a valid state for issuing a SYNCPOINT command. In this example, it is in **send state** (state 2).
6. Transaction F sees that its half of its conversation with transaction D is in **syncreceive state** (state 9), so it issues a SYNCPOINT command.
7. All the transactions have now indicated, by issuing SYNCPOINT commands, that they are ready to commit their changes. This process begins with transaction F, which has no agents and has responded to "request commit" by issuing a SYNCPOINT command.
8. The distributed syncpoint is complete and control returns to transaction A following the SYNCPOINT command.

The previous discussion of the SYNCPOINT command assumed that all the agent transactions were ready to take a syncpoint by issuing SYNCPOINT when their conversation entered **syncreceive state** (state 9).

If, however, an agent has detected an error, it can reject the syncpoint request with one of the following commands:

- SYNCPOINT ROLLBACK (preferred response)
- ISSUE ERROR
- ISSUE ABEND

The SYNCPOINT ROLLBACK command enables a transaction to initiate a backout operation across the entire distributed unit of work. When it is issued in response to a syncpoint request, it has the following effects:

1. Any changes made to recoverable resources by the transaction that issues the rollback request are backed out.
2. The syncpoint initiator is also backed out (EIBRLDBK set).

This causes the syncpoint initiator to initiate a backout operation across the distributed unit of work.

SYNCPOINT ROLLBACK in response to SYNCPOINT

In a complex distributed transaction, during sync point processing, one of the agents determines that the transaction should be backed out.

Figure 43 on page 110 shows the sequence of events for a sync point involving six conversing transactions, when one of the agents determines that the distributed transaction should be backed out. The topology, and initial states are the same as in Figure 42 on page 107 :

Transaction A

- is in conversation with transactions B and D. Before the syncpoint, its conversations with B and D are in send state.
- is the sync point initiator with respect to transactions B and D.

Transaction B

- is in conversation with transactions A, C, and E. Before the syncpoint, its conversation with A is in receive state, and its conversations with C and E are in send state.

- is a sync point agent of transaction A, and the sync point initiator with respect to transactions C and E.

Transaction C

- is in conversation with transaction B. Before the syncpoint, its conversation with B is in receive state.
- is a sync point agent of transaction B.

Transaction D

- is in conversation with transactions A and F. Before the syncpoint, its conversation with A is in receive state, and its conversation F is in send state.
- is a sync point agent of transaction A, and the sync point initiator with respect to transaction F.

Transaction E

- is in conversation with transaction B. Before the syncpoint, its conversation with B is in receive state.
- is a sync point agent with respect to transaction B.

Transaction F

- is in conversation with transaction D. Before the syncpoint, its conversation with D is in receive state.
- is the only sync point agent of transaction D.

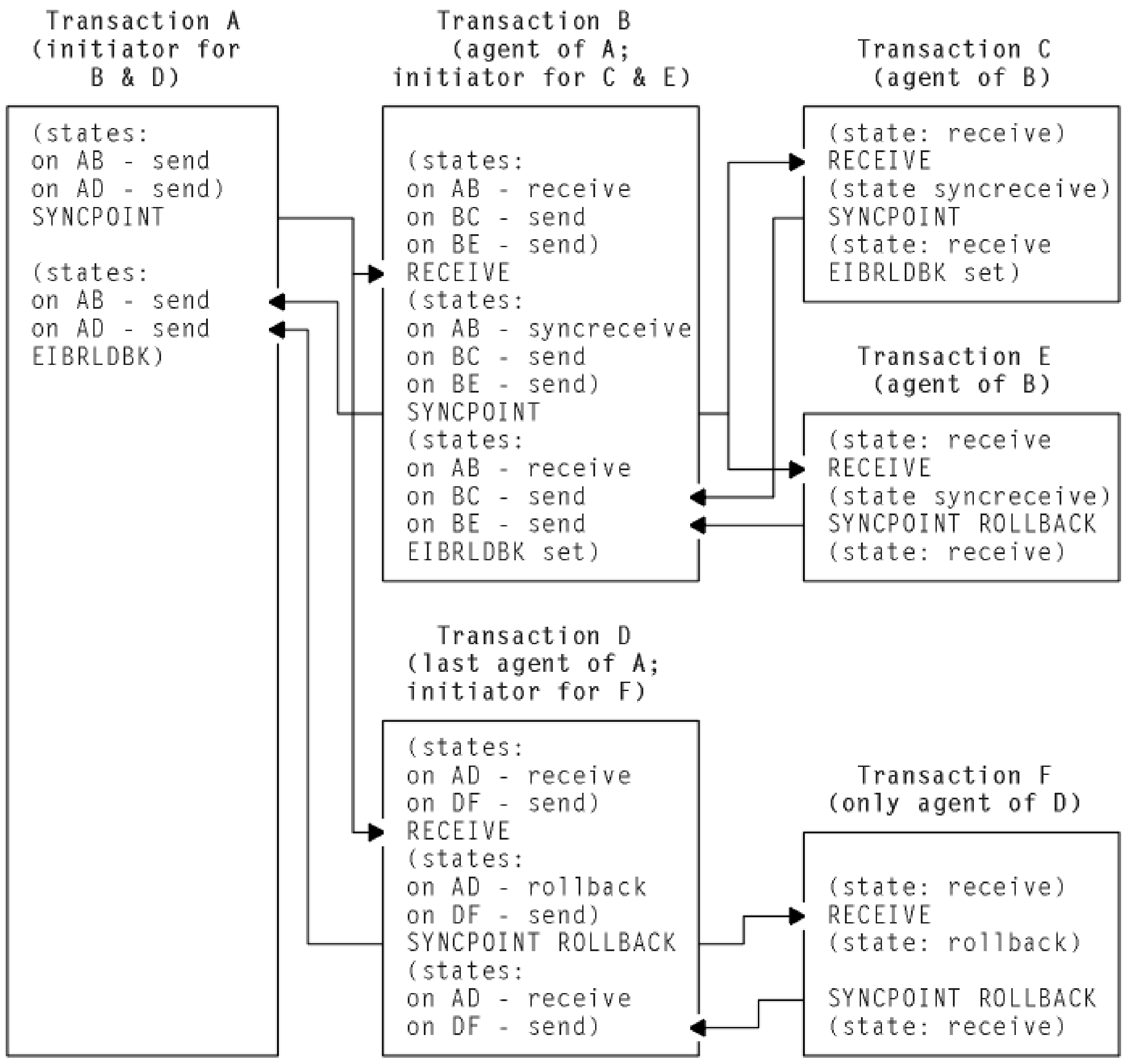
Transaction A (initiator for B & D)
(states: on AB - send on AD - send) SYNCPOINT
(states: on AB - send on AD - send EIBRLDBK)
Transaction B (agent of A; initiator for C & E)
(states: on AB - receive on BC - send on BE - send) RECEIVE
(states: on AB - syncreceive on BC - send on BE - send) SYNCPOINT
(states: on AB - receive on BC - send on BE - send EIBRLDBK set)
Transaction C (agent of B)
(state: receive) RECEIVE (state syncreceive) SYNCPOINT (state: receive EIBRLDBK set)
Transaction E (agent of B)
(state: receive RECEIVE (state syncreceive) SYNCPOINT ROLLBACK (state: receive)
Transaction D (last agent of A; initiator for F)
(states: on AD - receive on DF - send) RECEIVE
(states: on AD - rollback on DF - send) SYNCPOINT ROLLBACK
(states: on AD - receive on DF - send)
Transaction F (only agent of D)
(state: receive) RECEIVE (state: rollback)
SYNCPOINT ROLLBACK (state: receive)

Figure 43. Rollback during distributed syncpointing

As in Figure 42 on page 107 , transaction A (while in **send state** , state 2) issues the SYNCPOINT command, and CICS initiates a chain of events. Here, however, transaction E has detected an error that makes it unable to commit, and it issues SYNCPOINT ROLLBACK when it detects that the conversation on its principal facility is in **syncreceive state** (state 9, EIBSYNC is also set). This causes any changes that transaction E has made to be backed out, and initiates a distributed rollback.

Transactions B, C and A are rolled back (EIBRLDBK set). Transaction D senses that the conversation on its principal facility is in **rollback state** (state 13, EIBSYNRB is also set), and issues a SYNCPOINT ROLLBACK command. Transaction F too senses that the conversation on its principal facility is in **rollback state** , and issues a SYNCPOINT ROLLBACK command. The distributed rollback is now complete.

Session failure and the indoubt period

During the period between the sending of the syncpoint request to the partner region and the receipt of the reply, the local region does not know whether the partner region has committed the change. This is known as the *indoubt period* . If the intersystem session fails during this period, the local CICS system cannot tell whether the partner region has committed or backed out its resource changes.

This situation could occur for situations other than DTP and is discussed in Recovery functions and interfaces.

What really flows between APPC systems

This topic describes the commit protocols that flow between APPC systems during a syncpoint.

First, consider a simple distributed process involving only one conversation, as in Figure 44 on page 111 . Here is what happens:

1. The syncpoint initiator sends a "commit" request to the syncpoint agent.
2. The syncpoint agent commits all changes it made to recoverable resources, and responds with “committed”.
3. The syncpoint initiator then commits its changes, and the UOW is complete.

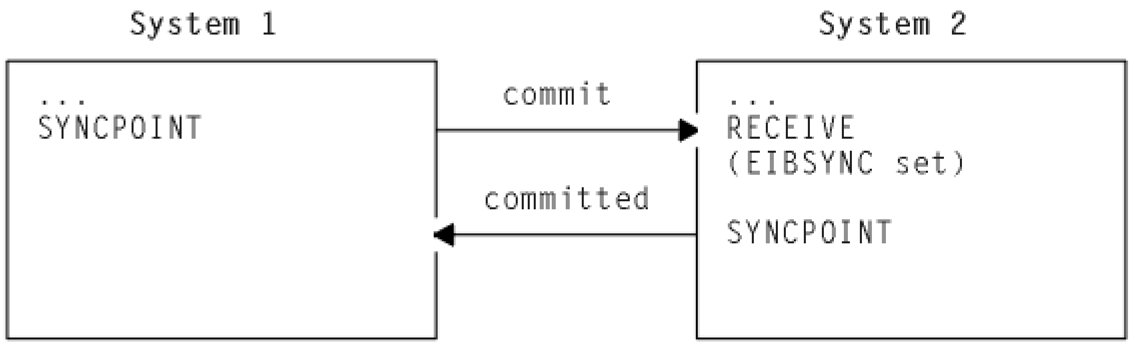

Figure 44. Syncpoint flows in a single conversation

When the syncpoint agent has a conversation with a third transaction, Figure 45 on page 111 shows the flows that occur. Here is what happens:

1. The syncpoint initiator sends a "commit" request to its agent.
2. The agent becomes the initiator on the conversation to its agent, and sends a "commit" request.
3. The second agent commits first and responds with “committed”.
4. The first agent commits and sends "committed" to the initiator.
5. The initiator commits.

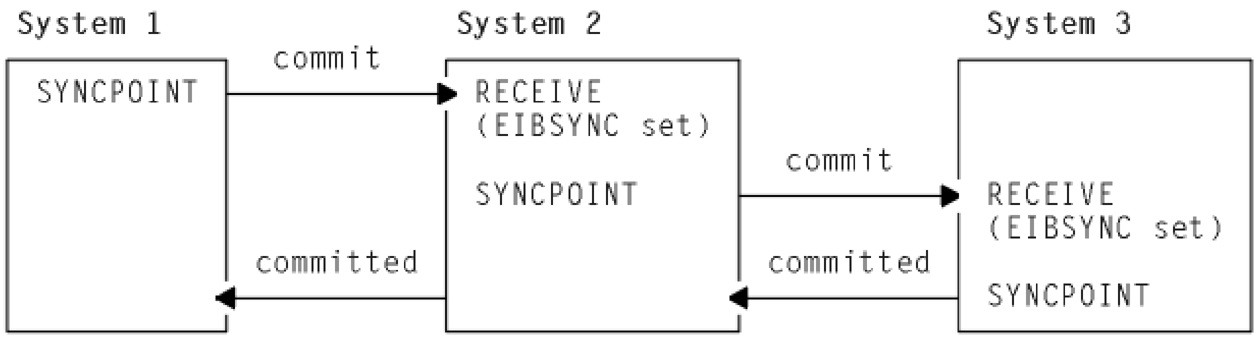

Figure 45. Syncpoint flows in concurrent conversations

When the syncpoint initiator has two concurrent conversations, the flows involved are shown in Figure 46 on page 112. Here is what happens:

1. The syncpoint initiator sends a "prepare" request to all its agents except one.
2. The agent receiving "prepare" responds by sending a "commit" request to the initiator.
3. When all the "prepare" requests have been sent, and the "commit" requests received, the initiator sends a “commit” request to its last agent.
4. The initiator receives "committed" from the last agent.
5. The initiator sends "committed" to the remaining agents.
6. The agents respond "forget" to indicate that they do not need to be resynchronized.

