LU 6.2 API Application Programmer's Reference Manual

HP 3000 MPE/iX Computer Systems

Edition 3

Manufacturing Part Number: 30294-90008
E0692

U.S.A. June 1992

Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability or fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for direct, indirect, special, incidental or consequential damages in connection with the furnishing or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

Restricted Rights Legend

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013. Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19 (c) (1,2).

Acknowledgments

UNIX is a registered trademark of The Open Group.

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Contents

Contents

Contents

Contents

Figures

Figures

Tables

Tables

Preface

This manual serves two purposes: It is a training manual for programmers who wish to create LU 6.2 API applications, and it also serves as a reference manual.

NOTE

The information in this manual can be used to create LU 6.2 API applications on MPE V systems or MPE XL systems. Any differences between LU 6.2 API on MPE V and LU 6.2 API on MPE XL are noted in the manual.

MPE/iX, Multiprogramming Executive with Integrated POSIX, is the latest in a series of forward-compatible operating systems for the HP 3000 line of computers.

In HP documentation and in talking with HP 3000 users, you will encounter references to MPE XL, the direct predecessor of MPE/iX. MPE/iX is a superset of MPE XL. All programs written for MPE XL will run without change under MPE/iX. You can continue to use MPE XL system documentation, although it may not refer to features added to the operating system to support POSIX (for example, hierarchical directories).

Finally, you may encounter references to MPE V, which is the operating system for HP 3000s, not based on the PA-RISC architecture. MPE V software can be run on the PA-RISC (Series 900) HP 3000s in what is known as compatibility mode.

Audience

The audience for this manual is the HP 3000 applications programmer who will participate in writing an LU 6.2 API application. The programmer is assumed to have little or no knowledge of data communications or the IBM environment. For more information in these areas, see the list of related publications at the end of this preface.

Organization

This manual is divided into the following sections and appendices:

Chapter 1, The SNA Network and LU 6.2 API, gives an overview of the SNA environment and Advanced Program-to-Program Communication. It explains the LU 6.2 architecture and Hewlett-Packard's implementation of LU 6.2 API.

Chapter 2, Conversations, describes what a conversation is and the different kinds of conversations that application programs can have. It explains the tasks necessary to establish a conversation.

Chapter 3, Intrinsic Overview, discusses how LU 6.2 API intrinsics are used to implement the conversations described in chapter 2.

Chapter 4, Conversation States, gives an explanation of conversation states and explains the relationship between states and conversations.

Chapter 5, Intrinsic Descriptions, describes each of the intrinsics available with LU 6.2 API.

Chapter 6, Buffer Management, explains how LU 6.2 API manages its send and receive buffers and the impact this has on LU 6.2 API applications.

Chapter 7, Debugging, explains the user trace and the steps for debugging LU 6.2 API applications.

Appendix A, Status Info, explains all the status info values that can be returned by LU 6.2 API intrinsics.

Appendix B, Sample Programs, contains a sample transaction program, in COBOL II and Pascal on the HP 3000 side, and in PL/1 on the IBM side.

Appendix C, State Transition Tables, lists the intrinsics that can be called from each state and the effect of each intrinsic on the state of a conversation.

Appendix D, LU 6.2 Verb Table, maps the LU 6.2 architected verbs to Hewlett-Packard's LU 6.2 intrinsics.

Appendix E, Transact Parameter Masks, describes the parameter masks used in Transact programs on MPE V systems.

Appendix F, Migrating Transaction Programs, provides information on migrating LU 6.2 API applications from MPE V to MPE XL or from earlier versions of LU 6.2 API/XL to the Node Type 2.1 version of LU 6.2 API/XL.

Related HP Publications

- *COBOL II 3000 Reference Manual* (32233-90001)
- *COBOL II 3000/XL Reference Manual* (31500-60001)
- *Pascal 3000 Reference Manual* (32106-90001)
- *HP Pascal Reference Manual* (31502-60005)
- *HP C/XL Reference Manual* (31506-60001)
- *Transact Reference Manual* (32247-60003)
- *MPE XL Languages Migration Guides: Pascal/XL, FORTRAN 77/XL, COBOL II/XL* (31502-60004)
- *MPE V Command Reference Manua*l (32033-90006)
- *MPE XL Commands Reference Manual* (32650-60002)
- *MPE/V Intrinsics Reference Manual* (32033-90007)
- *MPE XL Intrinsics Reference Manual* (32650-60013)
- *APPC Subsystem on MPE V Node Manager's Guide* (30253-90004)
- *APPC Subsystem on MPE XL Node Manager's Guide* (30294-61002)
- *LU 6.2 API/V Node Manager's Guide* (30253-90002)
- *LU 6.2 Base Node Manager's Guide* (30252-90001)
- *SNA Link Services Reference Manual* (30246-90003)
- *SNA Link/XL Node Manager's Guide* (30291-61000)
- *HP SNA Server/Access User's Guide* (30254-61000)
- *Using the Node Management Services Utilities* (32022-61005)
- *HP SNA Products: IBM Host System Programmer's Guides:*

  HP SNA Products: Manager's Guide (5958-8542)

  HP SNA Products: ACF/VTAM and ACF/NCP Guide (5958-8543)

  HP SNA Products: Information Management Subsystem Guide (5958-8545)

  HP SNA Products: CICS Guide (5958-8546)

  HP SNA Products: DISOSS Guide (5958-8547)

  HP SNA Products: AS/400 Guide (5960-1629)

Related IBM Publications

- *Systems Network Architecture Transaction Programmer's Reference Manual for LU Type 6.2* (GC30-3084)
- *An Introduction to Advanced Program-to-Program Communication (APPC)* (GG24-1584)

1 The SNA Network and LU 6.2 API

Systems Network Architecture (SNA)

IBM has established a set of protocols that govern communication between various types of machines and applications. This set of protocols is called **Systems Network Architecture (SNA)**.

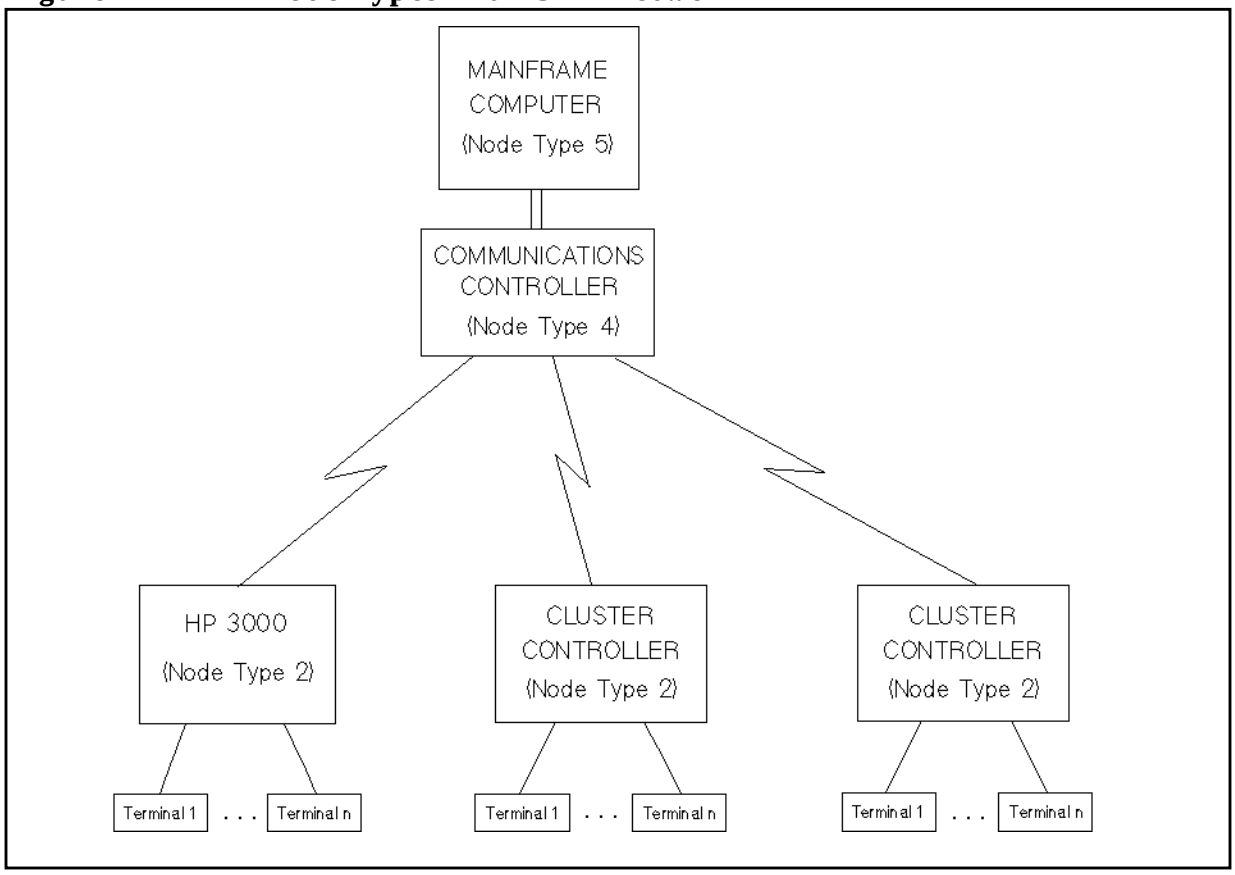
SNA is an architecture designed to be independent of specific software or hardware. In SNA, machines and applications are defined only in terms of the functions they perform. Each machine or node in an SNA network has a **node type**, which is determined by the set of data communications functions it performs. For example, a Type 5 node is a mainframe or host computer. Any machine that performs the data communication functions defined by SNA for a mainframe computer can act as a Type 5 node in an SNA network.

In the past, IBM has focused on centralized data processing, where data is kept in one central location (usually a mainframe computer), and remote processors use the data communication network to access the data. Figure 1-1 shows a typical IBM network, with a centralized mainframe that serves smaller processors in remote locations. An HP 3000 serves as a remote processor in the IBM environment.

At the central location, the mainframe computer and the **communications controller** work as a unit to send and receive data. The communications controller helps the mainframe manage communication with the remote locations so that the mainframe has more power to process and manage data. A communications controller is a Type 4 node.

At each of the remote locations, a **cluster controller** multiplexes several terminals or other peripheral devices to a single data communication line, making efficient use of the line between the cluster controller and the mainframe. A cluster controller is a Type 2 node. The HP 3000 functions as a Type 2 node.

Figure 1-1 Node Types in an SNA Network



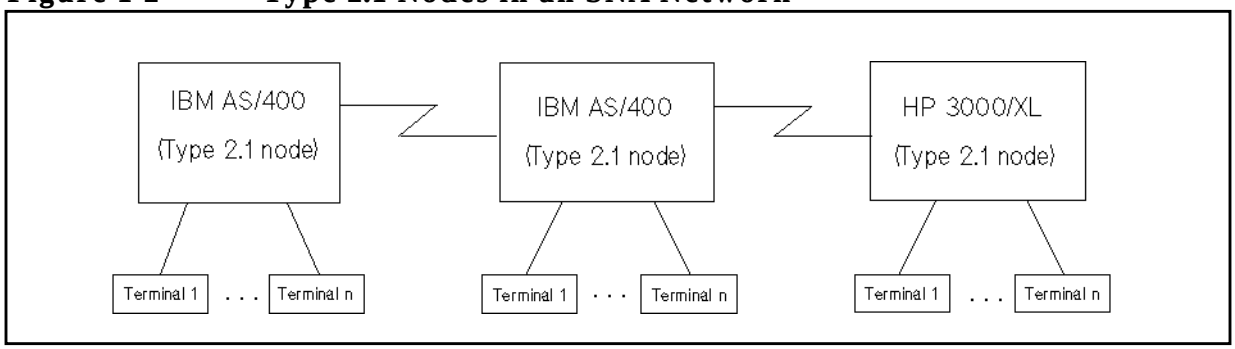
Peer-to-Peer Communication

As computers have become smaller and less expensive, companies have replaced cluster controllers with computers like the HP 3000 that adhere to Node Type 2 protocols. Using an HP 3000 as a cluster controller allows a remote location to have local processing power as well as a connection to the mainframe computer.