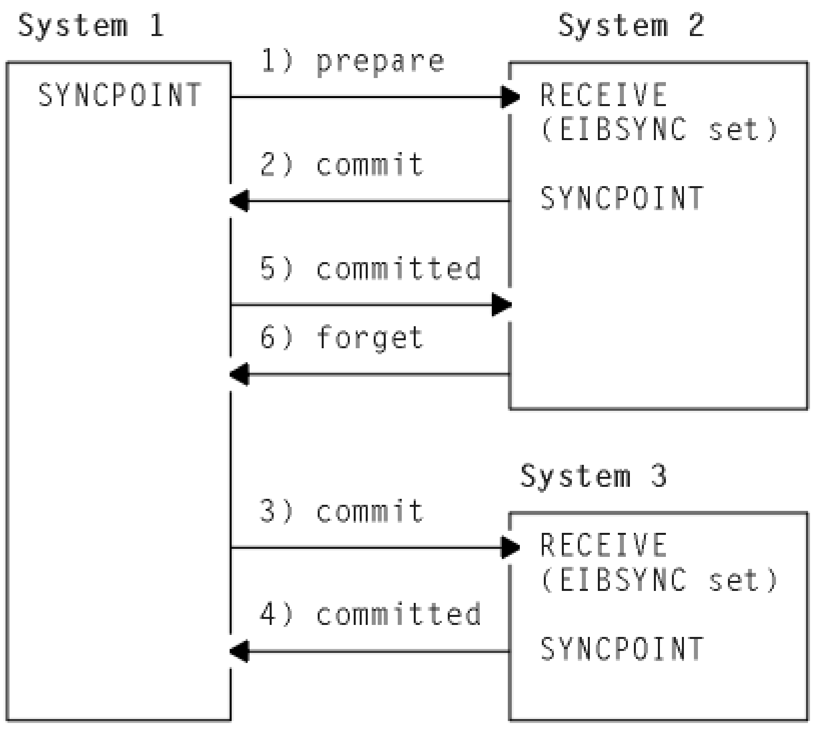

Figure 46. Syncpoint flows in concurrent conversations with one initiator

If the syncpoint initiator decides to prepare the conversation with system 2 explicitly before issuing a syncpoint, the flows involved are shown in Figure 47 on page 112. In this case, the application program in system 1 issues an ISSUE PREPARE command, followed by SYNCPOINT command, rather than just a SYNCPOINT command; however, the flows across the links are exactly the same as those in the previous example. Using the ISSUE PREPARE command gives the application the opportunity to "change its mind" and rollback, depending on the response to ISSUE PREPARE.

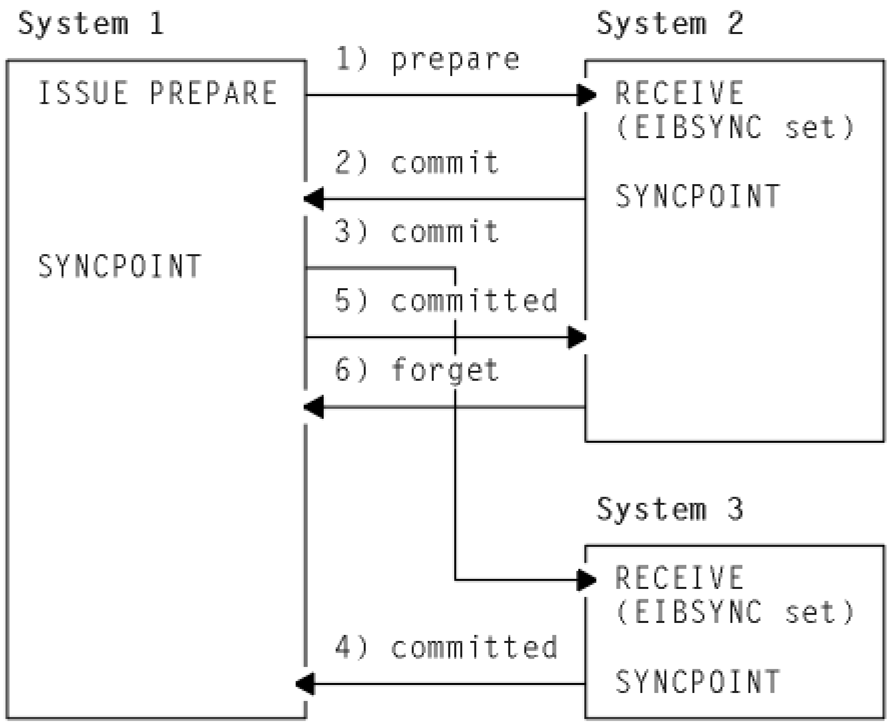

Figure 47. Syncpoint flows in concurrent conversations with one initiator

For further information on the flows in a distributed process, see the book *LU6.2 Reference: Peer Protocols* , SC31-6808.

Appendix A. CICS mapping to the APPC architecture

This appendix shows how the APPC programming language is implemented by CICS.

The APPC programming language is described in the SNA publication *Transaction Programmer's Reference Manual for LU Type 6.2*.

For information on how the CICS application programming interface for basic and unmapped conversations maps to the APPC verbs, see the .

Supported option sets

This table shows which APPC option sets are supported by CICS and which are not.

Table 46. CICS support of APPC options sets

Set #	Set name	Supported
101	Clear the LU's send buffer	Yes
102	Get attributes	Yes
103	Post on receipt with test for posting	No
104	Post on receipt with wait	No
105	Prepare to receive	Yes
106	Receive immediate **Note:** CICS programs support receive_immediate requests provided these requests are coded using the common programming Interface for communications.	Yes
108	Sync point services	Yes
109	Get TP name and instance identifier	No
110	Get conversation type	Yes
111	Recovery from program errors detected during syncpoint	Yes
201	Queued allocation of a contention-winner session	No
203	Immediate allocation of a session	Yes
204	Conversations between programs located at the same LU	No
211	Session-level LU-LU verification	Yes
212	User ID verification	Yes
213	Program-supplied user ID and password	No
214	User ID authorization	Yes
215	Profile verification and authorization	Yes
217	Profile pass-through	No
218	Program-supplied profile	No
241	Send PIP data	Yes
242	Receive PIP data	Yes

Table 46. CICS support of APPC options sets (continued)

Set #	Set name	Supported
243	Accounting	Yes
244	Long locks	No
245	Test for request-to-send received	Yes
246	Data mapping	No
247	FMH data	No
249	Vote read-only response to a syncpoint operation	No
251	Extract transaction and conversation identity information	No
290	Logging of data in a system log	No
291	Mapped conversation LU services component	Yes
401	Reliable one-way brackets	No
501	CHANGE_SESSION_LIMIT verb	Yes
502	ACTIVATE_SESSION verb	Yes
504	DEACTIVATE_SESSION verb	No
505	LU-definition verbs	Yes
601	MIN_CONWINNERS_TARGET parameter	No
602	RESPONSIBLE(TARGET) parameter	No
603	DRAIN_TARGET(NO) parameter	No
604	FORCE parameter	No
605	LU-LU session limit	No
606	Locally known LU names	Yes
607	Uninterpreted LU names	No
608	Single-session reinitiation	No
610	Maximum RU size bounds	Yes
611	Session-level mandatory cryptography	No
612	Contention-winner automatic activation limit	No
613	Local maximum (LU, mode) session limit	Yes
616	CPSVCMG modename support	No
617	Session-level selective cryptography	No

CICS implementation of control operator verbs

This section describes how CICS implements the APPC control operator verbs. It includes tables showing how these verbs map to CICS commands. CICS supports control operator verbs in a variety of ways.

Some verbs are supported by the CICS master terminal transaction CEMT. The relevant CEMT commands are:

- **`CEMT INQUIRE CONNECTION`**

- **`CEMT SET CONNECTION`**
- **`CEMT INQUIRE MODENAME`**
- **`CEMT SET MODENAME`**

Tip: In CICS Explorer, the ISC/MRO connections operations view provides a functional equivalent to the INQUIRE and SET CONNECTION commands. See ISC/MRO Connections view in the CICS Explorer product documentation.

CEMT is normally entered by an operator at a display device. It is described in CEMT - master terminal.

The inquire and set operations for connections and modenames are also available at the CICS SPI, using the following commands:

- **`EXEC CICS INQUIRE CONNECTION`**
- **`EXEC CICS SET CONNECTION`**
- **`EXEC CICS INQUIRE MODENAME`**
- **`EXEC CICS SET MODENAME`**

Programming information about these commands is given in INQUIRE CONNECTION.

Some control operator verbs are supported by CICS resource definition. The definition of APPC links is described in Defining APPC links.

You can change some CONNECTION and SESSION attributes while CICS is running by discarding the resource and creating a new one.

Control operator verbs

The way in which CICS implements APPC control operator verbs is shown in a set of tables.

See "Return codes for control operator verbs" on page 124 for details of the corresponding return code mapping.

Note: Wherever CEMT is shown, the equivalent form of EXEC CICS command can be used.

Tip: In CICS Explorer®, the **ISC/MRO Connections** view provides a functional equivalent to the SET and INQUIRE CONNECTION commands. The **Terminals**, **Local Transactions**, and **Remote Transactions** views provide functional equivalents to the INQUIRE TERMINAL and INQUIRE TRANSACTION commands, respectively. See CICSPlex SM Operations views in the CICS Explorer product documentation.

Table 47. CHANGE_SESSION_LIMIT

CHANGE_SESSION_LIMIT	**CEMT SET MODENAME**
LU_NAME(vble)	CONNECTION( )
MODE_NAME(vble)	MODENAME( )
LU_MODE_SESSION_LIMIT(vble)	AVAILABLE( )
MIN_CONWINNERS_SOURCE(vble)	CICS negotiates a revised value, based on the AVAILABLE request and the MAXIMUM attribute of the SESSIONS resource.
MIN_CONWINNERS_TARGET(vnle)	Not supported.
RESPONSIBLE(source)	Yes.
RESPONSIBLE(target)	Not supported. CICS does not support receipt of RESP(TARGET).
RETURN_CODE	Supported.

Table 48. INITIALIZE_SESSION_LIMIT

INITIALIZE_SESSION_LIMIT	**Specified in SESSIONS resource**
LU_NAME(vble)	CONNECTION( )
MODE_NAME(vble)	MODENAME( )
LU_MODE_SESSION_LIMIT(vble)	MAXIMUM(value1,)
MIN_CONWINNERS_SOURCE(vble)	MAXIMUM(,value2)
MIN_CONWINNERS_TARGET(vnle)	Not supported.
RETURN_CODE	Supported.

Table 49. PROCESS_SESSION_LIMIT

PROCESS_SESSION_LIMIT	**Automatic action by CICS-supplied transaction CLS1 when CNOS is received by a target CICS system.**
RESOURCE(vble)	Connection resource.
LU_NAME(vble)	Passed internally.
MODE_NAME(vble1,vble2)	Passed internally.
RETURN_CODE	Supported.

Table 50. RESET_SESSION_LIMIT

RESET_SESSION_LIMIT	**CEMT SET MODENAME (for individual modegroups) or CEMT SET CONNECTION RELEASED (to reset all modegroups)**
LU_NAME(vble)	CONNECTION( )
MODE_NAME(ALL)	SET CONNECTION( ) RELEASED
MODE_NAME(ONE(vble))	MODENAME( ) AVAILABLE(0)
MODE_NAME(ONE('SNASVCMG'))	SET CONNECTION( ) RELEASED
RESPONSIBLE(SOURCE)	Yes.
RESPONSIBLE(TARGET)	Not supported.
DRAIN_SOURCE(NO|YES)	CICS supports YES.
DRAIN_TARGET(NO|YES)	CICS supports YES.
FORCE(NO|YES)	Not supported.
RETURN_CODE	Supported.

Table 51. ACTIVATE_SESSION

ACTIVATE_SESSION	**CEMT SET MODENAME ACQUIRED (for individual modegroups) or CEMT SET CONNECTION ACQUIRED (for SNASVCMG sessions)**
LU_NAME(vble)	CONNECTION( )
MODE_NAME(vble)	MODENAME( ) ACQUIRED

Table 51. ACTIVATE_SESSION (continued)

ACTIVATE_SESSION	CEMT SET MODENAME ACQUIRED (for individual modegroups) or CEMT SET CONNECTION ACQUIRED (for SNASVCMG sessions)
MODE_NAME('SNASVCMG')	Activated when CEMT SET CONNECTION ACQUIRED is issued.
RETURN_CODE	Supported.

Table 52. DEACTIVATE_CONVERSATION_GROUP

DEACTIVATE_CONVERSATION_GROUP	Not supported.

Table 53. DEACTIVATE_SESSION

DEACTIVATE_SESSION	Not supported.

Table 54. DEFINE_LOCAL_LU

DEFINE_LOCAL_LU	SESSION resource and system initialization parameters
FULLY_QUALIFIED_LU_NAME(vble)	Cannot be specified. CICS uses the network LU name (APPLID on DFHSIT).
LU_SESSION_LIMIT(NONE)	Not supported.
LU_SESSION_LIMIT(VALUE(vble))	Total of MAX(nn) on all sessions.
SECURITY(ADD USER_ID(vble))	In an external security manager (ESM).
SECURITY(ADD PASSWORD(vble))	Not supported; defined in an ESM.
SECURITY(ADD PROFILE(vble))	Not supported; defined in an ESM.
SECURITY(DELETE USER_ID(vble))	Supported in an ESM.
SECURITY(DELETE PASSWORD(vble))	Not supported; defined in an ESM.
MAP_NAME(ADD(vble))	Not supported.
MAP_NAME(DELETE(vble))	Not supported.
BIND_RSP_QUEUE_CAPACITY(YES|NO)	Not supported.

Table 55. DEFINE_MODE

DEFINE_MODE	EXEC CICS CONNECT PROCESS + MODEENT macro (ACF/Communications Server systems definition) + SESSIONS resource
FULLY_QUALIFIED_LU_NAME(vble)	Cannot be specified. LU identified via CONNECTION on SESSIONS.
MODE_NAME(vble)	MODENAME on SESSIONS is mapped to LOGMODE on MODEENT.
SEND_MAX_RU_SIZE_LOWER_BOUND (vble)	Fixed at 8.
SEND_MAX_RU_SIZE_UPPER_BOUND (vble)	SENDSIZE on SESSIONS.
PREFERRED_RECEIVE_RU_SIZE (vble)	Not supported.

Table 55. DEFINE_MODE (continued)

DEFINE_MODE	EXEC CICS CONNECT PROCESS + MODEENT macro (ACF/Communications Server systems definition) + SESSIONS resource
PREFERRED_SEND_RU_SIZE (vble)	Not supported.
RECEIVE_MAX_RU_SIZE_LOWER _BOUND (vble)	Fixed at 256.
RECEIVE_MAX_RU_SIZE_UPPER _BOUND (vble)	RECEIVESIZE on SESSIONS.
SINGLE_SESSION_REINITIATION OPERATOR	Not supported.
SINGLE_SESSION_REINITIATION PLU	Not supported.
SINGLE_SESSION_REINITIATION SLU	Not supported.
SINGLE_SESSION_REINITIATION PLU_OR_SLU	Not supported.
SESSION_LEVEL_CRYPTOGRAPHY (NOT_SUPPORTED)	Default.
SESSION_LEVEL_CRYPTOGRAPHY (MANDATORY)	Not supported.
SESSION_LEVEL_CRYPTOGRAPHY (SELECTIVE)	Not supported.
CONWINNER_AUTO_ACTIVATE_LIMIT (vble)	MAXIMUM(,value2) on SESSIONS.
SESSION_DEACTIVATED_TP_NAME (vble)	Not supported.
LOCAL_MAX_SESSION_LIMIT (vble)	MAXIMUM(nn,) on SESSIONS.

Table 56. DEFINE_REMOTE_LU

DEFINE_REMOTE_LU	CONNECTION resource
FULLY_QUALIFIED_LU_NAME(vble)	Cannot be specified.
LOCALLY_KNOWN_LU_NAME(NONE)	Not supported.
LOCALLY_KNOWN_LU_NAME (NAME(vble))	CONNECTION(name)
UNINTERPRETED_LU_NAME(NONE)	Defaults to CONNECTION(name).
UNINTERPRETED_LU_NAME (NAME(vble))	NETNAME on CONNECTION.
INITIATE_TYPE(INITIATE_ONLY)	Not supported.
INITIATE_TYPE(INITIATE_OR_QUEUE)	Not supported.
PARALLEL_SESSION_SUPPORT(YES|NO)	SINGLESESS(NO|YES) on CONNECTION.
CNOS_SUPPORT(YES|NO)	Always YES.
LU_LU_PASSWORD(NONE)	Default on CONNECTION.
LU_LU_PASSWORD(VALUE(vble))	BINDPASSWORD on CONNECTION, or SESSKEY in RACF® APPCLU profile.
SECURITY_ACCEPTANCE(NONE)	ATTACHSEC(LOCAL)
SECURITY_ACCEPTANCE (CONVERSATION)	ATTACHSEC(VERIFY)
SECURITY_ACCEPTANCE (ALREADY_VERIFIED)	ATTACHSEC(IDENTIFY) or ATTACHSEC(PERSISTENT).

Table 57. DEFINE_TP

DEFINE_TP	TRANSACTION resource
TP_NAME(vble)	TRANSACTION(name)
STATUS(ENABLED)	STATUS(ENABLED)
STATUS(TEMP_DISABLED)	Not supported.
STATUS(PERM_DISABLED)	STATUS(DISABLED)
CONVERSATION_TYPE(MAPPED|BASIC)	Supported for all TPs (determined by choice of command).
SYNC_LEVEL(NONE|CONFIRMvSYNCPT)	SYNCPT for all TPs (actual level specified on CONNECT PROCESS).
SECURITY_REQUIRED(NONE)	Not supported; defined in an ESM.
SECURITY_REQUIRED(CONVERSATION)	Not supported; defined in an ESM.
SECURITY_REQUIRED (ACCESS(PROFILE))	Not supported.
SECURITY_REQUIRED (ACCESS(USER_ID))	Not supported; defined in an ESM.
SECURITY_REQUIRED (ACCESS(USER_ID_PROFILE))	Not supported.
SECURITY_ACCESS(ADD(USER_ID(vble)))	Transaction can be redefined.
SECURITY_ACCESS(ADD(PROFILE(vble)))	Transaction can be redefined.
SECURITY_ACCESS (DELETE(USER_ID(vble)))	Transaction can be redefined.
SECURITY_ACCESS (DELETE(PROFILE(vble)))	Transaction can be redefined.
PIP(NO)	Specified for all TPs.
PIP(YES(vble))	Specified on CONNECT PROCESS.
PIP(NO_LU_VERIFICATION)	Default for all PIP data.
DATA_MAPPING(NO|YES)	DATA_MAPPING(NO) for all TPs.
FMH_DATA(NO|YES)	FMH_DATA(YES) for all TPs.
PRIVILEGE(NONE)	Not supported.
PRIVILEGE(CNOS)	Not supported.
PRIVILEGE(SESSION_CONTROL)	Not supported.
PRIVILEGE(DEFINE)	Not supported.
PRIVILEGE(DISPLAY)	Not supported.
PRIVILEGE(ALLOCATE_SERVICE_TP)	Not supported.
INSTANCE_LIMIT(vble)	Not supported.
RETURN_CODE	Supported.

Table 58. DELETE

DELETE	EXEC CICS DISCARD
LOCAL_LU_NAME(vble)	Not supported.
REMOTE_LU_NAME	Not supported.

Table 58. DELETE (continued)

DELETE	EXEC CICS DISCARD
MODE_NAME	Not supported.
TP_NAME	DISCARD TRANSACTION( )
RETURN_CODE	Supported.

Table 59. DISPLAY_LOCAL_LU

DISPLAY_LOCAL_LU	CEMT INQUIRE CONNECTION + CEMT INQUIRE MODENAME + CEMT INQUIRE TRANSACTION
FULLY_QUALIFIED_LU_NAME(vble)	Cannot be specified in CICS. The APPLID on DFHSIT serves as identifier for the local LU. Specific information can be had by identifying the remote LU. Otherwise, the universal ID * can be used.
LU_SESSION_LIMIT(vble)	MAXIMUM on INQ MODENAME.
LU_SESSION_COUNT(vble)	ACTIVE on INQ MODENAME
SECURITY(vble)	Not available.
MAP_NAMES(vble)	Not supported.
REMOTE_LU_NAMES(vble)	INQ CONNECTION(*)
TP_NAMES(vble)	INQ TRANSACTION(*)
BIND_RSP_QUEUE_CAPABILITY(vble)	Not supported.
RETURN_CODE	Supported.

Table 60. DISPLAY_REMOTE_LU

DISPLAY_REMOTE_LU	CEMT INQUIRE CONNECTION + CEMT INQUIRE MODENAME
FULLY_QUALIFIED_LU_NAME(vble)	Cannot be specified; CONNECTION or MODENAME may be used.
LOCALLY_KNOWN_LU_NAME(vble)	CONNECTION name.
UNINTERPRETED_LU_NAME(vble)	NETNAME on INQ CONNECTION.
INITIATE_TYPE(vble)	Not supported.
PARALLEL_SESSION_SUPPORT(vble)	SINGLESESS(Y|N) attribute.
CNOS_SUPPORT(vble)	Always YES.
SECURITY_ACCEPTANCE_LOCAL_LU (vble)	Not available.
SECURITY_ACCEPTANCE_REMOTE_LU (vble)	Not available.
MODE_NAMES(vble)	MODENAME attribute of the SESSIONS resource.
RETURN_CODE	Supported.

Table 61. DISPLAY_MODE

DISPLAY_MODE	CEMT INQUIRE MODENAME + CEMT INQUIRE TERMINAL
FULLY_QUALIFIED_LU_NAME(vble)	Cannot be specified.
MODE_NAME(vble)	MODENAME attribute of the SESSIONS resource.
LOCAL_MAX_SESSION_LIMIT(vble)	AVA on CEMT INQ MODENAME.
CONVERSATION_GROUP_IDS(vble)	Not supported.
SEND_MAX_RU_SIZE_LOWER_BOUND (vble)	Fixed at 8.
SEND_MAX_RU_SIZE_UPPER_BOUND (vble)	Not available.
RECEIVE_MAX_RU_SIZE_LOWER_BOUND (vble)	Fixed at 256.
RECEIVE_MAX_RU_SIZE_UPPER_BOUND (vble)	Not available.
PREFERRED_SEND_RU_SIZE(vble)	Not supported.
PREFERRED_RECEIVE_RU_SIZE(vble)	Not supported.
SINGLE_SESSION_REINITIATION(vble)	Not supported.
SESSION_LEVEL_CRYPTOGRAPHY(vble)	Not available.
SESSION_DEACTIVATED_TP_NAME	Not supported.
CONWINNER_AUTO_ACTIVATE_LIMIT (vble)	Not available.
LU_MODE_SESSION_LIMIT(vble)	MAXIMUM on INQ MODENAME.
MIN_CONWINNERS(vble)	Not supported.
MIN_CONLOSERS(vble)	Not supported.
TERMINATION_COUNT(vble)	Not supported.
DRAIN_LOCAL_LU(vble)	Not supported.
DRAIN_REMOTE_LU(vble)	Not supported.
LU_MODE_SESSION_COUNT(vble)	ACTIVE on INQ MODENAME.
CONWINNERS_SESSION_COUNT(vble)	Not available.
CONLOSERS_SESSION_COUNT(vble)	Not available.
SESSION_IDS(vble)	INQ TERMINAL(*).
RETURN_CODE	Supported.

Table 62. DISPLAY_TP

DISPLAY_TP	CEMT INQUIRE TRANSACTION
TP_NAME(vble)	TRANSACTION(tranid)
STATUS(vble)	ENABLED/DISABLED.
CONVERSATION_TYPE(vble)	CICS TPs allow both types.
SYNC_LEVEL(vble)	CICS TPs allow all sync levels.
SECURITY_REQUIRED(vble)	Not available.
SECURITY_ACCESS(vble)	Not available.

Table 62. DISPLAY_TP (continued)

DISPLAY_TP	CEMT INQUIRE TRANSACTION
PIP(vble)	CICS TPs allow PIP YES and NO.
DATA_MAPPING(vble)	Always NO.
FMH_DATA(vble)	Always YES.
PRIVILEGE(vble)	Not supported.
INSTANCE_LIMIT(vble)	Not supported.
INSTANCE_COUNT(vble)	CEMT INQ TRAN( )
RETURN_CODE	Supported.

Return codes for control operator verbs

When you change the state of a CONNECTION or a MODENAME, the LU services manager starts asynchronously.

Some of the errors that may occur are detected by immediately. Other errors are not detected until a later time, when the LU services manager transaction (CLS1) runs.

If CLS1 detects errors, it causes messages to be written to the CSMT log, as shown in Table 63 on page 124. In normal operation, the CICS master terminal operator may not want to inspect the CSMT log when a command has been issued. So in general, the operator, after issuing a command to change parameters should wait for a few seconds for the request to be carried out and then reissue the INQUIRE version of the command to check that the requested change has been made. In the few cases when an error occurs, the master terminal control operator can refer to the CSMT log.

The message used to report the results of CLS1 execution is DFHZC4900. The explanatory text that accompanies the message varies and is summarized in Table 63 on page 124. In certain cases, DFHZC4901 is also issued to give further information.