A newer version of the Type 2 node allows remote processors to take advantage of their local processing power and perform more complex data communication functions than traditional Type 2 nodes. This newer node type is called **Node Type 2.1**, and the older Node Type 2 is now referred to as **Node Type 2.0**. Type 2.1 nodes, or **peer nodes**, can establish direct connections between themselves without having to rely on a mainframe or a communications controller to manage data traffic. Communication between Type 2.1 nodes is called **peer-to-peer communication**.

IBM AS/400s function as Type 2.1 nodes. HP 3000s running MPE XL can also function as Type 2.1 nodes. Figure 1-2 shows an SNA network without a host node, where two AS/400s and one HP 3000 communicate peer-to-peer as Type 2.1 nodes.

Figure 1-2 Type 2.1 Nodes in an SNA Network



NOTE The APPC subsystem on MPE XL allows the HP 3000 to function as a Type 2.0 or Type 2.1 node. On MPE V, however, the HP 3000 functions only as a Type 2.0 node. For more information on Node Type 2.1, see the *APPC Subsystem on MPE XL Node Manager's Guide*.

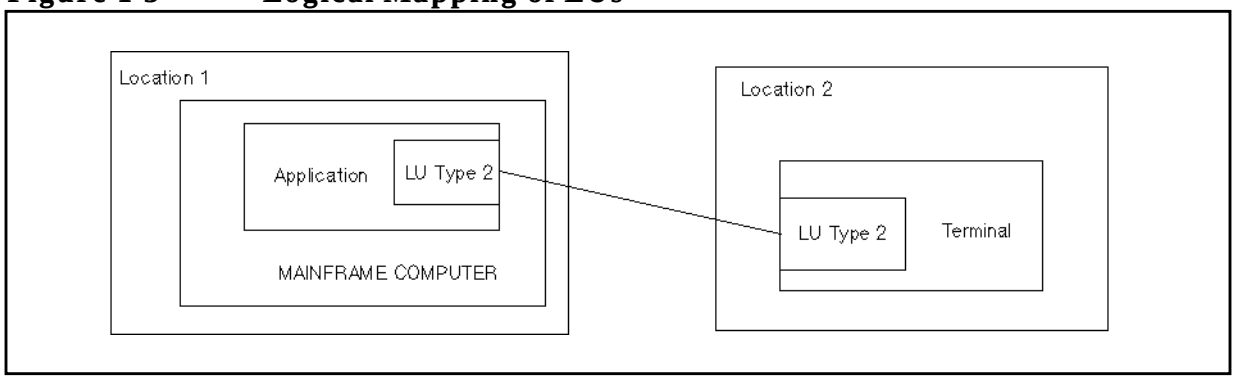
Logical Units (LUs)

A **Logical Unit** (LU) is the set of data communication functions required by an end user in an SNA network. In SNA, an **end user** is defined as the ultimate destination of data. So, an end user can be a peripheral device (like a printer or a terminal) or an application program. A Logical Unit is like a port through which an end user sends and receives data.

Just as different node types define how machines will communicate, different Logical Unit types define how end users will communicate across the network. Since different peripheral devices use different communication protocols and data formats, SNA assigns an **LU type** to each kind of peripheral. For example, an LU Type 2 (LU.T2) describes the communication protocols and data format for any terminal that uses the IBM 3270 data stream.

Logical Units can communicate only with other Logical Units of the same type. An application on the mainframe that will exchange data with an LU.T2 terminal must have its own Type 2 Logical Unit through which to transmit and receive data. Figure 1-3 shows a logical mapping between a terminal at a remote location and an application on the mainframe computer.

Figure 1-3 Logical Mapping of LUs



LU Type 6.2

Logical Unit types are usually used for device emulation. LU.T2 devices or applications emulate IBM 3270 terminals, and LU.T1 and LU.T3 devices or applications emulate printers. Logical Unit Type 6.2 differs from other LU types in that it does not emulate a device. LU 6.2 applications exchange raw data in a format called the **generalized data stream**. LU 6.2 applications can use the generalized data stream to exchange unformatted data, binary data, or data formatted for non-SNA applications and devices.

The LU 6.2 architecture defines two kinds of Type 6.2 LUs: **dependent LUs** and **independent LUs**.

- A **dependent LU** can communicate only with dependent LUs on a Type 5 (host) node. It cannot issue a BIND to initiate a session. When a dependent LU wants a session with the host, it sends an INIT_SELF request and waits for the host to send the BIND. A dependent LU can carry on only one session at a time.
- An **independent LU** can communicate directly with an independent LU on a Type 2.1 (peer) node, like an IBM AS/400. An independent LU can initiate a session by sending a BIND to the remote LU, or it can receive a BIND from the remote LU.

  An independent LU on the HP 3000 can carry on multiple, simultaneous (parallel) sessions with independent LUs on remote systems.Some remote systems, like the IBM AS/400, can perform intermediate routing between nodes in an SNA network. An independent LU connected to one of these systems can take advantage of its routing capabilities to communicate with nodes that are not directly connected to the HP 3000.

NOTE On MPE XL, Type 6.2 LUs may be configured as independent or dependent. On MPE V, all Type 6.2 LUs are dependent LUs.

APPC

Communication between LU 6.2 applications is called **Advanced Program-to-Program Communication (APPC)**. APPC and LU 6.2 allow programs or applications running on different processors to communicate and exchange data.

When data can be processed at both ends of a communication line, fewer data transfers are necessary to perform certain transactions. If a user must run applications on multiple processors to complete a transaction, a single program on the user's local system can automatically start up the remote applications. The user then has to issue only one command to initiate multiple processes on different computers.

The following examples illustrate the advantages of APPC. In the first example, a user at a site without APPC performs a transaction involving applications on the local processor and a remote processor. In the second example, a user performs the same transaction using APPC.

An Example Transaction Without APPC

A clerk works in a satellite office of a large company. The satellite office receives payments from residents in its local area. Payment information is kept on the local processor for up to three months. Any payment information older than three months is kept at the central office on a mainframe computer. The local processor does not have APPC.

A customer comes in who requires payment information for the last six months. The clerk must do the following:

1. Log on to the local processor.
2. Run the appropriate application.
3. Gather payment information for the last three months.
4. End the application.
5. Log on to the mainframe at the central office.
6. Run the appropriate mainframe application.
7. Gather the remainder of the payment information.
8. End the application.
9. Log off the mainframe.

If this is a process that has to be repeated many times during the day, it can become very cumbersome. The clerk must know which information is located on which computer. The clerk must also be able to run two different applications in order to retrieve the information.

An Example Transaction With APPC

The following example is the same transaction described above, but the local computer has APPC capabilities, and an APPC application on the local system automates communication with the central mainframe. Now the clerk must perform the following tasks:

1. Log on to the local processor.
2. Run the appropriate application.
3. Gather all pertinent information, whether it is on the local computer or the mainframe at the central office.
4. End the application.

The clerk has to run only one application on the local computer, because the local APPC application does the following:

1. Logs on to the mainframe.
2. Executes the mainframe application.
3. Gathers the appropriate information.
4. Ends the application.
5. Logs off the mainframe.

If this is a process that is done repeatedly, and part of the information must remain at the central location, then an APPC application saves a great deal of time. Also, the clerk does not have to know where the data resides in order to run the application. The clerk performs the same steps whether the data resides on the local processor or the central mainframe.

Hewlett-Packard's LU 6.2 API

APPC allows two programs to communicate, and LU 6.2 defines the actions or **verbs** that each program may execute in an APPC application.

The Application Program Interface (API) consists of a set of intrinsics. These intrinsics are predefined subroutines that implement the LU 6.2 architected verbs. They can be called from within application programs.

Hewlett-Packard's LU 6.2 API implements the set of **mapped conversation verbs** defined for LU 6.2. SNA defines two types of conversations between communicating programs: **basic conversations** and **mapped conversations**.

In a **basic conversation** (one that uses basic conversation verbs), the application must perform some error recovery and data formatting activities.

In a **mapped conversation** (one that uses mapped conversation verbs), the application focuses only on data handling and relies on an underlying program or system to provide error recovery and data formatting. Hewlett-Packard's LU 6.2 API supports mapped conversations.

Supported Languages

The programming languages supported by LU 6.2 API intrinsics are listed below, for MPE V and MPE XL systems:

MPE V:	MPE XL:
COBOL II	COBOL II
Pascal	Pascal
Transact	Transact
	C

IBM's CICS

Hewlett-Packard's LU 6.2 API intrinsics are often used to write applications that communicate with IBM LU 6.2 applications running under the **Customer Information Control System (CICS)**. CICS is a subsystem that enables data transmitted over communication lines to be processed by host application programs. These host application programs can be written in COBOL, PL/1, or Assembler. Appendix B contains a sample CICS program and an HP 3000 program that communicates with it using LU 6.2 API.

2 Conversations

An LU 6.2 application is called a **transaction program (TP)**. Transaction programs are written in pairs to communicate with each other across a network. Every TP on the HP 3000 will have at least one partner TP on a remote system that is designed to communicate with it. The communication between two TPs is called a **conversation**.

Conversations take place across **sessions**. An APPC session is analogous to the telephone connection that must be established before two people can conduct a conversation across the telephone network. Once a transaction program is running, it can allocate multiple conversations with TPs on remote systems. Each conversation requires one session.

On MPE XL, the APPC subsystem can support a maximum of 256 sessions at the same time. On MPE V, the maximum number of sessions is 8. These sessions must be shared by all the TPs running on the APPC subsystem. Therefore, on MPE XL, one TP can carry on 256 simultaneous conversations, but only if it is the only TP running. Likewise, 256 TPs can be running simultaneously (on MPE XL), but none of them may carry on more than one conversation. The available sessions can be activated, deactivated, and reapportioned among the active TPs as needed. For more information on session limits and session management, see the A*PPC Subsystem on MPE XL Node Manager's Guide*, or the *APPC Subsystem on MPE V Node Manager's Guide*.

Two types of conversations can be conducted between TPs: one-way conversations and two-way conversations. In a one-way conversation, data travels in only one direction, and in a two-way conversation, both sides send and receive data.

Whether a conversation is one-way or two-way, the two sides must remain synchronized for communication to take place. The LU 6.2 architecture allows a TP to ask its partner whether the conversation is synchronized and whether everything is going smoothly on the remote side. One TP sends a confirmation request, and the other TP must respond with a confirmation response.

How confirmation is used in a conversation is up to the TP programmers. It can be used, for example, to verify that a conversation has been allocated properly and that the remote TP is ready to receive data. It can also be used after data is sent, to verify that the remote TP received everything the local TP sent. If a conversation uses confirmation requests and responses, it is called a **conversation with confirm**.