Table 63. Messages triggered by CLS1

APPC RETURN CODE	CICS MESSAGE
OK	DFHZC4900 result = SUCCESSFUL
ACTIVATION_FAILURE_RETRY	DFHZC4900 result = VALUES AMENDED + DFHZC4901 MAX = 0
ACTIVATION_FAILURE_NO_RETRY	DFHZC4900 result = VALUES AMENDED + DFHZC4901 MAX = 0
ALLOCATION_ERROR	SYSTEM NOT ACQUIRED is returned to the operator.
COMMAND_RACE_REJECT	DFHZC4900 result = RACE DETECTED
LU_MODE_SESSION_LIMIT_CLOSED	DFHZC4900 result = VALUES AMENDED + DFHZC4901 MAX = 0
LU_MODE_SESSION_LIMIT_EXCEEDED	DFHZC4900 result = VALUES AMENDED + DFHZC4901 MAX = (negotiated value)
LU_MODE_SESSION_LIMIT_NOT_ZERO	DFHZC4900 result = VALUES AMENDED + DFHZC4901 MAX = (negotiated value)
LU_MODE_SESSION_LIMIT_ZERO	DFHZC4900 result = VALUES AMENDED + DFHZC4901 MAX = 0
LU_SESSION_LIMIT_EXCEEDED	DFHZC4900 result = VALUES AMENDED + DFHZC4901 MAX = (negotiated value)

Table 63. Messages triggered by CLS1 (continued)

APPC RETURN CODE	**CICS MESSAGE**
PARAMETER_ERROR	Checked immediately
REQUEST_EXCEEDS_MAX_ALLOWED	Checked immediately
RESOURCE_FAILURE_NO_RETRY	The LU services manager transaction (CLS1) abends with abend code ATNI.
UNRECOGNIZED_MODE_NAME	DFHZC4900 result = MODENAME NOT RECOGNIZED

CICS deviations from APPC architecture

This section describes the way in which the CICS implementation of APPC differs from the architecture described in the *Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2*.

There is one deviation:

- **CICS implementation**: CICS checks incoming BIND requests for valid combinations of the CNOS indicator (BIND RQ byte 24 bit 6) and the PARALLEL-SESSIONS indicator (BIND RQ byte 24 bit 7). If an incorrect combination is found (that is, PARALLEL-SESSIONS specified but CNOS not specified), CICS sends a negative response to the BIND request.

  APPC architecture: The secondary logical unit (SLU), or BIND request receiver, should negotiate the CNOS and PARALLEL-SESSIONS indicators to the supported level and return them in the BIND response. The SLU should not check for an incorrect combination of these indicators.

APPC transaction routing deviations from APPC architecture

A transaction program cannot use ISSUE SIGNAL while in syncfree, syncsend, or syncreceive state. Attempting to do so may result in a state check. This single deviation applies only to APPC transaction routing.

CICS mapping to the APPC verbs

The APPC verbs are implemented by equivalent CICS application programming commands.

The APPC programming language is described in *Transaction Programmer's Reference Manual for LU Type 6.2*.

For information on which APPC option sets are supported by CICS and which are not, or on how CICS implements the APPC control operator verbs, see Appendix A, "CICS mapping to the APPC architecture," on page 115.

Command mapping for APPC basic conversations

The APPC verbs for basic conversations are implemented by equivalent CICS application programming commands.

The following tables show the mapping between APPC verbs and CICS commands for basic conversations. See "Return codes for APPC basic conversations" on page 130 for details of the corresponding return code mapping.

ALLOCATE	**EXEC CICS GDS ALLOCATE** **+ EXEC CICS GDS CONNECT PROCESS**

LU_NAME(vble)
MODE_NAME(vble)
	SYSID on ALLOCATE
	MODENAME on ALLOCATE

ALLOCATE	**EXEC CICS GDS ALLOCATE** **+ EXEC CICS GDS CONNECT PROCESS**
MODE_NAME('SNASVCMG')	MODENAME on ALLOCATE
TPN(vble)	PROCNAME on CONNECT PROCESS (with PROCLENGTH)
TYPE(BASIC_CONVERSATION)	Supported by GDS
TYPE(MAPPED_CONVERSATION)	Not supported
RETURN_CONTROL(WHEN_SESSION_ALLOCATED)	Default on ALLOCATE
RETURN_CONTROL(WHEN_CONWINNER_ALLOCATED)	Not supported
RETURN_CONTROL (WHEN_CONVERSATION_GROUP_ALLOCATED)	Supported
RETURN_CONTROL(IMMEDIATE)	NOQUEUE/NOSUSPEND on ALLOCATE
SYNC_LEVEL	SYNCLEVEL on CONNECT PROCESS 0 – None 1 – Confirm 2 – Syncpoint
SECURITY(NONE)	Not supported
SECURITY(SAME)	Default on ALLOCATE
SECURITY(PGM(USED_ID(vble)	Not supported
(PASSWORD(vble)))	Not supported
PIP(NO)	Supported by PIPLENGTH(0)
PIP(YES(vble1,vble2 ... vblen))	Supported by PIPLIST+PIPLENGTH
RESOURCE	Returned by GDS ASSIGN
RETURN_CODE	Supported

BACKOUT	**EXEC CICS SYNCPOINT ROLLBACK**
RETURN_CODE	Supported

CONFIRM	**EXEC CICS GDS CONFIRM**
RESOURCE	CONVID
RETURN_CODE	Supported
REQUEST_TO_SEND_RECEIVED	Returned in CDBSIG

CONFIRMED	**EXEC CICS GDS ISSUE CONFIRMATION**
RESOURCE	CONVID
RETURN_CODE	Supported

DEALLOCATE	**EXEC CICS GDS SEND LAST** + **EXEC CICS SYNCPOINT** + **EXEC CICS GDS FREE**
TYPE(SYNC_LEVEL) None	EXEC CICS GDS SEND LAST WAIT + EXEC CICS GDS FREE
TYPE(SYNC_LEVEL) Confirm	EXEC CICS GDS SEND LAST CONFIRM + EXEC CICS GDS FREE
TYPE(SYNC_LEVEL) Syncpt	EXEC CICS GDS SEND LAST + EXEC CICS SYNCPOINT + EXEC CICS GDS FREE
TYPE(FLUSH)	EXEC CICS GDS SEND LAST WAIT + EXEC CICS GDS FREE
TYPE(CONFIRM)	EXEC CICS GDS SEND LAST CONFIRM + EXEC CICS GDS FREE
TYPE(ABEND_PROG) Depends on setting of CDBFREE by previous command:	
CDBFREE = X'00	EXEC CICS GDS ISSUE ABEND + EXEC CICS GDS FREE
CDBFREE = X'FF	EXEC CICS GDS FREE
TYPE(ABEND_SVC)	Not supported at API (option set 11)
TYPE(ABEND_TIMER)	Not supported at API (option set 11)
TYPE(LOCAL)	EXEC CICS GDS FREE
LOG_DATA(vble)	Not available at API. CICS inserts the appropriate values
RETURN_CODE	Supported

FLUSH	**EXEC CICS GDS WAIT**

GET_ATTRIBUTES	**EXEC CICS GDS EXTRACT PROCESS** or **EXEC CICS GDS ASSIGN** or **EXEC CICS ASSIGN**
RESOURCE	CONVID
SYNC_LEVEL	SYNCLEVEL on GDS EXTRACT PROCESS 0 – None 1 – Confirm 2 – Syncpoint
UOW_IDENTIFIER	See note
OWN_FULLY_QUALIFIED_LU_NAME	See note

PARTNER_LU_NAME	GDS ASSIGN PRINSYSID
PARTNER_FULLY_QUALIFIED_LU_NAME	See note
MODE_NAME	See note
USERID	ASSIGN USERID
	Note: These values are not normally required in CICS applications and are not available at the API.
RETURN_CODE	Supported

GET_TYPE	**EXEC CICS GDS ASSIGN** (+ return code test)
RESOURCE	PRINCONVID
TYPE(vble)	RETCODE clear = GDS (BASIC) 03 04 = wrong conversation level

POST_ON_RECEIPT	**Not supported**

PREPARE_FOR_SYNCPT	**EXEC CICS GDS ISSUE PREPARE**
RESOURCE	CONVID
RETURN_CODE	Supported

PREPARE_TO_RECEIVE	**EXEC CICS GDS SEND INVITE**
TYPE(SYNC_LEVEL) none	EXEC CICS GDS SEND INVITE WAIT
TYPE(SYNC_LEVEL) confirm	EXEC CICS GDS SEND INVITE CONFIRM
TYPE(SYNC_LEVEL) syncpt	EXEC CICS GDS SEND INVITE + EXEC CICS SYNCPOINT
TYPE(FLUSH)	EXEC CICS GDS SEND INVITE WAIT
TYPE(CONFIRM)	EXEC CICS GDS SEND INVITE CONFIRM
LOCKS(SHORT)	Defaulted
LOCKS(LONG)	Not supported
RETURN_CODE	Supported

RECEIVE_AND_WAIT	**EXEC CICS GDS RECEIVE** (for both LL and BUFFER)
RESOURCE	CONVID field
FILL(BUFFER)	BUFFER option
FILL(LL)	LLID option
LENGTH(vble) Input	MAXFLENGTH option
LENGTH(vble) Output	FLENGTH option
RETURN_CODE	Supported

REQUEST_TO_SEND_RECEIVED	Returned in CDBSIG
DATA	INTO or SET option
WHAT_RECEIVED CONFIRM CONFIRM_DEALLOCATE CONFIRM_SEND DATA DATA_COMPLETE DATA_INCOMPLETE LL_TRUNCATED SEND TAKE_SYNCPT TAKE_SYNCPT_DEALLOCATE TAKE_SYNCPT_SEND	CICS Settings CDBCONF + CDBRECV CDBCONF + CDBFREE CDBCONF FLENGTH field ¬= 0 [+ CDBRECV] CDBCOMPL [+ CDBRECV] ¬CDBCOMPL [+ CDBRECV] RETCODE = X'0310....' ¬CDBRECV CDBSYNC + CDBRECV CDBSYNC + CDBFREE CDBSYNC

Notes:

1. Mapping of RECEIVE_AND_WAIT to EXEC CICS GDS RECEIVE is not always one to one.

   When a CICS RECEIVE command is issued, CICS returns all the information and data (the DATA, the WHAT_RECEIVED flags, and the RETURN_CODE) at once. On completion of a CICS command, more than one indicator may be set, as shown in the WHAT_RECEIVED mapping. It may be necessary to perform more than one subsequent command to honor the actions required by the indicators. For this reason, the action flags must be saved when they are received, and then acted on one by one. If the same data area is used for CONVDATA on successive GDS commands, the flags are overwritten and lost.

   APPC does not work this way; a RECEIVE_AND_WAIT verb returns either data or information about the conversation state (as indicated by WHAT_RECEIVED), but never both.

   It is necessary to program around this difference in philosophy when translating APPC verbs into CICS commands.
2. APPC allows a RECEIVE_AND_WAIT to be issued immediately after an ALLOCATE verb. When you are writing basic conversations in CICS, however, you must supply the PREPARE_TO_RECEIVE explicitly, as follows:

```
ALLOCATE                 EXEC CICS GDS ALLOCATE
                        +EXEC CICS CONNECT PROCESS
(Required by CICS)       EXEC CICS GDS SEND INVITE WAIT
RECEIVE_AND_WAIT         EXEC CICS GDS RECEIVE
```

REQUEST_TO_SEND	**EXEC CICS GDS ISSUE SIGNAL**
RESOURCE	CONVID field
RETURN_CODE	Supported

SEND_DATA	**EXEC CICS GDS SEND**
RESOURCE	CONVID field
DATA	FROM option
LENGTH	FLENGTH option
RETURN_CODE	Supported
REQUEST_TO_SEND_RECEIVED	Returned in CDBSIG
ENCRYPT	Not supported

SEND_ERROR	**EXEC CICS GDS ISSUE ERROR**
RESOURCE	CONVID field

TYPE(PROG)	Default
TYPE(SVC)	Not supported
LOG_DATA	Not supported
RETURN_CODE	Supported
REQUEST_TO_SEND_RECEIVED	Returned in CDBSIG

SYNCPT	**EXEC CICS SYNCPOINT**
`RETURN_CODE`	`Zero - Control returned to program.` `Non-zero - CICS takes action; to backout the UOW (and abend the task or set EIBRLDBK).`

Notes:

1. EXEC CICS SYNCPOINT is not a GDS command.
2. For certain specialized applications, the PREPARE flow (the first flow in syncpoint exchanges) may be sent for a particular conversation by using the command:

   ```
   EXEC CICS GDS ISSUE PREPARE
   ```

   This enables any outstanding messages in the network (for example, SEND ERROR) to be received before proceeding, or deciding not to proceed, with the full syncpoint.

TEST	Check CDB flags
RETURN_CODE	Not supported
TEST(POSTED)	Check CDB flags
TEST(REQUEST_TO_SEND_RECEIVED)	Check CDBSIG

WAIT	Not supported

Return codes for APPC basic conversations

The return codes for APPC basic conversation verbs are indicated by equivalent CICS return codes.

APPC RETURN_CODE	**CICS return codes**
OK	CDBERR and RETCODE are zero
ALLOCATION_ERROR	
Local allocation failures:	CICS is unable to allocate a session for an ALLOCATE command.
ALLOCATION_FAILURE_NO_RETRY	RETCODE = 01.... The second and subsequent bytes give further information

APPC RETURN_CODE	CICS return codes
ALLOCATION_FAILURE_RETRY	For temporary problems, CICS waits in the ALLOCATE command until the problem has cleared and then continues. See also the UNSUCCESSFUL return code, which relates to the NOQUEUE option on the CICS ALLOCATE command.
Remote allocation failures:	These are returned to the program after the CONNECT PROCESS command has been issued, and the partner system has been unable to start the requested task. They may be returned on any subsequent command that relates to the session in use
CONVERSATION_TYPE_MISMATCH	CDBERRCD = 10086034
PIP_NOT_ALLOWED	CDBERRCD = 10086031
PIP_NOT_SPECIFIED_CORRECTLY	CDBERRCD = 10086032
SECURITY_NOT_VALID	CDBERRCD = 080F6051
SYNC_LEVEL_NOT_SUPPORTED_BY_PGM	CDBERRCD = 10086041
SYNC_LEVEL_NOT_SUPPORTED_BY_LU	RETCODE = 030C Note: CICS remembers SYNC_LEVEL negotiated at bind time and does not permit a request to be sent for a sync level not supported by the remote LU.
TPN_NOT_RECOGNIZED	CDBERRCD = 10086021
TRANS_PGM_NOT_AVAIL_NO_RETRY	CDBERRCD = 084C0000
TRANS_PGM_NOT_AVAIL_RETRY	CDBERRCD = 084B6031
BACKED_OUT	CDBERRCD = 08240000
DEALLOCATE_ABEND_PROG	CDBERRCD = 08640000
DEALLOCATE_ABEND_SVC	CDBERRCD = 08640001
DEALLOCATE_ABEND_TIMER	CDBERRCD = 08640002
DEALLOCATE_NORMAL	CDBFREE + ¬CDBERR
PARAMETER_ERROR	RETCODE = 01 0C .. This return code relates ONLY to the ALLOCATE command (for CICS). It is given when an invalid LU name or MODE name has been specified. The third byte gives additional information.
PROG_ERROR_NO_TRUNC	CDBERRCD = 08890000 (RECEIVE Only)
PROG_ERROR_TRUNC	CDBERRCD = 08890001
PROG_ERROR_PURGING	CDBERRCD = 08890000
RESOURCE_FAILURE_RETRY	CDBERRCD = A000
RESOURCE_FAILURE_NO_RETRY	CDBERRCD = A000

APPC RETURN_CODE	CICS return codes

SVC_ERROR_NO_TRUNC
SVC_ERROR_TRUNC
SVC_ERROR_PURGING
	CDBERRCD = 08890100 (RECEIVE Only)
	CDBERRCD = 08890101
	CDBERRCD = 08890100

UNSUCCESSFUL This return code relates ONLY to the APPC ALLOCATE verb with RETURN_CONTROL(IMMEDIATE) specified. This is implemented in CICS with the NOQUEUE option on the ALLOCATE command.	RETCODE = 01 04 04 Control returned to the program because a session was not immediately available.
Note: In all cases, where a value for CDBERRCD is given, CDBERR will be set to X'FF'. It is intended that the program should first test CDBERR and then examine CDBERRCD if additional information is required.	

Command mapping for APPC mapped conversations

The APPC verbs for mapped conversations are implemented by equivalent CICS application programming commands.

See “Return codes for APPC mapped conversations” on page 137 for details of the corresponding return code mapping.

This table has two columns. The headers are in the first row.

MC_ALLOCATE	EXEC CICS ALLOCATE + EXEC CICS CONNECT PROCESS
LU_NAME(vble)	SYSID on ALLOCATE
MODE_NAME(vble)	MODENAME on ALLOCATE
TPN(vble)	PROCNAME on CONNECT PROCESS (with PROCLENGTH)
RETURN_CONTROL(WHEN_SESSION_ALLOCATED)	Default on ALLOCATE
RETURN_CONTROL(WHEN_CONWINNER_ALLOCATED)	Not supported
RETURN_CONTROL (WHEN_CONVERSATION_GROUP_ALLOCATED)	Not supported
RETURN_CONTROL(IMMEDIATE)	NOQUEUE/NOSUSPEND on ALLOCATE
SYNC_LEVEL	SYNCLEVEL on CONNECT PROCESS 0 – None 1 – Confirm 2 – Syncpoint
CONVERSATION_GROUP_ID	Not supported
SECURITY(NONE)	Not supported
SECURITY(SAME)	Default on ALLOCATE
SECURITY(PGM(USED_ID(vble)	Not supported
(PASSWORD(vble)))	Not supported

This table has two columns. The headers are in the first row.

(continued)

MC_ALLOCATE	**EXEC CICS ALLOCATE** **+ EXEC CICS CONNECT PROCESS**
PIP(NO)	Supported by PIPLENGTH(0)
PIP(YES(vble1,vble2 ... vblen))	Supported by PIPLIST+PIPLENGTH
RESOURCE	Returned in CONVID field
RETURN_CODE	Supported

BACKOUT	**EXEC CICS SYNCPOINT ROLLBACK**
RETURN_CODE	Supported

MC_CONFIRM	**EXEC CICS CONFIRM**
RESOURCE	CONVID
RETURN_CODE	Supported
REQUEST_TO_SEND_RECEIVED	Returned in EIBSIG

MC_CONFIRMED	**EXEC CICS ISSUE CONFIRMATION**
RESOURCE	CONVID
RETURN_CODE	Supported

MC_DEALLOCATE	**EXEC CICS SEND LAST** **+ EXEC CICS SYNCPOINT** **+ EXEC CICS FREE**
RESOURCE	CONVID
TYPE(SYNC_LEVEL) None	EXEC CICS SEND LAST WAIT + EXEC CICS FREE
TYPE(SYNC_LEVEL) Confirm	EXEC CICS SEND LAST CONFIRM + EXEC CICS FREE
TYPE(SYNC_LEVEL) Syncpt	EXEC CICS SEND LAST + EXEC CICS SYNCPOINT + EXEC CICS FREE
TYPE(FLUSH)	EXEC CICS SEND LAST WAIT + EXEC CICS FREE
TYPE(CONFIRM)	EXEC CICS SEND LAST CONFIRM + EXEC CICS GDS FREE

`TYPE(ABEND_PROG)`
`   Depends on setting of EIBFREE by`
`   previous command:`	

EIBFREE = X'00	EXEC CICS ISSUE ABEND + EXEC CICS FREE
EIBFREE = X'FF	EXEC CICS FREE
TYPE(LOCAL)	EXEC CICS FREE
RETURN_CODE	Supported

MC_FLUSH	**EXEC CICS WAIT** or **EXEC CICS SEND WAIT**
RESOURCE	CONVID
RETURN_CODE	Supported

MC_GET_ATTRIBUTES	**EXEC CICS EXTRACT PROCESS** or **EXEC CICS ASSIGN**
RESOURCE	CONVID on EXTRACT PROCESS
SYNC_LEVEL	SYNCLEVEL on EXTRACT PROCESS 0 – None 1 – Confirm 2 – Syncpoint
PARTNER_LU_NAME	ASSIGN PRINSYSID
PARTNER_FULLY_QUALIFIED_LU_NAME	See note
MODE_NAME	See note
CONVERSATION_STATE(vble)	STATE on EXTRACT PROCESS
CONVERSATION_CORRELATOR	See note
CONVERSATION_GROUP_ID	Not supported
	Note: These values are not normally required in CICS applications and are not available at the API.
RETURN_CODE	Supported

GET_TYPE	**(Examine EIBRSRCE)**
RESOURCE	EIBRSRCE
TYPE(vble)	EIBRSRCE set - mapped EIBRSRCE not set - not mapped

MC_POST_ON_RECEIPT	**Not supported**

MC_PREPARE_FOR_SYNCPT	**EXEC CICS ISSUE PREPARE**
RESOURCE	CONVID
RETURN_CODE	Supported

MC_PREPARE_TO_RECEIVE	**EXEC CICS SEND INVITE**
TYPE(SYNC_LEVEL) none	EXEC CICS SEND INVITE WAIT
TYPE(SYNC_LEVEL) confirm	EXEC CICS SEND INVITE CONFIRM
TYPE(SYNC_LEVEL) syncpt	EXEC CICS SEND INVITE + EXEC CICS SYNCPOINT
TYPE(FLUSH)	EXEC CICS SEND INVITE WAIT
TYPE(CONFIRM)	EXEC CICS SEND INVITE CONFIRM
LOCKS(SHORT)	Defaulted
LOCKS(LONG)	Not supported
RETURN_CODE	Supported

MC_RECEIVE_AND_WAIT	**EXEC CICS RECEIVE [NOTRUNCATE]**
RESOURCE	CONVID field
LENGTH(vble) Input	MAXFLENGTH option
RETURN_CODE	Supported
REQUEST_TO_SEND_RECEIVED	Returned in EIBSIG
DATA	INTO or SET option
MAP_NAME	Not supported
WHAT_RECEIVED	CICS Settings
CONFIRM	EIBCONF + EIBRECV
CONFIRM_DEALLOCATE	EIBCONF + EIBFREE
CONFIRM_SEND	EIBCONF
DATA_COMPLETE	EIBCOMPL [+ EIBRECV]
DATA_INCOMPLETE	¬EIBCOMPL [+ EIBRECV}
DATA_TRUNCATED	¬EIBCOMPL if NOTRUNCATE not specified on RECEIVE
FMH_DATA_COMPLETE	EIBFMH + EIBCOMPL [+ EIBRECV]
FMH_DATA_INCOMPLETE	EIBFMH + ¬EIBCOMPL [+ EIBRECV]
FMH_DATA_TRUNCATED	EIBFMH + ¬EIBCOMPL [+ EIBRECV] if NOTRUNCATE not specified on RECEIVE
SEND	¬EIBRECV + no other flags
TAKE_SYNCPT	EIBSYNC + EIBRECV
TAKE_SYNCPT_DEALLOCATE	EIBSYNC + EIBFREE
TAKE_SYNCPT_SEND	EIBSYNC

Notes:

1. Mapping of MC_RECEIVE_AND_WAIT to EXEC CICS RECEIVE is not always one to one.

   When a CICS RECEIVE command is issued, CICS returns all the information and data (the DATA, the WHAT_RECEIVED flags, and the RETURN_CODE) at once. On completion of a CICS command, more than one indicator may be set, as shown in the WHAT_RECEIVED mapping. It may be necessary to perform more than one subsequent command to honor the actions required by the indicators. For this reason, the action flags must be saved when they are received (because the EIB can be overwritten by subsequent CICS commands), and then acted on one by one.

   APPC does not work this way; an MC_RECEIVE_AND_WAIT verb returns either data or information about the conversation state (as indicated by WHAT_RECEIVED), but never both.

   It is necessary to program around this difference in philosophy when translating APPC verbs into CICS commands.
2. CICS EIBCOMPL settings are applicable only if NOTRUNCATE is specified on the CICS RECEIVE command.

   If NOTRUNCATE is specified, DATA_INCOMPLETE is indicated by a zero value in EIBCOMPL. CICS will save the remaining data for retrieval by subsequent RECEIVE NOTRUNCATE commands. EIBCOMPL is set when the last part of the data is passed back.

   If the NOTRUNCATE option is not specified, DATA_INCOMPLETE is indicated by the CICS LENGERR condition, and the data remaining after the RECEIVE is discarded.

MC_REQUEST_TO_SEND	**EXEC CICS ISSUE SIGNAL**
RESOURCE	CONVID field
RETURN_CODE	Supported

MC_SEND_DATA	**EXEC CICS SEND**
RESOURCE	CONVID field
DATA	FROM option
LENGTH	LENGTH option
FMH_DATA(NO)	Default
FMH_DATA(YES)	See note
MAP_NAME(NO)	Not supported
MAP_NAME(YES)	Not supported
ENCRYPT(NO)	Not supported
ENCRYPT(YES)	Not supported
RETURN_CODE	Supported
REQUEST_TO_SEND_RECEIVED	Returned in EIBSIG

Note: FMH_DATA(YES) permits the sending of LU6.1 FMHs within an APPC conversation (for example, when running a CICS program which was originally written for use on LU6.1). An LU6.1 FMH may be built either by using the EXEC CICS BUILD ATTACH command, before issuing the EXEC CICS SEND command, or by building the FMH within the program, putting it in the output area, and specifying the FMH option on the SEND command. Either of these two actions is equivalent to specifying FMH_DATA(YES)

MC_SEND_ERROR	**EXEC CICS ISSUE ERROR**
RESOURCE	CONVID field

RETURN_CODE REQUEST_TO_SEND_RECEIVED	Supported Returned in EIBSIG

SYNCPT	**EXEC CICS SYNCPOINT**
RETURN_CODE	Zero - Control returned to program. Non-zero - CICS takes action to backout the UOW (and abend the task or set EIBRLDBK).
Note: For certain specialized applications, the PREPARE flow (the first flow in syncpoint exchanges) may be sent for a particular conversation by using the command: `EXEC CICS ISSUE PREPARE` This enables any outstanding messages in the network (for example, SEND ERROR) to be received before proceeding, or deciding not to proceed, with the full syncpoint.	