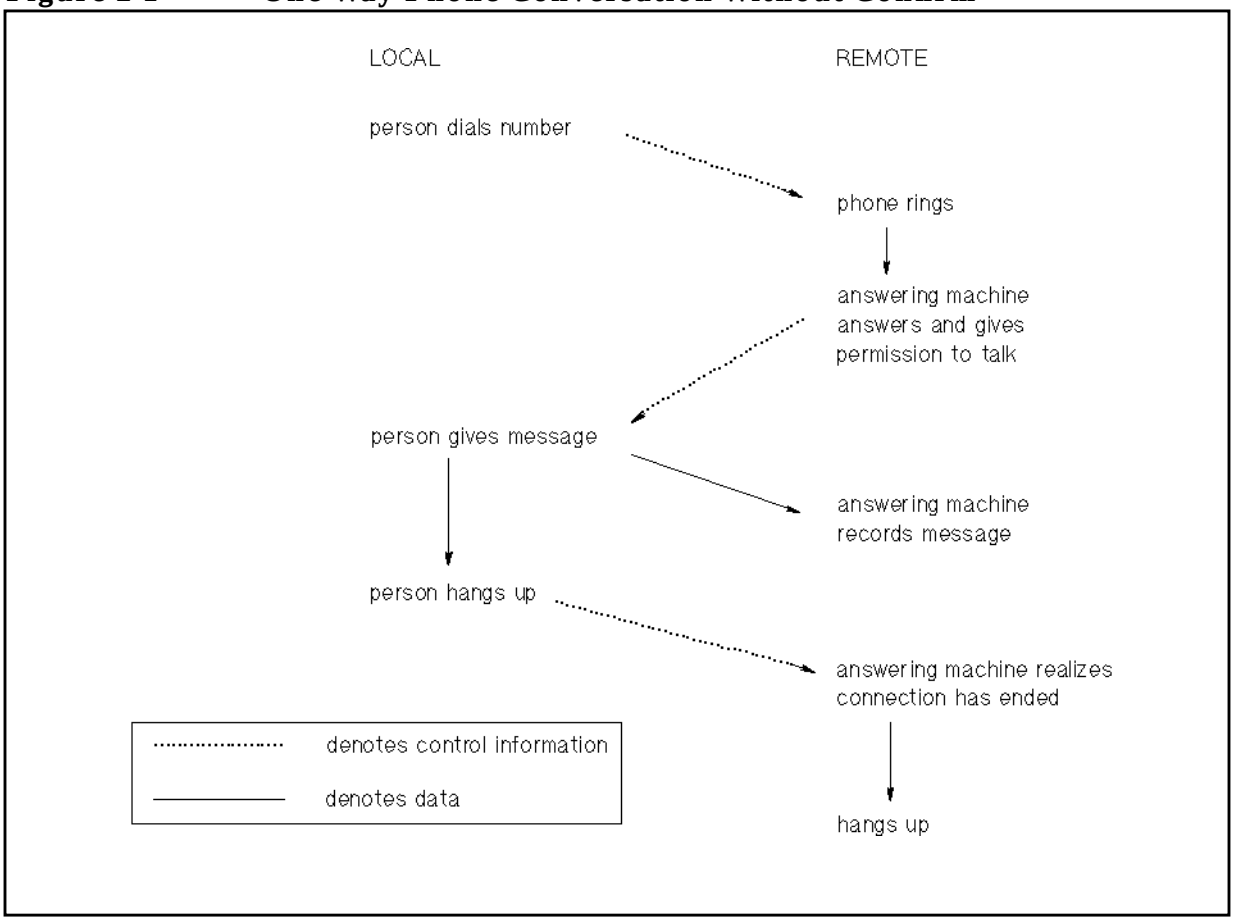
This chapter uses a phone conversation as an analogy to illustrate one-way and two-way conversations with and without confirm.

One-Way Conversation Without Confirm

In a one-way conversation, only one TP sends data. This section describes the simplest type of conversation: a one-way conversation without confirm.

Figure 2-1 is an illustration of a one-way phone conversation without confirm. Notice that data (in this case, the message) is only transmitted in one direction. All other transmitted information is **control information**, used by the two sides to mark the beginning and ending of the conversation and to establish whose turn it is to talk.

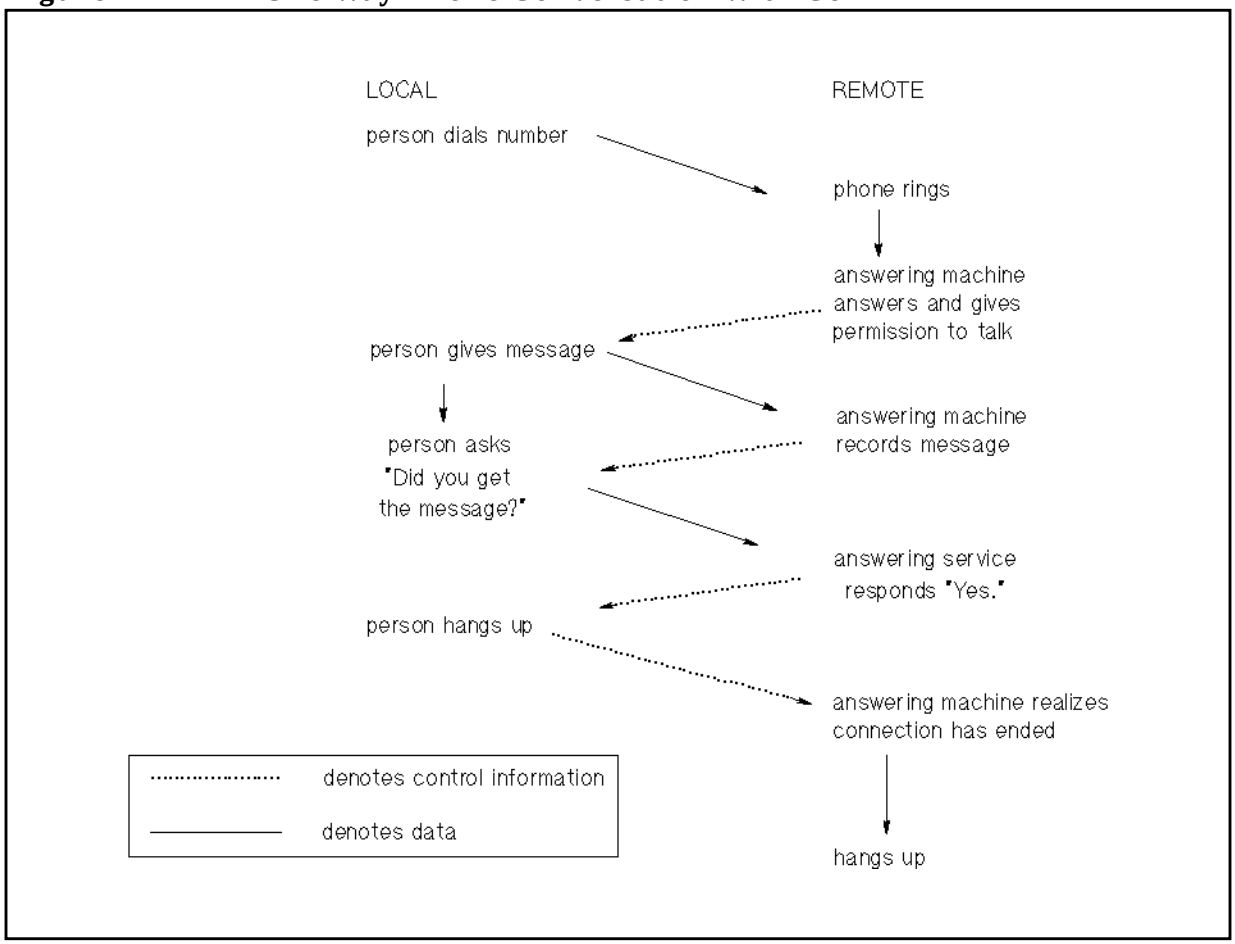
Figure 2-1 One-Way Phone Conversation Without Confirm



One-Way Conversation With Confirm

In a one-way conversation with confirm, the side that sends the data requests confirmation from the receiving side that the conversation is synchronized. Figure 2-2 illustrates a one-way conversation in which the local TP asks the remote TP for confirmation that it received all the data the local TP sent.

Figure 2-2 One-Way Phone Conversation With Confirm



In a one-way conversation with confirm, the caller does not hang up until it has received confirmation that the conversation is synchronized. Once synchronization has been confirmed, each side ends the conversation independently of the other.

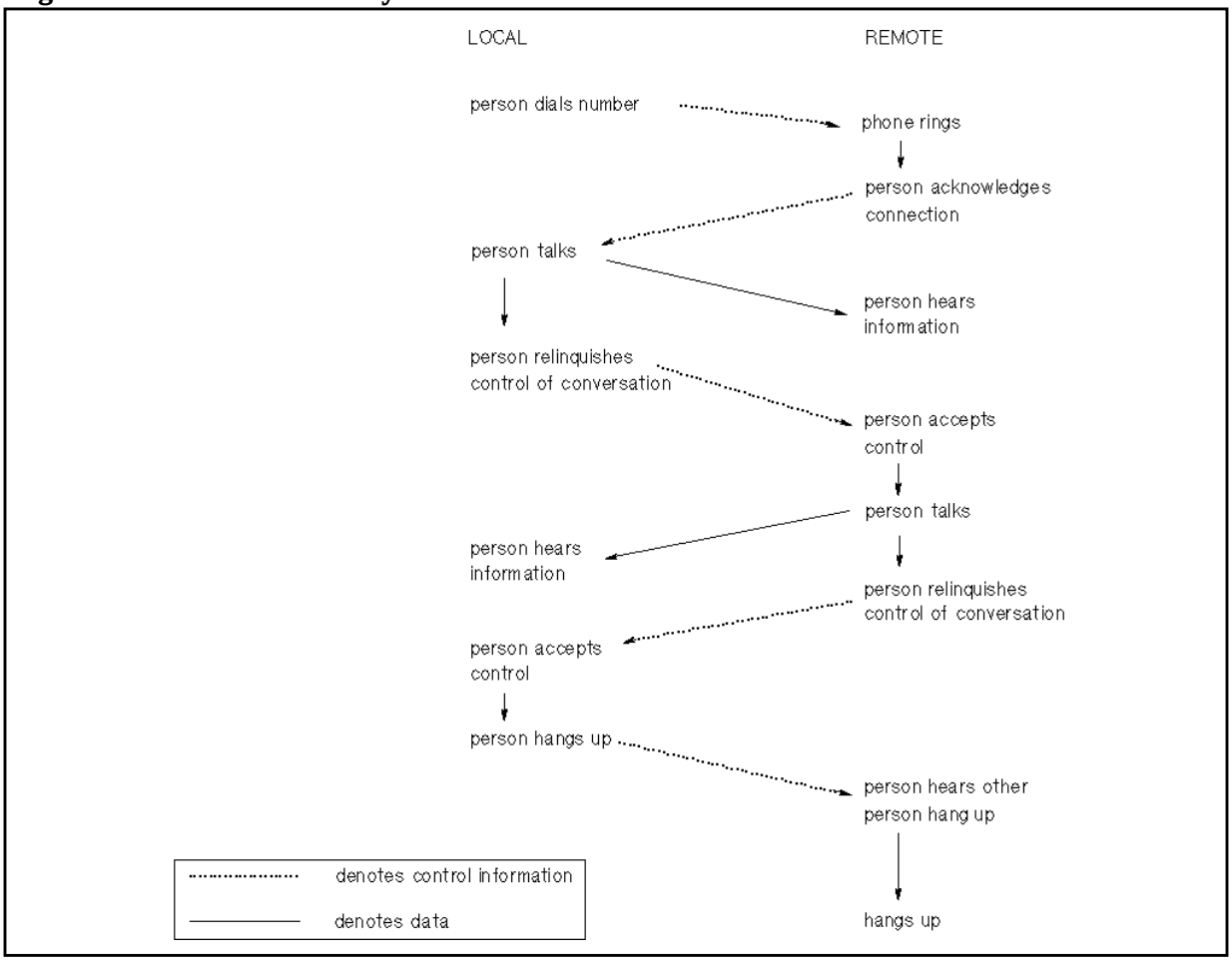
Two-Way Conversation Without Confirm

In a two-way conversation, both sides send and receive data. Because the two sides must take turns talking, they use the following protocol:

1. Only one person can talk at a time.
2. The person that initiated the conversation talks first after communication is established.
3. The person talking must relinquish control before the other person can speak.
4. The person with permission to speak is the only one who can end the conversation.