MC_TEST	**Check EIB flags**

RESOURCE
TEST(POSTED)
TEST(REQUEST_TO_SEND_RECEIVED)
RETURN_CODE
	EIBRSRCE
	Check EIB flags
	Check EIBSIG
	Not supported

WAIT	**Not supported**

Return codes for APPC mapped conversations
The return codes for APPC mapped conversation verbs are indicated by equivalent CICS return codes.

APPC RETURN_CODE	**CICS return codes**
OK	EIBERR zero + INVREQ not raised
ALLOCATION_ERROR Local allocation failures: ALLOCATION_FAILURE_NO_RETRY ALLOCATION_FAILURE_RETRY	 CICS is unable to allocate a session for an ALLOCATE command. SYSIDERR raised The second and subsequent bytes of EIBRCODE give further information SYSBUSY raised if there is a HANDLE for it. Otherwise, CICS queues the request until a session is available See also the UNSUCCESSFUL return code, which relates to the NOQUEUE option on the CICS ALLOCATE command.

APPC RETURN_CODE	CICS return codes
Remote allocation failures:	These will be returned to the program after the CONNECT PROCESS command has been issued, and the partner system has been unable to start the requested task. They may be returned on any subsequent command that relates to the session in use
CONVERSATION_TYPE_MISMATCH	TERMERR (EIBERRCD = 10086034)
PIP_NOT_ALLOWED	TERMERR (EIBERRCD = 10086031)
PIP_NOT_SPECIFIED_CORRECTLY	TERMERR (EIBERRCD = 10086032)
SECURITY_NOT_VALID	TERMERR (EIBERRCD = 080F6051)
SYNC_LEVEL_NOT_SUPPORTED_BY_PGM	TERMERR (EIBERRCD = 10086041)
SYNC_LEVEL_NOT_SUPPORTED_BY_LU	INVREQ (EIBRCODE = E000000C) Note: CICS remembers SYNC_LEVEL negotiated at bind time and does not permit a request to be sent for a sync level not supported by the remote LU.
TPN_NOT_RECOGNIZED	TERMERR (EIBERRCD = 10086021)
TRANS_PGM_NOT_AVAIL_NO_RETRY	TERMERR (EIBERRCD = 084C0000)
TRANS_PGM_NOT_AVAIL_RETRY	TERMERR (EIBERRCD = 084B6031)
BACKED_OUT	EIBSYNRB (EIBERRCD = 08240000)
DEALLOCATE_ABEND	The transaction is abended with code AZCH (EIBERRCD = 08640000)
DEALLOCATE_NORMAL	EIBFREE + ¬EIBERR
FMH_DATA_NOT_SUPPORTED	TERMERR (EIBERRCD = 08890100)
MAP_EXECUTION_FAILURE MAP_NOT_FOUND MAPPING_NOT_SUPPORTED	Not applicable. Map requests are not sent because the option is not supported.
PARAMETER_ERROR	This return code relates ONLY to the CICS ALLOCATE command. It is given when an invalid LU name or MODE name has been specified.
PARAMETER_ERROR (Invalid LU name)	SYSIDERR (EIBRCODE = D0 04 .. or D0 0C ..)
PARAMETER_ERROR (Invalid mode name)	CBIDERR raised for invalid PROFILE on ALLOCATE command.
PROG_ERROR_NO_TRUNC	EIBERRCD = 08890000 (RECEIVE Only)
PROG_ERROR_PURGING	CDBERRCD = 08890000
RESOURCE_FAILURE_RETRY	EIBERRCD = A000
RESOURCE_FAILURE_NO_RETRY	EIBERRCD = A000
UNSUCCESSFUL	RETCODE = 01 04 04

APPC RETURN_CODE	CICS return codes
This return code relates ONLY to the APPC ALLOCATE verb with RETURN_CONTROL(IMMEDIATE) specified. This is implemented in CICS with the NOQUEUE option on the ALLOCATE command.	Control returned to the program because a session was not immediately available.
Note: In all cases, where a value for EIBERRCD is given, EIBERR will be set to X'FF'. It is intended that the program should first test EIBERR and then examine EIBERRCD if additional information is required.	

CICS deviations from the APPC architecture

CICS deviates from the APPC architecture in a small number of detailed respects.

CICS allows EXEC CICS commands to be issued on APPC conversations when a backout (rollback) is required but the conversation is not in **rollback state** (state 13).

When a session is being allocated, the back-end CICS system checks the incoming bind request for valid combinations of CNOS (change number of sessions) and parallel-sessions indicators. If CICS finds that parallel-sessions is specified but CNOS is not, it sends a negative response to the bind request.

CICS allows a sync level-2 conversation to be terminated using the SEND LAST WAIT or SEND LAST CONFIRM commands. However, doing this is a deviation from the APPC architecture and should be avoided. Figure 48 on page 139 illustrates the problems that can be caused by not syncpointing a sync level-2 conversation.

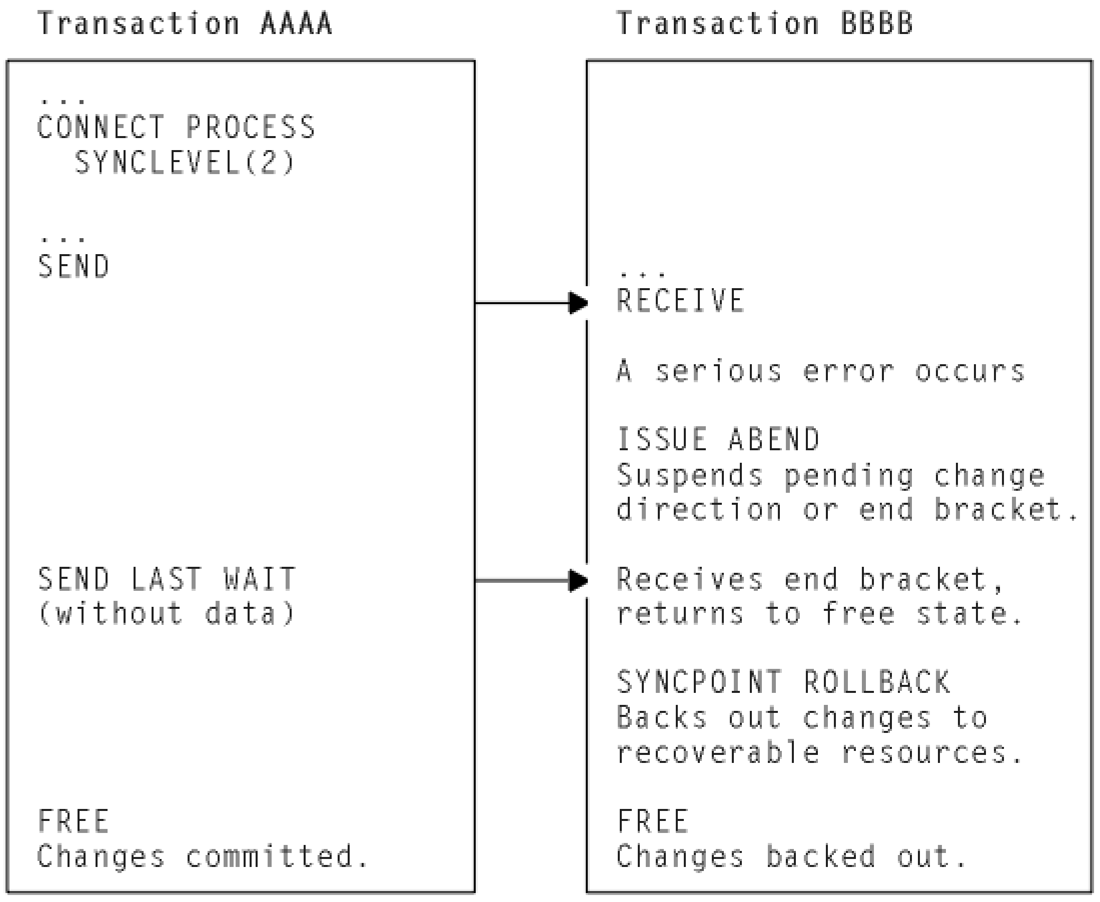

Figure 48. Losing data integrity on a sync level-2 conversation

Because transaction AAAA ends the conversation using the SEND LAST WAIT command, transaction BBBB cannot inform it that an error has occurred. The ISSUE ABEND command causes the backout-required condition to be raised in transaction BBBB; so a SYNCPOINT ROLLBACK is needed. Transaction AAAA commits changes to its resources and data integrity is lost.

The resulting state errors may also lead to the session being unbound.

Effects of CICS deviations on the transaction programmer
Where CICS deviates from the APPC architecture, there may be some effect on transaction programs running on products other than CICS and having conversations with CICS transactions.

The effects can be avoided by using the following programming conventions (the verbs and return codes referred to here are described in *Transaction Programmer's Reference Manual for LU Type 6.2*):

- When writing a transaction program that will converse with a CICS transaction program, do not use the verb PREPARE_TO_RECEIVE with the TYPE(CONFIRM) and LOCKS(LONG) parameters, or with the TYPE(SYNC_LEVEL) and LOCKS(LONG) when the SYNC_LEVEL is CONFIRM. Instead, use the LOCKS(SHORT) parameter to achieve the same function. Use of the LOCKS(LONG) parameter results in less messages being passed on the APPC connection.
- When writing a transaction program that will converse with a CICS transaction program, do not depend on the distinction between the return codes PROG_ERROR_PURGING and PROG_ERROR_NO_TRUNC, and between the return codes SVC_ERROR_PURGING and SVC_ERROR_NO_TRUNC. Instead, the CICS transaction program must be coded to send additional error information after it issues the CICS EXEC ISSUE ERROR in order to describe the reason for sending the error indication.
- When writing a transaction program that will run on CICS, do not depend on the receipt of the sense data X'08890000' or X'08890100' to indicate the state of the other end of the conversation when the partner transaction program sent the error indication. Instead, the partner transaction program must be coded to send additional error information after it sends the error indication in order to describe the reason for sending the error indication.
- Because CICS may omit the negative response before an FMH-7 (ALLOCATION_ERROR), a transaction program in conversation with CICS can receive an ALLOCATION_ERROR **after** the point where the partner transaction appears to have been successfully allocated. The transaction program must therefore be written to handle this possibility.

Appendix B. Migration of LUTYPE6.1 applications to APPC links

If your installation is changing its CICS-to-CICS Intersystem communication (ISC) links from LUTYPE6.1 to APPC (LUTYPE6.2), you may want to redesign some of your existing ISC applications to take advantage of APPC function. Alternatively, you can continue to run your existing applications in *migration mode*.

Migration mode

In migration mode, the front-end and back-end transactions use LUTYPE6.1 commands just as if the session was an LUTYPE6.1 session.

CICS takes data from the transaction in the normal way, and formats it as an APPC mapped data stream for transmission over the link. At the receiving side, CICS analyses the APPC mapped data stream and presents the LUTYPE6.1 data and function management headers to the receiving transaction.

In general, you will not have to modify existing CICS-to-CICS ISC applications to enable them to run in migration mode on APPC links. A notable exception is the use of the ALLOCATE SESSION command. If your installation previously had individually defined ISC sessions, and your application used the ALLOCATE SESSION command to acquire a specific session, you must change this command to ALLOCATE SYSID.

The ISSUE SIGNAL command is valid for both LU types, but the WAIT SIGNAL command is available only for LUTYPE6.1.

Table 64 on page 141 compares the commands that you can use for:

- LUTYPE6.1 applications on LUTYPE6.1 links
- LUTYPE6.1 applications on APPC links (migration mode)
- APPC applications on APPC links.

As Table 64 on page 141 shows, migration mode allows you to start adding new function to an application (for example, using ISSUE ERROR or ISSUE ABEND) without converting it entirely to APPC. You can also implement different sync levels by modifying the application to use the CONNECT PROCESS command. Applications not modified to use CONNECT PROCESS will use sync level 2. The migration of an application towards the “pure” APPC level can thus be made stepwise.

To aid migration, the SESSION and CONVID options can be used interchangeably.

If a migration-mode transaction abends, the architected APPC flows take place. How this affects the connected transaction depends where the abend occurs and is often different from what you would expect if the connection were native LUTYPE6.1.