Figure 2-3 is an illustration of a two-way conversation without confirm. Because a two-way conversation is more complex than a one-way conversation, more control information is needed to manage it.

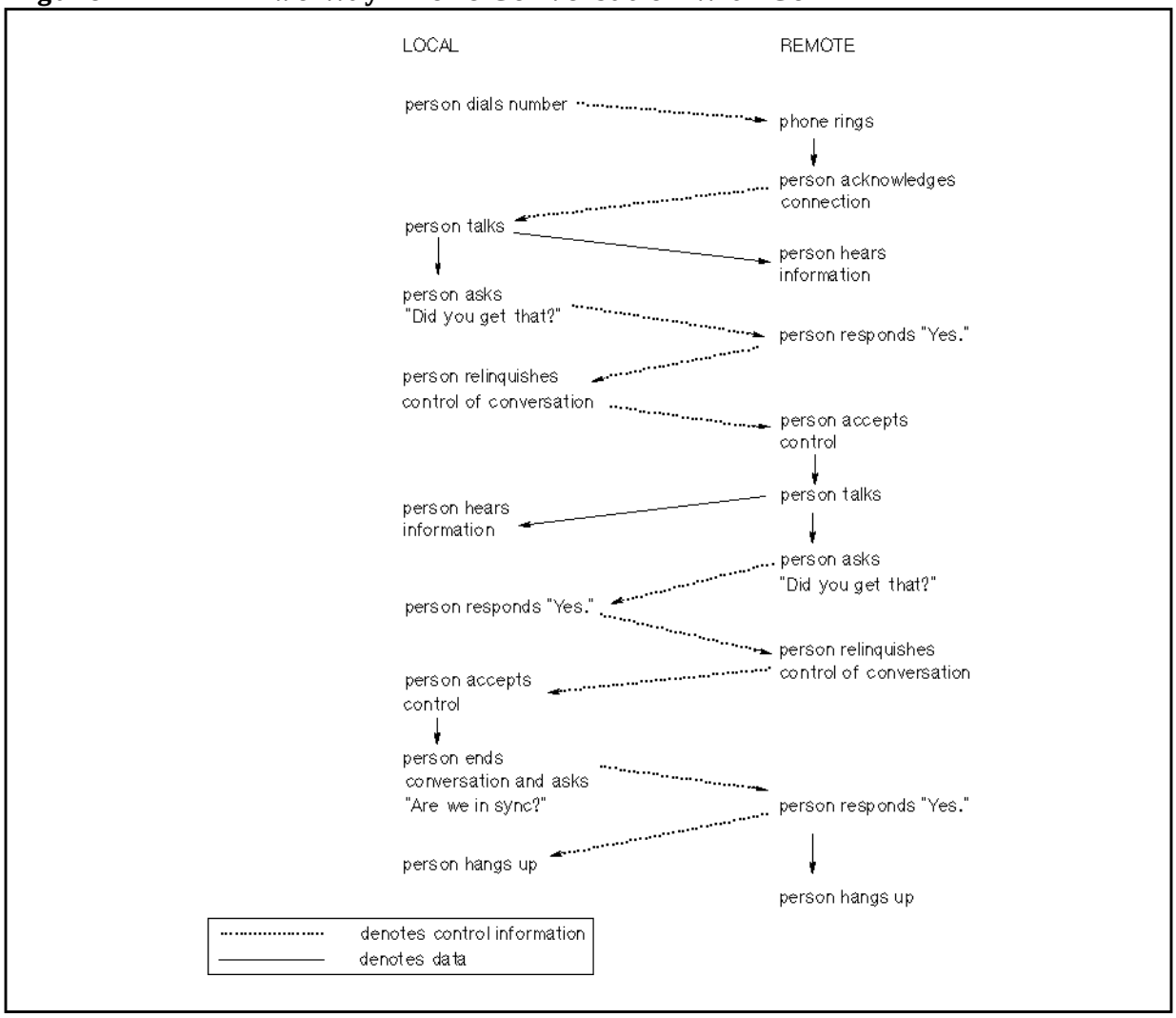
Figure 2-3 Two-Way Phone Conversation Without Confirm



Two-Way Conversation With Confirm

In a two-way conversation with confirm, the sending side issues a confirmation request and must wait for confirmation from the receiving side before transmitting any more information. Figure 2-4 is an illustration of a two-way conversation with confirm.

Figure 2-4 Two-Way Phone Conversation With Confirm



Confirmation ensures that the two sides of the conversation remain synchronized, but it increases the amount of control information and the number of transmissions necessary to transmit data.

Establishing Conversations

TP conversations can be locally initiated (initiated by the TP on the HP 3000), or remotely initiated (initiated by the TP on the remote system). This section describes the tasks that the local application programmer, the node manager, and the remote application programmer must perform in order to establish a locally or remotely initiated conversation.

Locally Initiated Conversations

The following things must occur for a local TP to initiate a conversation:

1. An APPC session of the appropriate session type must be established.
2. A local end user must run the local TP.
3. The local TP must send an allocate request over the session assigned to it, to request a conversation with the remote TP.
4. The remote TP must be coded to receive the allocate request from the local TP.

Note that the session must be established before the local TP can use it to send the allocate request. This section describes the tasks that the local application programmer and the node manager must perform in order to establish a locally initiated conversation.

Local Programmer Tasks

To prepare for a locally initiated conversation, you must do the following:

1. Work with the programmer on the remote system to design and code the TP.
2. Ask the node manager for the name of an appropriately configured session type, or ask the node manager to configure a session type for the conversation. The session type must direct data to the remote LU that serves the remote TP.
3. Code the name of the session type into the *SessionType* parameter of the `MCAllocate` intrinsic.

Node Manager Tasks

To prepare for a locally initiated conversation, the node manager must do the following:

1. Configure an appropriate session type. For information on configuring session types, see the *APPC Subsystem on MPE XL Node Manager's Guide* or the *LU 6.2 API/V Node Manager's Guide*.
2. Tell the application programmer the name of the session type. The programmer must code the name of the session type into the local TP.
3. Activate a session of the appropriate session type, or configure one for automatic activation at subsystem startup. For more information on session activation, see the *APPC Subsystem on MPE XL Node Manager's Guide* or the *APPC Subsystem on MPE V Node Manager's Guide*.

Remotely Initiated Conversations on MPE V

The following things must occur for a remote TP to initiate a conversation on MPE V:

1. An APPC session of the appropriate session type must be established.
2. The remote TP must issue an allocate request over that session, specifying the name of the job that runs the local TP with which it wants a conversation.
3. The APPC subsystem must receive the allocate request and stream the job that runs the local TP.
4. The local TP must be coded to receive the allocate request from the remote TP.

Local Programmer Tasks

To prepare for remotely initiated conversations, you must do the following:

1. Work with the programmer on the remote system to design and code the TP. The local TP must call the `MCGetAllocate` intrinsic to receive an allocate request from the remote TP.
2. Ask the node manager for the name of an appropriately configured session type, or ask the node manager to configure a session type for the conversation. The session type must direct data to the remote LU that serves the remote TP.
3. Code the name of the session type into the *`SessionType`* parameter of the `MCGetAllocate` intrinsic.
4. Tell the node manager the executable file name of the TP. The executable file may reside in any group and account. The node manager will create a job to run the TP.

5. Ask the node manager for the name of the job file that runs the local TP, and tell the programmer on the remote system what the job file name is. The remote system must send the name of the job file in the allocate request.

Node Manager Tasks

To prepare for a remotely initiated conversation on MPE V, the node manager must do the following:

1. Configure an appropriate session type. See the *LU 6.2 API/V Node Manager's Guide* for information on session type configuration.
2. Tell the application programmer the name of the session type. The programmer must code the name of the session type into the local TP.
3. Activate a session of the appropriate session type, or configure one for automatic activation at subsystem startup.
4. Create a job that runs the executable TP file. The job file must be located in the APPC.SYS group and account. See the *LU 6.2 API/V Node Manager's Guide* for more information.

Remote Programmer Tasks

To prepare a remote program to initiate a conversation with an HP TP on MPE V, the remote programmer must do the following:

1. Design and code the program to initiate a conversation with the corresponding TP on the HP 3000.
2. Make sure that the remote TP passes the proper job name in the allocate request. The HP application programmer must tell the remote TP programmer which job name to use.

Remotely Initiated Conversations on MPE XL

The following things must occur for a remote TP to initiate a conversation with a local TP on MPE XL:

1. An APPC session of the appropriate session type must be established.
2. The remote TP must issue an allocate request over that session, specifying the name of the local TP with which it wants a conversation.
3. The APPC subsystem must receive the allocate request, look up the local TP name in the APPC subsystem configuration file, and determine from the configuration file what to do with the allocate request.

a. If the local TP is configured to conduct multiple remotely initiated conversations, and if it is already active and in conversation, the APPC subsystem must queue the allocate request until the local TP finishes the current conversation and calls the `MCGetAllocate` intrinsic again. If the local TP is not currently running, the APPC subsystem must stream the job that runs the local TP.

b. If the local TP is configured to conduct only one remotely initiated conversation, the APPC subsystem must stream the job that runs the local TP.

4. The local TP must be coded to receive the allocate request from the remote TP.

NOTE On MPE XL, any local TPs that will conduct remotely initiated conversations must be configured through NMMGR/XL. See the *APPC Subsystem on MPE XL Node Manager's Guide* for information on TP configuration.

Local Programmer Tasks

To prepare for remotely initiated conversations on MPE XL, you must do the following:

1. Work with the programmer on the remote system to design and code the TP. The local TP can be designed to receive multiple allocate requests from the remote TP, or it can be designed to receive only one allocate request. A TP designed to receive multiple allocate requests must call the `MCGetAllocate` intrinsic multiple times. A TP designed to receive only one allocate request must call the `MCGetAllocate` intrinsic only once.
2. Together with the programmer on the remote system, decide on a name for the TP. Make sure you and the other programmer agree on the TP name; it must be coded into the local TP, and it must be sent by the remote system in the allocate request. Tell the node manager the TP name. The node manager must configure the TP name in the APPC subsystem configuration file.
3. Code the TP name into the *`LocalTPName`* parameter of the `MCGetAllocate` intrinsic and the *`LocalTPName`* parameter of the `TPStarted` intrinsic.
4. Ask the node manager for the name of an appropriately configured session type, or ask the node manager to configure a session type for the conversation. The session type must direct data to the remote LU that serves the remote TP.
5. Code the name of the session type into the *`SessionType`* parameter of the `MCGetAllocate` intrinsic.

6. Tell the node manager the executable file name of the TP. The executable file may reside in any group and account. The node manager will create a job to run the TP.
7. Tell the node manager whether the TP calls the `MCGetAllocate` intrinsic multiple times or only once. The node manager must configure the TP to accept either single or queued allocate requests.
8. Decide whether you want to start the TP yourself or have the APPC subsystem start the TP. While you are debugging the TP, you might want to start it yourself. However, once the TP is working, you should have the APPC subsystem start the TP when it receives an allocate request from the remote TP. Tell the node manager whether to configure the TP for manual or automatic startup.
9. Tell the node manager how long the local TP should wait for an allocate request from the remote TP before the `MCGetAllocate` intrinsic times out. The node manager will configure the time-out value.

Node Manager Tasks

To prepare for a remotely initiated conversation on MPE XL, the node manager must do the following:

1. Configure an appropriate session type.
2. Tell the application programmer the name of the session type. The programmer must code the name of the session type into the local TP.
3. Activate a session of the appropriate session type, or configure one for automatic activation at subsystem startup.
4. Create a job that runs the executable TP file. See the APPC Subsystem on *MPE XL Node Manager's Guide* for more information.
5. Configure the TP name, the job name, the time-out value, whether the TP accepts queued allocate requests, and whether the programmer will start the TP manually. See the *APPC Subsystem on MPE XL Node Manager's Guide* for information on TP configuration.