Because APPC uses different modules from LUTYPE6.1, the user exits XZCIN and XZCOUT are not taken for APPC sessions. Any programs making use of these exits on LUTYPE6.1 will need consideration.

Table 64. Migration of LUTYPE6.1 programs to APPC links

Operation	**Command**	**LU6.1**	**Migration**	**APPC**
Obtain use of a session	ALLOCATE SESSION	yes	no	no
Obtain use of a session	ALLOCATE SYSID	yes	yes	yes
Build an LUTYPE6.1 attach FMH	BUILD ATTACHID	yes	yes	no
Start a partner transaction	SEND	yes(“1” on page 142)	yes(“4” on page 142)	no

Table 64. Migration of LUTYPE6.1 programs to APPC links (continued)

Operation	Command	LU6.1	Migration	APPC
Start a partner transaction	SEND ATTACHID	yes("2" on page 142)	yes("5" on page 142)	no
Start a partner transaction	SEND FMH	yes("3" on page 142)	yes("6" on page 142)	no
Start a partner transaction	CONNECT PROCESS	no	yes("7" on page 142)	yes("7" on page 142)
Retrieve information about how the transaction was initiated	EXTRACT ATTACH	yes	yes	no
	EXTRACT PROCESS	no	yes	yes
Send data	SEND	yes	yes	yes
Send further LUTYPE6.1 FMHs	SEND ATTACHID	yes	yes	no
Send further LUTYPE6.1 FMHs	SEND FMH	yes	yes	no
Receive LUTYPE6.1 FMHs	EXTRACT ATTACH	yes	yes	no
Receive data	RECEIVE	yes	yes	yes
Send and receive data	CONVERSE	yes	yes	yes
Program error	ISSUE ERROR	no	yes	yes
Abend conversation	ISSUE ABEND	no	yes	yes
Request change of direction	ISSUE SIGNAL	yes	yes	yes
Await SIGNAL condition	WAIT SIGNAL	yes	no	no
Synchronize	**Level 0**	no	yes("8" on page 142)	yes
Synchronize	**Level 1** SEND CONFIRM ISSUE CONFIRMATION	no no	yes("8" on page 142) yes	yes yes
Synchronize	**Level 2** SEND CONFIRM ISSUE CONFIRMATION SYNCPOINT SYNCPOINT ROLLBACK	no no yes no	yes("8" on page 142) yes yes yes	yes yes yes yes

Notes on migration of LUTYPE6.1 programs:

1. The CICS transaction identifier is included in the first four bytes of the data. No attach FMH generated.
2. An LUTYPE6.1 attach FMH is generated.
3. An LUTYPE6.1 FMH provided by the application program is sent.
4. An APPC attach FMH is generated, but with no TPN (TPNL=0). The CICS transaction identifier is included in the first four bytes of the data.
5. An APPC attach FMH and an LUTYPE6.1 attach FMH are generated.
6. An APPC attach FMH and an LUTYPE6.1 FMH (provided by the application program) are sent.
7. An APPC attach FMH is generated.
8. Sync levels 0 and 1 can be used if CONNECT PROCESS has been used to define the sync level in operation. If CONNECT PROCESS has not been used, sync level 2 is assumed.

State transitions in LUTYPE6.1 migration-mode conversations

In this section, the state table shows the state transitions that occur when transactions engage in LUTYPE6.1 conversations in migration mode. The state table includes the commands available and the states returned when starting a back-end transaction using the SEND [FMH|ATTACHID] command with the transaction identifier imbedded in first four bytes of user data.

For back-end transactions started by CONNECT PROCESS, use the tables in State transitions in APPC mapped conversations, but remember that the BUILD ATTACH, SEND ATTACHID, SEND FMH, and EXTRACT ATTACH commands are also available.

The commands you can issue, coupled with the EIB flags that can be set after execution, are shown on the first column of the table. The possible conversation states are shown across the top of the table. The states correspond to the columns of the table. The intersection of a row (command and EIB flag) and a column (state) represents the state transition, if any, that occurs when a particular command returning a particular EIB flag is issued in a particular state. A number at an intersection indicates the state number of the next state. Other symbols represent other conditions, as follows:

Symbol	Meaning
N/A	Cannot occur.
×	The EIB flag is any one that has not been covered in earlier rows, or it is irrelevant (but see the note on EIBSIG if you want to use ISSUE SIGNAL).
Abend *code*	The command is not valid in this state. Issuing a command in a state in which it is not valid usually causes an ATCV abend. When a different abend code applies, this is shown in the tables.
=	Remains in current state.
End	End of conversation.

State tables for LUTYPE6.1 migration-mode conversations

Tables showing the state transitions that occur when transactions engage in LUTYPE6.1 migration mode conversations, under the EXEC CICS API.

The `ISSUE SIGNAL` command and the EIBSIG flag

In the tables, the EIBSIG flag is not mentioned. This is because its use is optional and is entirely a matter of agreement between the two conversation partners. In the worst case, it can occur at any time after every command that affects the EIB flags. However, used for the purpose for which it was intended, it usually occurs after a SEND command. Its priority in the order of testing depends on the role you give it in the application.

The EIBSIG flag is set when the partner issues the **`ISSUE SIGNAL`** command.

The `RECEIVE NOTRUNCATE` command

The **`RECEIVE NOTRUNCATE`** command returns a zero value in EIBCOMPL to indicate that the user buffer was too small to contain all the data received from the partner transaction. Normally, you would continue to issue **`RECEIVE NOTRUNCATE`** commands until the last section of data is passed to you, which is indicated by EIBCOMPL = X'FF'. If NOTRUNCATE is not specified, and the data area specified by the RECEIVE command is too small to contain all the data received, CICS truncates the data and sets the LENGERR condition.

State changes for the SYNCPOINT and SYNCPOINT ROLLBACK commands

When the SYNCPOINT and SYNCPOINT ROLLBACK commands are issued, they are propagated on, and affect the state of, all the conversations that are currently active for the task, including MRO conversations.

Following rollback, the conversation can be in **SEND** or **RECEIVE** state, depending on the conversation state at the start of the current distributed unit of work. The conversation can be in **FREE** state if it ended abnormally due to session failure or due to deallocate abend being received, or if the partner transaction issued a SEND LAST WAIT or FREE command.

After a syncpoint or rollback, it is advisable to determine the conversation state before issuing any further commands against the conversation.

State changes following the ISSUE PREPARE command

Although ISSUE PREPARE can return with the conversation in either **SYNCSEND** state, **SYNCRECEIVE** state, or **SYNCFREE** state, the only commands allowed on that conversation following an ISSUE PREPARE are SYNCPOINT and SYNCPOINT ROLLBACK. All other commands Abend.

State tables

Table 65. States 1 - 6

		Command returns	ALLO-CATED	SEND	PEND-RECEIVE	PEND-FREE	RECEIVE	CONF-RECEIVE
Command issued	**EIB flag returned**		**State 1**	**State 2**	**State 3**	**State 4**	**State 5**	**State 6**
BUILD ATTACH	×	Immediately	=	=	=	=	=	=
EXTRACT ATTACH	×	Immediately	INVREQ	INVREQ	INVREQ	INVREQ	=	INVREQ
EXTRACT PROCESS (back-end transaction only)	×	Immediately	Abend	=	=	=	=	=
EXTRACT ATTRIBUTES	×	Immediately	=	=	=	=	=	=
SEND (any valid form)	EIBERR + EIBSYNRB	After error flow detected	Abend	13	13	13	Abend	Abend
SEND (any valid form)	EIBERR + EIBFREE	After error flow detected	12	12	12	12	Abend	Abend
SEND (any valid form)	EIBERR	After error flow detected	Abend	5	5	5	Abend	Abend
SEND INVITE WAIT	×	After data flows	5	5	Abend	Abend	Abend	Abend
SEND INVITE CONFIRM	×	After response from partner	5	5	Abend	Abend	Abend	Abend
SEND INVITE	×	After data buffered	3	3	Abend	Abend	Abend	Abend
SEND LAST WAIT	×	After data flows	12	12	Abend	Abend	Abend	Abend
SEND LAST CONFIRM	×	After response from partner	12	12	Abend	Abend	Abend	Abend
SEND LAST	×	After data buffered	4	4	Abend	Abend	Abend	Abend
SEND WAIT	×	After data flows	2	=	Abend	Abend	Abend	Abend
SEND CONFIRM	×	After response from partner	2	=	5	12	Abend	Abend
SEND	×	After data buffered	2	=	Abend	Abend	Abend	Abend
RECEIVE	EIBERR + EIBSYNRB	After rollback flow detected	Abend	13	13	Abend	13	Abend
RECEIVE	EIBERR + EIBFREE	After error detected	Abend	12	12	Abend	12	Abend
RECEIVE	EIBERR	After error detected	Abend	5	5	Abend	=	Abend
RECEIVE	EIBSYNC + EIBFREE	After sync flow detected	Abend	11	11	Abend	11	Abend
RECEIVE	EIBSYNC + EIBRECV	After sync flow detected	Abend	9	9	Abend	9	Abend

Table 65. States 1 - 6 (continued)

Command issued	EIB flag returned	Command returns	ALLO-CATED	SEND	PEND-RECEIVE	PEND-FREE	RECEIVE	CONF-RECEIVE
			State 1	State 2	State 3	State 4	State 5	State 6
RECEIVE	EIBSYNC	After sync flow detected	Abend	10	10	Abend	10	Abend
RECEIVE	EIBCONF + EIBFREE	After confirm flow detected	Abend	8	8	Abend	8	Abend
RECEIVE	EIBCONF + EIBRECV	After confirm flow detected	Abend	6	6	Abend	6	Abend
RECEIVE	EIBCONF	After confirm flow detected	Abend	7	7	Abend	7	Abend
RECEIVE	EIBFREE	After error flow detected	Abend	12	12	Abend	12	Abend
RECEIVE	EIBRECV	When data available	Abend	5	5	Abend	=	Abend
RECEIVE NOTRUNCATE	EIBCOMPL	When data available	Abend	5	5	Abend	=	Abend
RECEIVE	×	When data available	Abend	=	2	Abend	2	Abend
CONVERSE	As for RECEIVE		As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE
ISSUE CONFIRMATION	×	Immediately	Abend	Abend	Abend	Abend	Abend	5
ISSUE ERROR	EIBFREE	After response from partner	Abend	12	12	Abend	12	12
ISSUE ERROR	×	After response from partner	Abend	=	2	Abend	2	2
ISSUE ABEND	×	Immediately	Abend	12	12	12	12	12
ISSUE SIGNAL	×	Immediately	Abend	=	=	Abend	=	=
ISSUE PREPARE	EIBERR + EIBSYNRB	After response from partner	INVREQ	13	13	13	INVREQ	INVREQ
ISSUE PREPARE	EIBERR + EIBFREE	After error detected	INVREQ	12	12	12	INVREQ	INVREQ
ISSUE PREPARE	EIBERR	After error detected	INVREQ	5	5	5	INVREQ	INVREQ
ISSUE PREPARE	×	After response from partner	INVREQ	10	9	11	INVREQ	INVREQ
SYNCPOINT	EIBRLDBK	After response from partner	=	2 or 5	2 or 5	2 or 5	Abend ASP2	Abend ASP2
SYNCPOINT	×	After response from partner	=	=	5	12	Abend ASP2	Abend ASP2
SYNCPOINT ROLLBACK	×	After rollback across UOW	=	2 or 5	2 or 5	2 or 5	2 or 5	2 or 5
WAIT	×	Immediately	Abend	=	5	12	Abend	Abend
FREE	×	Immediately	End	End	Abend	End	Abend	Abend

Table 66. States 7 - 13

Command issued	EIB flag returned	CONF- SEND	CONF-FREE	SYNC-RECEIVE	SYNC-SEND	SYNC-FREE	FREE	ROLL-BACK
		State 7	State 8	State 9	State 10	State 11	State 12	State 13
BUILD ATTACH	×	=	=	=	=	=	=	=
EXTRACT ATTACH	×	INVREQ	INVREQ	INVREQ	INVREQ	INVREQ	INVREQ	INVREQ
EXTRACT PROCESS (back-end transaction only)	×	=	=	=	=	=	=	=
EXTRACT ATTRIBUTES	×	=	=	=	=	=	=	=

Table 66. States 7 - 13 (continued)

Command issued	**EIB flag returned**	**CONF- SEND**	**CONF-FREE**	**SYNC-RECEIVE**	**SYNC-SEND**	**SYNC-FREE**	**FREE**	**ROLL-BACK**
		State 7	**State 8**	**State 9**	**State 10**	**State 11**	**State 12**	**State 13**
SEND (any valid form)	EIBERR + EIBSYNRB	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND (any valid form)	EIBERR + EIBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND (any valid form)	EIBERR	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND INVITE WAIT	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND INVITE CONFIRM	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND INVITE	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND LAST WAIT	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND LAST CONFIRM	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND LAST	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND WAIT	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND CONFIRM	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
SEND	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBERR + EIBSYNRB	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBERR + EIBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBERR	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBSYNC + EIBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBSYNC + EIBRECV	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBSYNC	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBCONF + EIBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBCONF + EIBRECV	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBCONF	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBFREE	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	EIBRECV	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE NOTRUNCATE	EIBCOMPL	Abend	Abend	Abend	Abend	Abend	Abend	Abend
RECEIVE	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
CONVERSE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE	As for RECEIVE
ISSUE CONFIRMATION	×	2	12	Abend	Abend	Abend	Abend	Abend
ISSUE ERROR	EIBFREE	12	12	12	12	12	Abend	Abend
ISSUE ERROR	×	2	2	2	2	2	Abend	Abend
ISSUE ABEND	×	12	12	12	12	12	Abend	Abend
ISSUE SIGNAL	×	=	=	=	=	=	Abend	Abend
ISSUE PREPARE	EIBERR + EIBSYNRB	INVREQ	INVREQ	INVREQ	INVREQ	INVREQ	INVREQ	INVREQ
ISSUE PREPARE	EIBERR + EIBFREE	INVREQ	INVREQ	INVREQ	INVREQ	INVREQ	INVREQ	INVREQ
ISSUE PREPARE	EIBERR	INVREQ	INVREQ	INVREQ	INVREQ	INVREQ	INVREQ	INVREQ
ISSUE PREPARE	×	INVREQ	INVREQ	INVREQ	INVREQ	INVREQ	INVREQ	INVREQ
SYNCPOINT	EIBRLDBK	Abend	Abend	2 or 5	2 or 5	2 or 5	=	Abend
SYNCPOINT	×	Abend	Abend	2	2	12	=	Abend
SYNCPOINT ROLLBACK	×	2 or 5	2 or 5	2 or 5	2 or 5	2 or 5	=	2 or 5
WAIT	×	Abend	Abend	Abend	Abend	Abend	Abend	Abend
FREE	×	Abend	Abend	Abend	Abend	Abend	End	Abend

Appendix C. Differences between APPC mapped and MRO conversations

When a SEND command is issued on an MRO session, CICS does not defer sending the data, so control indicators cannot be added to the data after a SEND command has been issued.

The same command sequence may therefore require more flows on an MRO session than it does on an APPC session but, if the receiving transaction is correctly designed to be driven by the conversation state, the same effects are achieved.

Different treatment of command sequences

Although you can use similar sequences of commands for a APPC mapped conversations and MRO conversations, there are some cases where the same command sequences operate differently in each conversation type.

Some of the differences between APPC mapped and MRO conversations are shown in the command sequence in Table 67 on page 147.

Table 67. How the same command sequence operates differently in APPC mapped and MRO conversations

Commands	APPC mapped	MRO

`EXEC CICS SEND`
`  CONVID(REM1)`
`  FROM(data1)`
`  LENGTH(251)`
	sending is deferred	`data1` is sent

`EXEC CICS`
`  SYNCPOINT`
	syncpoint request added to `data1`, and both are sent	syncpoint request is sent with null data

`EXEC CICS SEND`
`  CONVID(REM1)`
`  FROM(data2)`
`  LENGTH(251)`
`  INVITE`
	sending of `data2`, with `INVITE`, is deferred	`data2` with `INVITE` is sent

`EXEC CICS WAIT`
`  CONVID(REM1)`
	`data2`, with `INVITE`, is sent	(nothing to send)

`EXEC CICS RECEIVE`
`  CONVID(REM1)`
`        .`
`        .`
`(INVITE received)`		

`EXEC CICS SEND`
`  CONVID(REM1)`
`  FROM(data3)`
`  LENGTH(251)`
`  LAST`
	sending of `data3`, with LAST indicator, is deferred	`data3` is sent, but without LAST indicator

`EXEC CICS`
`  SYNCPOINT`
	syncpoint request and LAST indicator added to `data3` and sent	syncpoint request and LAST indicator are sent with null data

The WAIT option can, of course, be added to the SEND command to cause immediate transmission on APPC links; for example:

```
SEND CONVID(REM1)
     FROM(data2)
     LENGTH(251)
     INVITE
     WAIT
RECEIVE SESSION(REM1)
```

There are no significant differences between the MRO and APPC mapped implementations of this command sequence. However, with MRO, a SEND command with the WAIT option causes CICS to suspend the transaction until the partner system has received the data.

Unlike APPC, MRO allows only one outstanding SEND to be transmitted. This means that when a transaction issue two successive SEND commands (without the WAIT option) to transmit data, the second piece of data does not flow until the partner system has received the first.

A further implementation difference arises between APPC mapped and MRO for command sequences that contain an implicit change of direction. For MRO, a RECEIVE command must not be issued unless the conversation is in **receive state** (state 5).

Using the LAST option

The LAST option on the SEND command indicates the end of the conversation. No further data flows can occur on the session, and the next action must be to free the session.

However, the session can still carry CICS syncpointing flows before it is freed, provided the LAST request has not been flushed.

A syncpoint, whether on an APPC or MRO session, is initiated explicitly by a SYNCPOINT command, or implicitly by a RETURN command. However, the circumstances under which session syncpointing occurs, and the ways in which syncpointing can be avoided on the session, differ for APPC and MRO.

The LAST option and syncpoint flows on APPC sessions

If an APPC mapped conversation has been terminated by a SEND LAST command, without the WAIT option, transmission will have been deferred, and the syncpointing activity causes the final transmission to occur with an added syncpoint request. The conversation is thus automatically involved in the syncpoint.

If the conversation is not to be involved in the syncpoint (for example, because the partner transaction does not access any recoverable resources), the transaction must issue a SEND LAST WAIT command, or a FREE command, to force the transmission before using a command that causes a syncpoint.

The LAST option and syncpoint flows on MRO sessions

If an MRO conversation is terminated by a SEND LAST command, without the WAIT option, the WAIT implicit in all MRO commands is applied, and the data is transmitted. However, in anticipation of subsequent syncpoint flows, CICS does not send the LAST indicator with this data.

If the conversation is not to be involved in the syncpoint (for example, because the partner transaction does not access any recoverable resources) you must specify the WAIT option explicitly on the SEND LAST command to force the LAST indicator to be sent with the data. Alternatively, you could follow the SEND LAST command by a FREE command.

Appendix D. Below the SNA interface

To design high-performance distributed processes, you need some understanding of the SNA protocols and corresponding data flow control (DFC) indicators that CICS uses for DTP, and how the DFC indicators relate to the CICS commands and options. In addition, you need this knowledge to understand the CICS trace.

Except for some commands that can cause transmissions "against the flow" (such as ISSUE SIGNAL), the conversation flow and indicators set are dictated by the transaction currently in **send state** (state 2).

SNA indicators and records

SNA indicators and records can be generated either explicitly as a result of a CICS command, or automatically when CICS detects that they are needed.

The most common SNA indicators and records:

Begin_bracket and conditional_end_bracket
The begin_bracket (BB) and condition_end_bracket (CEB) indicators in the request header (RH) denote respectively the beginning and end of a conversation between two transactions. Because the BB is generated automatically at the start of a conversation, you need only consider the CEB. The CEB is generated by a SEND with the LAST option, an ISSUE ABEND, a FREE command, or task termination before the conversation is ended.

Function management headers
Function management headers (FMHs) are records sent on a conversation which contain SNA control data. Several types of FMH are defined under SNA; but only two (FMH5 and FMH7) are relevant to APPC DTP.

The FMH5, also known as the attach FMH, is sent with BB and contains the information required to initiate the back-end transaction.

The FMH7 is issued by the ISSUE ERROR, ISSUE ABEND, and SYNCPOINT ROLLBACK commands. In addition, if the back-end system rejects the FMH5, an FMH7 is sent to the front-end transaction. The FMH7 contains a 4-byte code, called the sense code, which describes the error. This code is set in EIBERRCD (or CDBERRCD for basic conversations). The FMH7 may be followed by log data. This log data is included in message DFHZN2701 on the sending system and DFHZC3433 on the receiving system.

Change direction
The change direction (CD) indicator, found in the RH, switches the issuing transaction from **send state** (state 2) to **receive state** (state 5). CD is generated explicitly by either of the following:

- A SEND command with the INVITE option
- A CONVERSE command.

PS header (type 10)
PS headers (type 10) are records sent on a conversation which contain syncpoint requests. These headers contain a 2-byte syncpoint request code (for example, *prepare, request commit, committed, and forget*). In addition, the initial record sent contains a 2-byte modifier specifying the conversation state after a successful syncpoint exchange.

Request mode and responses

When data is sent, a response confirming receipt of the data is not normally expected, unless data is sent with the CONFIRM option.

Data is normally sent in RQE (request exception response) mode, meaning that a response is required only if an error condition needs to be transmitted. This response is called -RSP (negative response) and might precede an FMH7. However, if data is sent with the CONFIRM option, the data is sent in RQD (request definite response) mode. This means that the sending transaction will suspend until a DR (definite response) or -RSP is received. The partner transaction generates a DR with the ISSUE CONFIRMATION command.

When SNA indicators are transmitted

To optimize the use of ISC sessions, CICS defers output processing for SEND commands. Deferred output often enables CICS to add SNA indicators to waiting data before transmitting it. The number of transmissions on the session is thereby reduced.

For APPC sessions, this reduction is achieved by accumulating as much data as possible in a CICS buffer before transmitting it across the link. Thus the data from a series of SEND commands is transmitted only when the buffer becomes full or when transmission must be forced (for example, if SEND WAIT is encountered).

Optimization of ISC transmission does not affect the number of data flows that the application programming interface sees.

Notices

Programming interface information

CICS supplies some documentation that can be considered to be Programming Interfaces, and some documentation that cannot be considered to be a Programming Interface.

Programming Interfaces that allow the customer to write programs to obtain the services of CICS Transaction Server for z/OS, Version 5 Release 4 are included in the following sections of the online product documentation:

- Developing applications
- Developing system programs
- Securing overview
- Developing for external interfaces
- Reference: application developmenth
- Reference: system programming
- Reference: connectivity

Information that is NOT intended to be used as a Programming Interface of CICS Transaction Server for z/OS, Version 5 Release 4, but that might be misconstrued as Programming Interfaces, is included in the following sections of the online product documentation:

- Troubleshooting and support
- Reference: diagnostics

If you access the CICS documentation in manuals in PDF format, Programming Interfaces that allow the customer to write programs to obtain the services of CICS Transaction Server for z/OS, Version 5 Release 4 are included in the following manuals:

- Application Programming Guide and Application Programming Reference
- Business Transaction Services
- Customization Guide

- C++ OO Class Libraries
- Debugging Tools Interfaces Reference
- Distributed Transaction Programming Guide
- External Interfaces Guide
- Front End Programming Interface Guide
- IMS Database Control Guide
- Installation Guide
- Security Guide
- Supplied Transactions
- CICSPlex SM Managing Workloads
- CICSPlex SM Managing Resource Usage
- CICSPlex SM Application Programming Guide and Application Programming Reference
- Java Applications in CICS

If you access the CICS documentation in manuals in PDF format, information that is NOT intended to be used as a Programming Interface of CICS Transaction Server for z/OS, Version 5 Release 4, but that might be misconstrued as Programming Interfaces, is included in the following manuals:

- Data Areas
- Diagnosis Reference
- Problem Determination Guide
- CICSPlex SM Problem Determination Guide

Trademarks

Terms and conditions for product documentation

IBM online privacy statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below:

For the CICSPlex® SM Web User Interface (main interface):
: Depending upon the configurations deployed, this Software Offering may use session and persistent cookies that collect each user’s user name and other personally identifiable information for purposes of session management, authentication, enhanced user usability, or other usage tracking or functional purposes. These cookies cannot be disabled.

For the CICSPlex SM Web User Interface (data interface):
: Depending upon the configurations deployed, this Software Offering may use session cookies that collect each user's user name and other personally identifiable information for purposes of session management, authentication, or other usage tracking or functional purposes. These cookies cannot be disabled.

For the CICSPlex SM Web User Interface ("hello world" page):
: Depending upon the configurations deployed, this Software Offering may use session cookies that collect no personally identifiable information. These cookies cannot be disabled.

For CICS Explorer:
: Depending upon the configurations deployed, this Software Offering may use session and persistent preferences that collect each user’s user name and password, for purposes of session management, authentication, and single sign-on configuration. These preferences cannot be disabled, although storing a user's password on disk in encrypted form can only be enabled by the user's explicit action to check a check box during sign-on.

If the configurations deployed for this Software Offering provide you, as customer, the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM Privacy Policy and IBM Online Privacy Statement, the section entitled "Cookies, Web Beacons and Other Technologies" and the IBM Software Products and Software-as-a-Service Privacy Statement.

Index