Remote Programmer Tasks

To prepare a remote program to initiate a conversation with an HP TP on MPE XL, the remote programmer must do the following:

1. Design and code the program to initiate a conversation with the corresponding TP on the HP 3000.
2. Make sure that the remote TP passes the proper TP name in the allocate request. The HP application programmer must tell the remote TP programmer which TP name to use.

3 Using Intrinsics

Chapter 2 , “Conversations,” introduced the types of conversations in which transaction programs can participate. This chapter explains how LU 6.2 API intrinsics are used to create the conversations described in Chapter 2 , “Conversations.”

Table 3-1 lists the intrinsics that will be discussed in this chapter. The list is a subset of the LU 6.2 API intrinsics. Chapter 5 , “Intrinsic Descriptions,” contains a complete description of all of the intrinsics.

Table 3-1 Intrinsics Used in Example Conversations

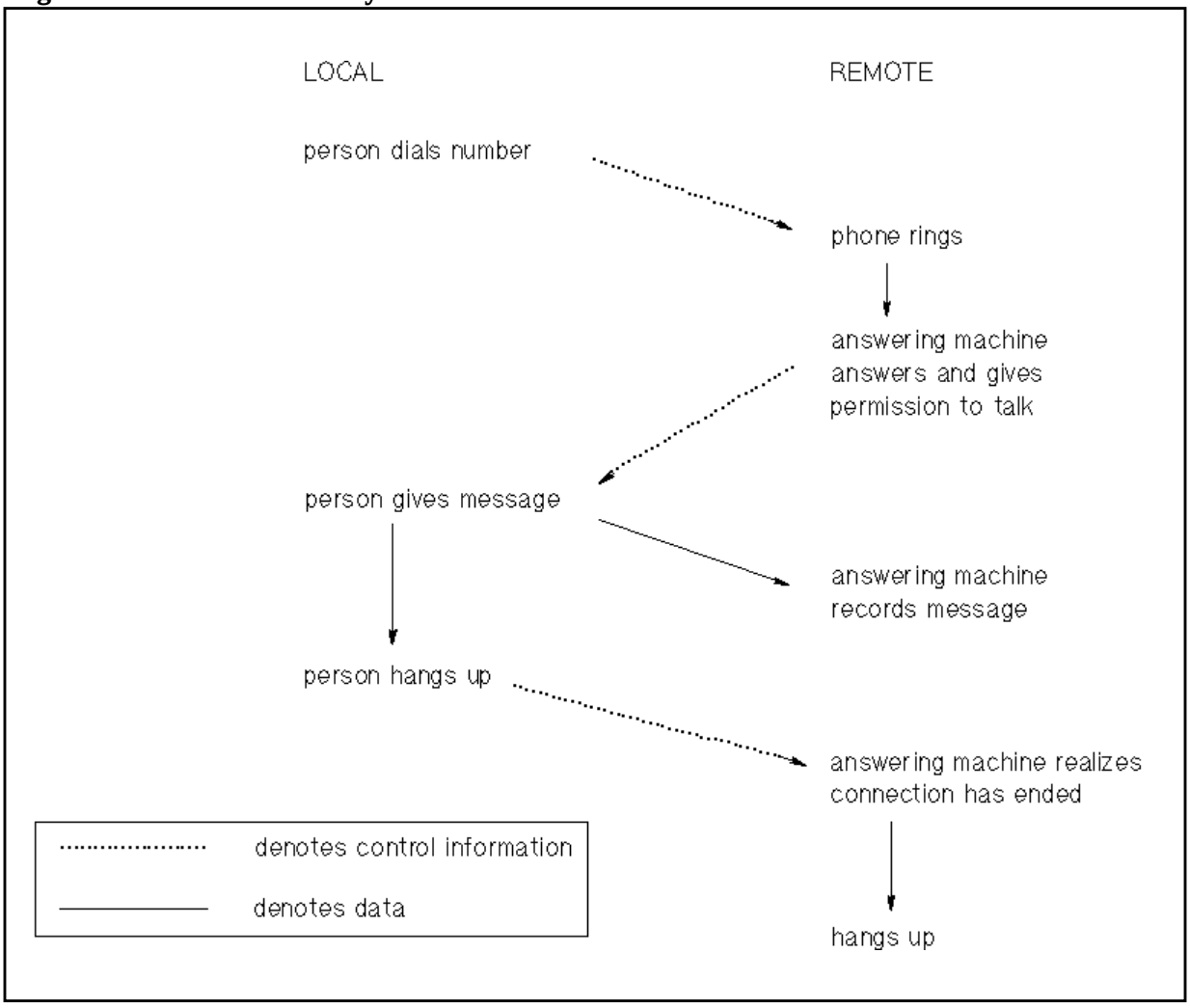
Intrinsic	Definition
`MCAllocate`	Establishes a mapped conversation between TPs.
`MCConfirm`	Sends a confirmation request to the remote TP and waits for a reply.
`MCConfirmed`	Sends a confirmation response to a remote TP that has issued a confirmation request.
`MCDeallocate`	Ends a mapped conversation between TPs.
`MCPrepToRcv`	Informs the remote TP that the local TP is now ready to receive data.
`MCRcvAndWait`	Waits for information to arrive on the mapped conversation and then receives the information. The information can be data, conversation status, or request for confirmation.
`MCSendData`	Sends data to the remote TP.

One-Way Conversation Without Confirm

In this section, the one-way conversation introduced in Chapter 2 , “Conversations,” is described in terms of the intrinsics you would use to create it.

Figure 3-1, the one-way phone conversation without confirm, is reproduced below. In the following sections, the intrinsics used to create this conversation will be discussed and added to the illustration.

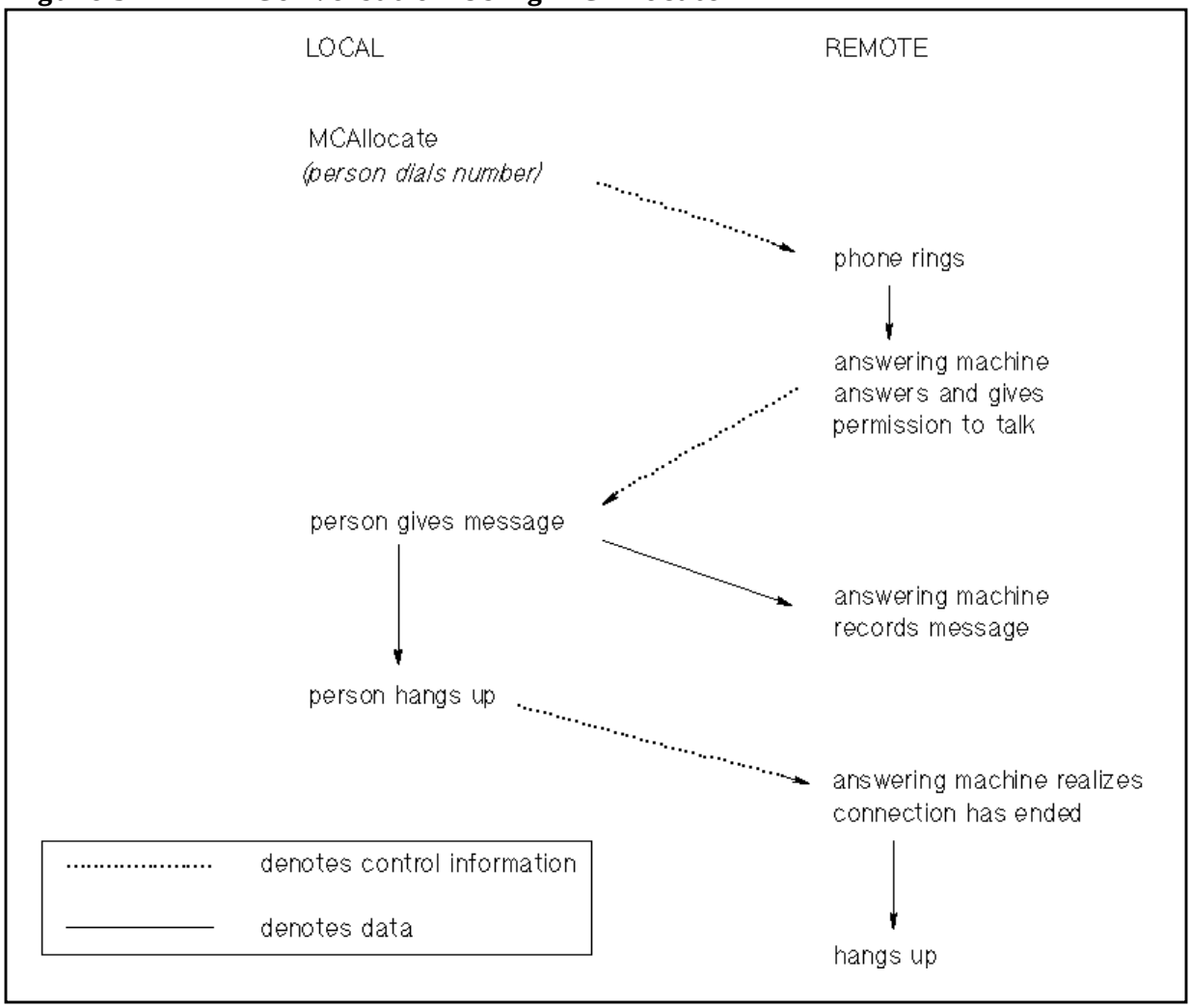
Figure 3-1 One-Way Phone Conversation Without Confirm



MCAllocate

MCAllocate allocates network resources and establishes a conversation with the remote TP. After the MCAllocate intrinsic has executed, the local TP is ready to send data, and the remote TP is ready to receive it. The MCAllocate intrinsic replaces "person dials number," as shown in Figure 3-2.

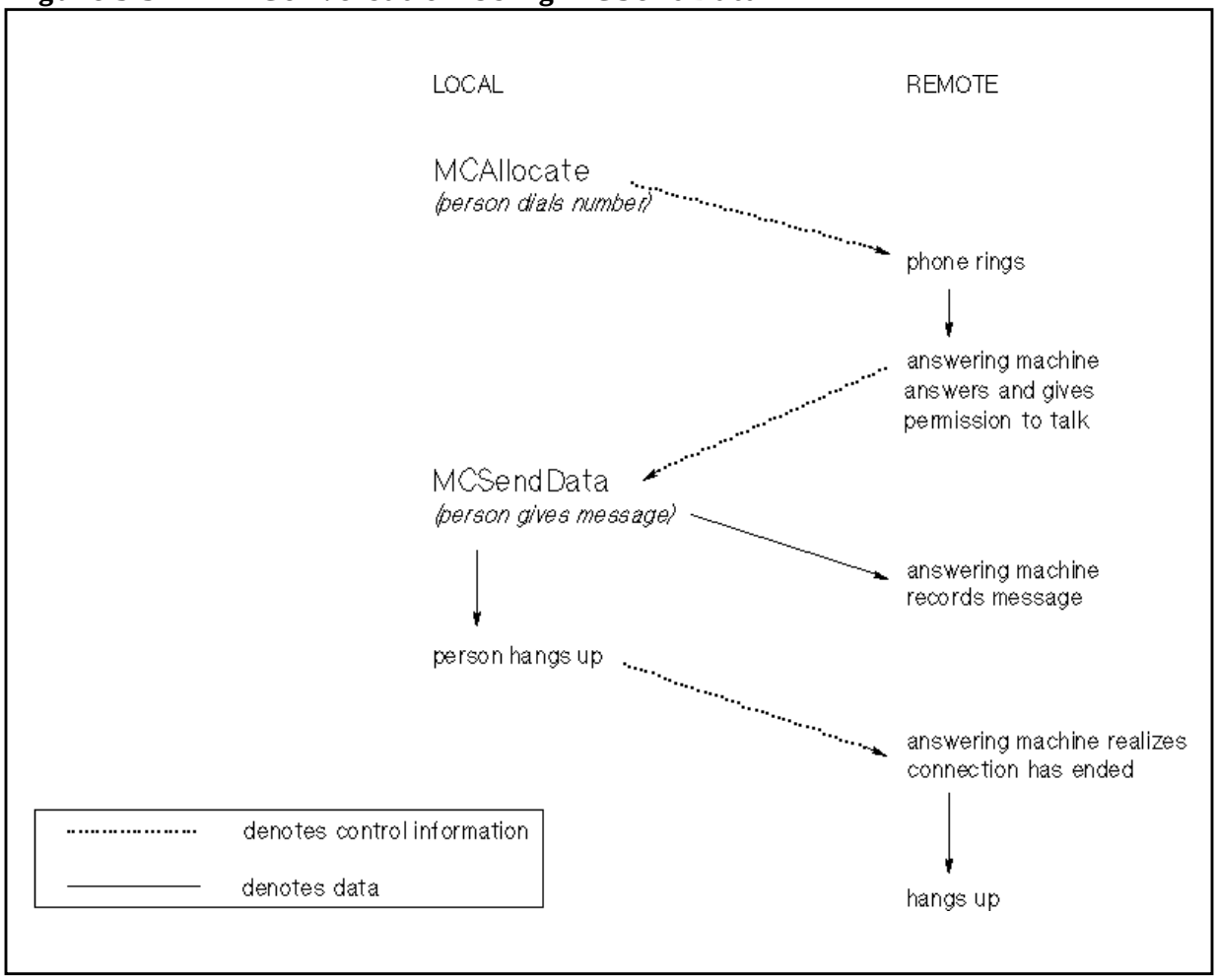
Figure 3-2 Conversation Using MCAllocate



MCSendData

The `MCSendData` intrinsic is the vehicle used by the local TP to send information to the remote TP. The `MCSendData` intrinsic replaces "person gives message," as shown in Figure 3-3.

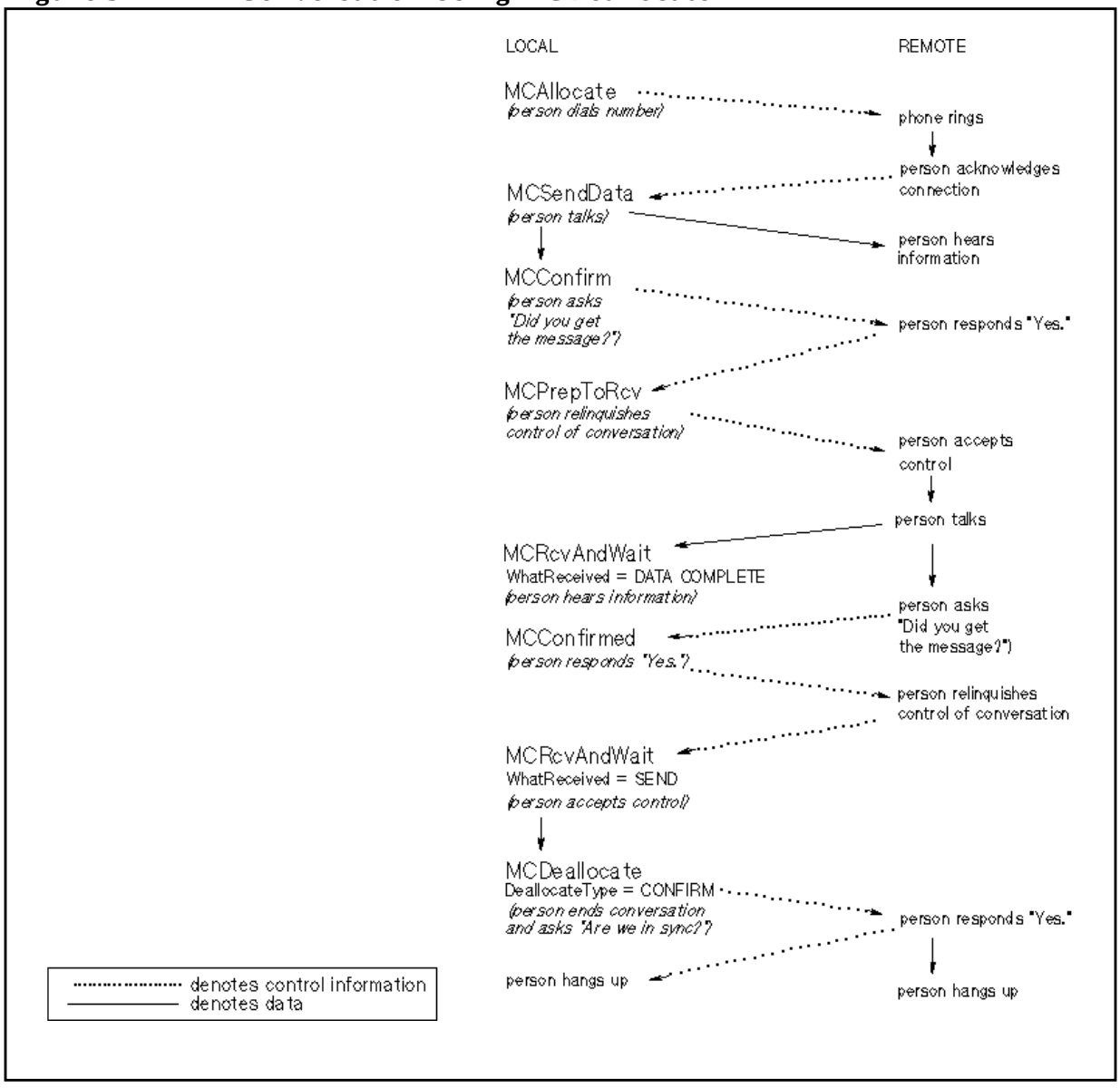
Figure 3-3 Conversation Using MCSendData



MCDeallocate

The local TP calls `MCDeallocate` to end the conversation and deallocate the resources used by the conversation. The `MCDeallocate` intrinsic replaces “person hangs up,” as shown in Figure 3-4.

Figure 3-4 Conversation Using MCDeallocate



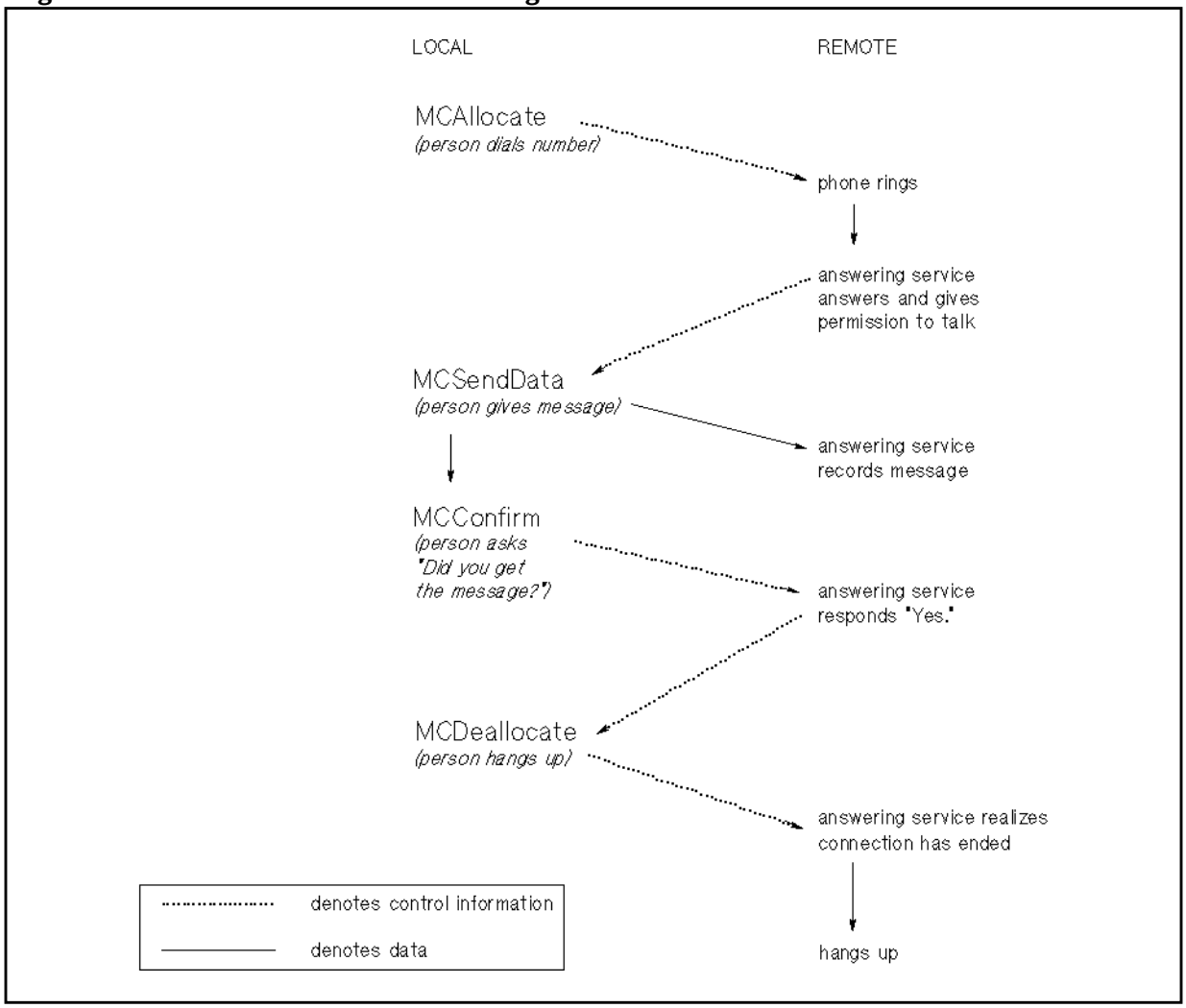
One-Way Conversation With Confirm

The synchronization level of a conversation (whether it will use confirm or not) is established when the conversation is initiated, in a parameter of the `MCAllocate` intrinsic. Figure 3-5 illustrates a one-way conversation with confirm, using the `MCConfirm` intrinsic.

MCConfirm

The `MCConfirm` intrinsic sends a confirmation request to the remote TP and waits for a response. `MCConfirm` replaces (person asks “Did you get the message?”).

Figure 3-5 Conversation Using MCConfirm



Two-Way Conversation Without Confirm

In order to carry on a two-way conversation, a TP must be able to inform its partner that it is finished sending data and is ready to receive. It also must wait for data sent by its partner and receive the data when it arrives. The two-way conversation illustrated in Figure 3-6 uses the `MCPrepToRcv` and `MCRcvAndWait` intrinsics as well as the intrinsics already introduced.

MCPrepToRcv

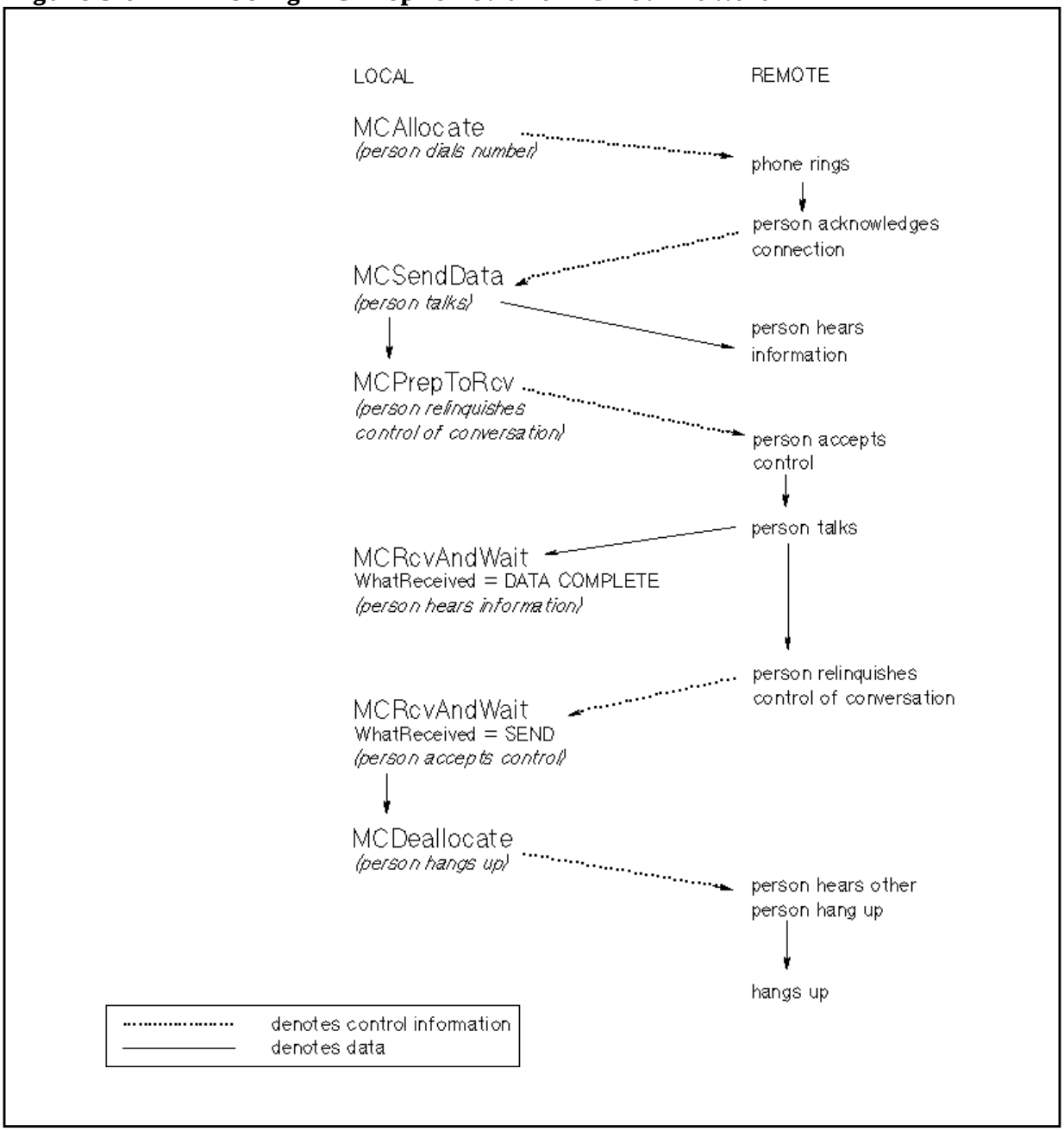
The local TP uses the `MCPrepToRcv` intrinsic to relinquish control of the conversation and give the remote TP permission to send data. It replaces “person relinquishes control of conversation.”

MCRcvAndWait

The `MCRcvAndWait` intrinsic waits for data from the remote TP and receives data when it arrives. The *`WhatReceived`* parameter of the `MCRcvAndWait` intrinsic tells the local TP what it has received: control information or data. A TP cannot receive both control information and data in the same call to `MCRcvAndWait`; it must issue a separate call to `MCRcvAndWait` for each one.

The two calls to `MCRcvAndWait` in Figure 3-6 perform different functions: The first call receives the data being sent, and the second call receives notification that the remote TP is ready to receive data.

Figure 3-6 Using MCPrepToRcv and MCRcvAndWait



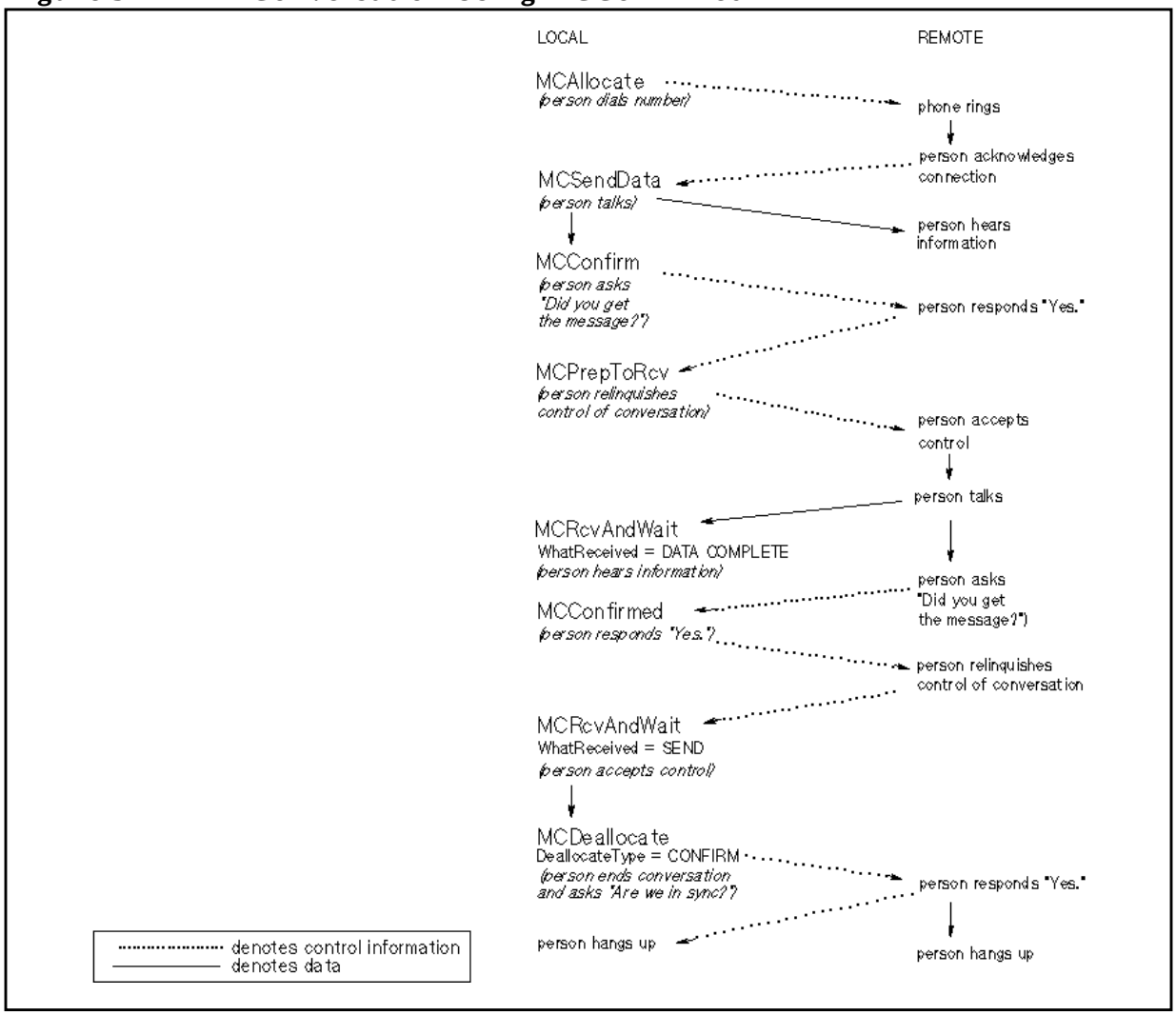
Two-Way Conversation With Confirm

Figure 3-7 illustrates a two-way conversation with confirm. Notice that, in this example, the `MCDeallocate` intrinsic is used to request confirmation, wait for a confirmation response, and then deallocate the conversation. This type of deallocation is specified by setting the *`DeallocateType`* parameter of the `MCDeallocate` intrinsic to CONFIRM.

MCConfirmed

In a two-way conversation with confirm, the `MCConfirmed` intrinsic is called in response to a confirmation request from the remote TP. It replaces person responds “Yes.”

Figure 3-7 Conversation Using MCConfirmed



LU 6.2 API Intrinsics

LU 6.2 API intrinsics are used to set up the resources for distributed transactions, allocate and conduct conversations between transaction programs, and release LU 6.2 resources after transactions have completed. Table 3-2 lists all the LU 6.2 API intrinsics and gives a brief description of each.

Table 3-2 LU 66.2 API Intrinsics

Intrinsic	Definition
TPStarted	Initializes access to the LU 6.2 API intrinsics and reserves resources for a transaction program.
TPEnded	Terminates access to the LU 6.2 API intrinsics and releases resources for a transaction program.
MCAllocate	Establishes a mapped conversation between TPs.
MCConfirm	Sends a confirmation request to the remote TP and waits for a reply.
MCConfirmed	Sends a confirmation response to a remote TP that has issued a confirmation request.
MCDeallocate	Ends a mapped conversation between TPs.
MCErrMsg	Provides the message corresponding to a given status info value.
MCGetAllocate	Receives the request from a remote TP to start a conversation and then establishes the conversation.
MCGetAttr	Returns information about a mapped conversation.
MCFlush	Flushes the LU's send buffer.
MCPostOnRcpt	Allows the LU to check the contents of the receive buffer for the specified conversation.
MCPrepToRcv	Informs the remote TP that the local TP is now ready to receive data.
MCRcvAndWait	Waits for information to arrive on the mapped conversation and then receives the information. The information can be data, conversation status, or request for confirmation.
MCRcvNoWait	Similar to MCRcvAndWait, this intrinsic receives any information that has arrived on the conversation but will not wait if no data has arrived.
MCReqToSend	Notifies the remote TP that the local TP is requesting to send data.
MCSendData	Sends data to the remote TP.
MCSendError	Informs the remote TP that the local TP has detected an error.
MCTest	Tests the conversation for the receipt of information.
MCWait	Waits for the receipt of information on one or more conversations.

Control Operator Intrinsics

In addition to the intrinsics used by transaction programmers, LU 6.2 API/XL provides a set of control operator intrinsics that allow node managers and system managers to control APPC sessions and manage the APPC subsystem. The control operator intrinsics are described in detail in the *APPC Subsystem on MPE XL Node Manager's Guide.* Table 3-3 lists the control operator intrinsics and gives a brief description of each.

NOTE The control operator intrinsics are available only on MPE XL.

Table 3-3 Control Operator Intrinsics

Intrinsic	Description
`APPCStart`	Starts up the APPC subsystem and all APPC sessions configured for automatic activation.
`APPCSessions`	Changes the session limits for a session type and activates or deactivates APPC sessions to meet the new limits.
`APPCStatus`	Returns an integer code that indicates whether the APPC subsystem is active.
`APPCStop`	Terminates the APPC subsystem and all active APPC sessions.

4 Conversation States

Conversation states indicate the status of each side of a conversation. Application programmers on different processors use conversation states to discuss how their TPs will communicate. Certain intrinsics (or verbs, depending on the implementation) can be called from each state, and control information is exchanged through parameters of the intrinsics. Control information sent by the remote TP may determine the state of the local side of the conversation.

This chapter defines all the possible states of a conversation and explains how the various intrinsics affect these states. The possible states of a conversation are listed in Table 4-1.

Table 4-1 Conversation States

State	Meaning
Reset	The TP can allocate a mapped conversation.
Send	The TP can send data or request confirmation.
Receive	The TP can receive information from the remote TP.
Confirm	The TP can reply to a confirmation request and enter or reenter Receive state.
Confirm Send	The TP can reply to a confirmation request and enter Send state.
Confirm Deallocate	The TP can reply to a confirmation request and enter Deallocate state.
Deallocate	The TP can Deallocate the mapped conversation locally.

Reset State

A TP calls the TPStarted intrinsic to gain access to LU 6.2 API. Once TPStarted has executed successfully, the TP is in **Reset state**.

Reset state is the only state associated with the entire TP. The remainder of the states are associated with individual conversations. From Reset state, a TP can call either MCAllocate or MCGetAllocate to allocate a conversation with a remote TP.

When the local TP calls MCAllocate, two things occur: 1) the local side of a conversation is established and placed in Send state; 2) a request is sent to the remote TP to start a conversation.

The local TP calls MCGetAllocate when it receives a request from a remote TP to start a conversation. When a conversation is requested by the remote TP, the local side of the conversation begins in Receive state.

On MPE V, a single TP can allocate up to 8 conversations, but it can call MCGetAllocate only once, so only one of its conversations can be initiated by the remote TP; the local TP must allocate the rest of its conversations by calling the MCAllocate intrinsic. On MPE XL, a single TP can allocate up to 256 conversations, and any number of these can be initiated by the remote TP.

Table 4-2 lists the intrinsics that can be called from Reset state. It also indicates what state the local side of the conversation is in after each intrinsic executes successfully. See Chapter 5 , “Intrinsic Descriptions,” for more information on intrinsics.

Table 4-2 Reset State Intrinsics

Intrinsic	State Entered Upon Successful Execution
MCAllocate	Send
MCGetAllocate	Receive

Send State

In **Send state**, a TP controls the conversation. A TP in Send state stays in Send state unless it requests to enter another state or the remote TP detects an error and forces the local TP to enter another state.

Table 4-3 lists all the intrinsics that can be called from Send state. It also indicates what state the local TP is in after each intrinsic executes successfully. Notice that MCRcvAndWait can place the local side of the conversation in several different states, depending on the type of information it receives from the remote TP.

The `MCDeallocate` and `MCTest` intrinsics have certain parameters listed with them, because only calls with those parameters are allowed in Send state. See the descriptions of `MCDeallocate` and `MCTest` in Chapter 5 , “Intrinsic Descriptions,” for more information.

Table 4-3 Send State Intrinsics

Intrinsic		State Entered Upon Successful Execution
`MCConfirm`		Send
`MCDeallocate` *`DeallocateType=`*	{ 1 (FLUSH) } { 2 (ABEND) } { 6 (CONFIRM) }	Reset
`MCErrMsg`		Send
`MCFlush`		Send
`MCGetAttr`		Send
`MCPrepToRcv`		Receive
`MCRcvAndWait` *`WhatReceived=`*	{ 1 (DATA_COMPLETE) } { 2 (DATA_INCOMPLETE) } { 4 (SEND) } { 5 (CONFIRM) } { 6 (CONFIRM_SEND) } { 7 (CONFIRM_DEALLOCATE) }	Receive Receive Send Confirm Confirm Send Confirm Deallocate
`MCSendData`		Send
`MCSendError`		Send
`MCTest` *`Test = 1`* (REQUEST_TO_SEND_RECEIVED		Send

Receive State

In **Receive state**, a local TP can receive data and control information from a remote TP. A local TP can enter Receive state from Send state by calling either MCPrepToRcv or MCRcvAndWait. Both of these intrinsics cause the local side of the conversation to change from Send state to Receive state and the remote side to change from Receive state to Send state.

Since the TP in Send state controls the conversation, a TP in Receive state must wait to be placed in Send state by the controlling TP. A local TP in Receive state can request to enter Send state by calling the MCReqToSend intrinsic. The remote TP in Send state receives the request, enters Receive state, and places the local TP in Send state.

NOTE If a TP in Receive state detects an error, it can enter Send state directly by calling MCSendError.

Table 4-4 lists the intrinsics that can be called from Receive state. It also indicates what state the TP is in after each intrinsic executes successfully. MCRcvAndWait and MCRcvNoWait can place the local side of the conversation in several different states, depending on the type of information received from the remote TP.

From Receive state, the MCDeallocate intrinsic can be called only with the *DeallocateType* parameter set to 2 (ABEND). See the description of MCDeallocate in Chapter 5 , “Intrinsic Descriptions,” for more information.

Table 4-4 Receive State Intrinsics

Intrinsic	State Entered Upon Successful Execution
MCDeallocate *DeallocateType= 2* (ABEND)	Reset
MCErrMsg	Receive
MCGetAttr	Receive
MCPostOnRcpt	Receive
MCRcvAndWait or MCRcvNoWait	
WhatReceived= { 1 (DATA_COMPLETE) }	Receive
{ 2 (DATA_INCOMPLETE) }	Receive
{ 4 (SEND) }	Send
{ 5 (CONFIRM) }	Confirm
{ 6 (CONFIRM_SEND) }	Confirm Send
{ 7 (CONFIRM_DEALLOCATE) }	Confirm Deallocate
MCReqToSend	Receive
MCSendError	Send
MCTest	Receive
MCWait	Receive

Confirm State

A local TP enters **Confirm state** from Receive state whenever it receives a confirmation request from the remote TP. Table 4-5 lists the intrinsics that can be called from Confirm state. The `MCDeallocate` intrinsic can be called from Confirm state only with the *`DeallocateType`* parameter set to 2 (ABEND). See the description of `MCDeallocate` in Chapter 5 , “Intrinsic Descriptions,” for more information.

Table 4-5 Confirm State Intrinsics

Intrinsic	State Entered Upon Successful Execution
`MCConfirmed`	Receive
`MCDeallocate` *`DeallocateType= 2`* (ABEND)	Reset
`MCErrMsg`	Confirm
`MCGetAttr`	Confirm
`MCReqToSend`	Confirm
`MCSendError`	Send

Confirm Send State

A TP enters **Confirm Send state** from Receive state when the remote TP issues the equivalent of the MCPrepToRcv intrinsic with a confirmation request, asking the local TP to respond to the confirmation request and enter Send state.

Table 4-6 lists the intrinsics that can be called from Confirm Send state. The MCDeallocate intrinsic can be called from Confirm Send state only with the *DeallocateType* parameter set to 2 (ABEND). See the description of MCDeallocate in Chapter 5 , “Intrinsic Descriptions,” for more information.

Table 4-6 Confirm Send State Intrinsics

Intrinsic	State Entered Upon Successful Execution
MCConfirmed	Send
MCDeallocate *DeallocateType= 2* (ABEND)	Reset
MCErrMsg	Confirm Send
MCGetAttr	Confirm Send
MCSendError	Send

Confirm Deallocate State

A TP enters **Confirm Deallocate state** from Receive state when it calls MCRcvAndWait or MCRcvNoWait and the *WhatReceived* parameter returns 7 (CONFIRM_DEALLOCATE). This indicates that the remote TP has issued the equivalent of the MCDeallocate intrinsic with a confirmation request.

Table 4-7 lists the intrinsics that can be called from Confirm Deallocate state. The MCDeallocate intrinsic can be called from Confirm Deallocate state only with the *DeallocateType* parameter set to 2 (ABEND). See the description of MCDeallocate in Chapter 5 , "Intrinsic Descriptions," for more information.

Table 4-7 Confirm Deallocate State Intrinsics

Intrinsic	State Entered Upon Successful Execution
MCConfirmed	Deallocate
MCDeallocate *DeallocateType= 2* (ABEND)	Reset
MCErrMsg	Confirm Deallocate
MCGetAttr	Confirm Deallocate
MCSendError	Send

Deallocate State

The local TP enters **Deallocate state** when it encounters an error condition or when the remote TP deallocates the conversation. Table 4-8 lists the intrinsics that can be called from Deallocate state. The `MCDeallocate` intrinsic can be called from Deallocate state only with the *`DeallocateType`* parameter set to `5` (LOCAL). See the description of `MCDeallocate` in Chapter 5 , “Intrinsic Descriptions,” for more information.

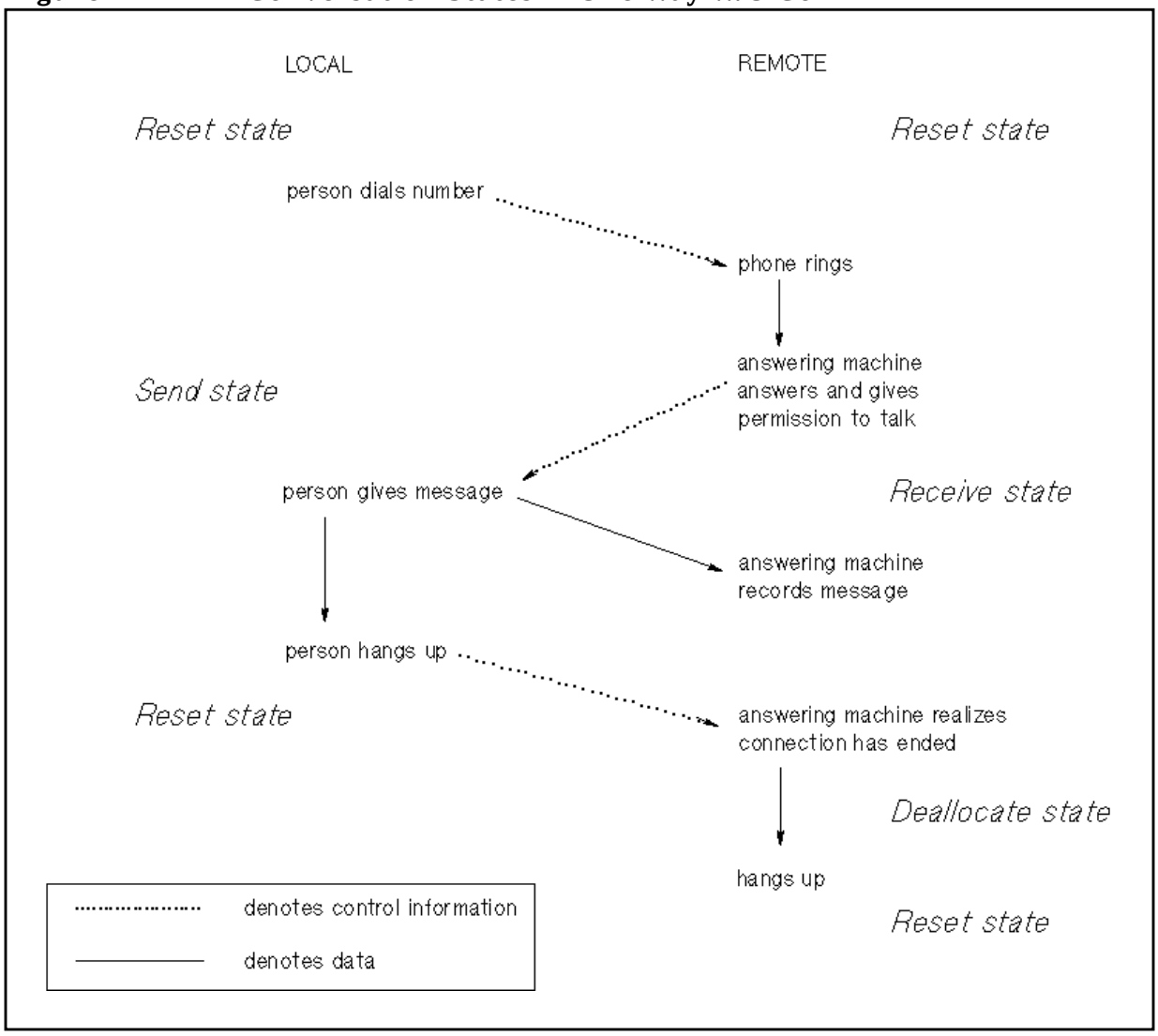
Table 4-8 Confirm Deallocate State Intrinsics

Intrinsic	State Entered Upon Successful Execution
`MCDeallocate` *`DeallocateType= 5`* (LOCAL)	Reset
`MCErrMsg`	Deallocate
`MCGetAttr`	Deallocate

One-Way Conversation Without Confirm

Figure 4-1 shows the state transitions for a one-way conversation without confirm.

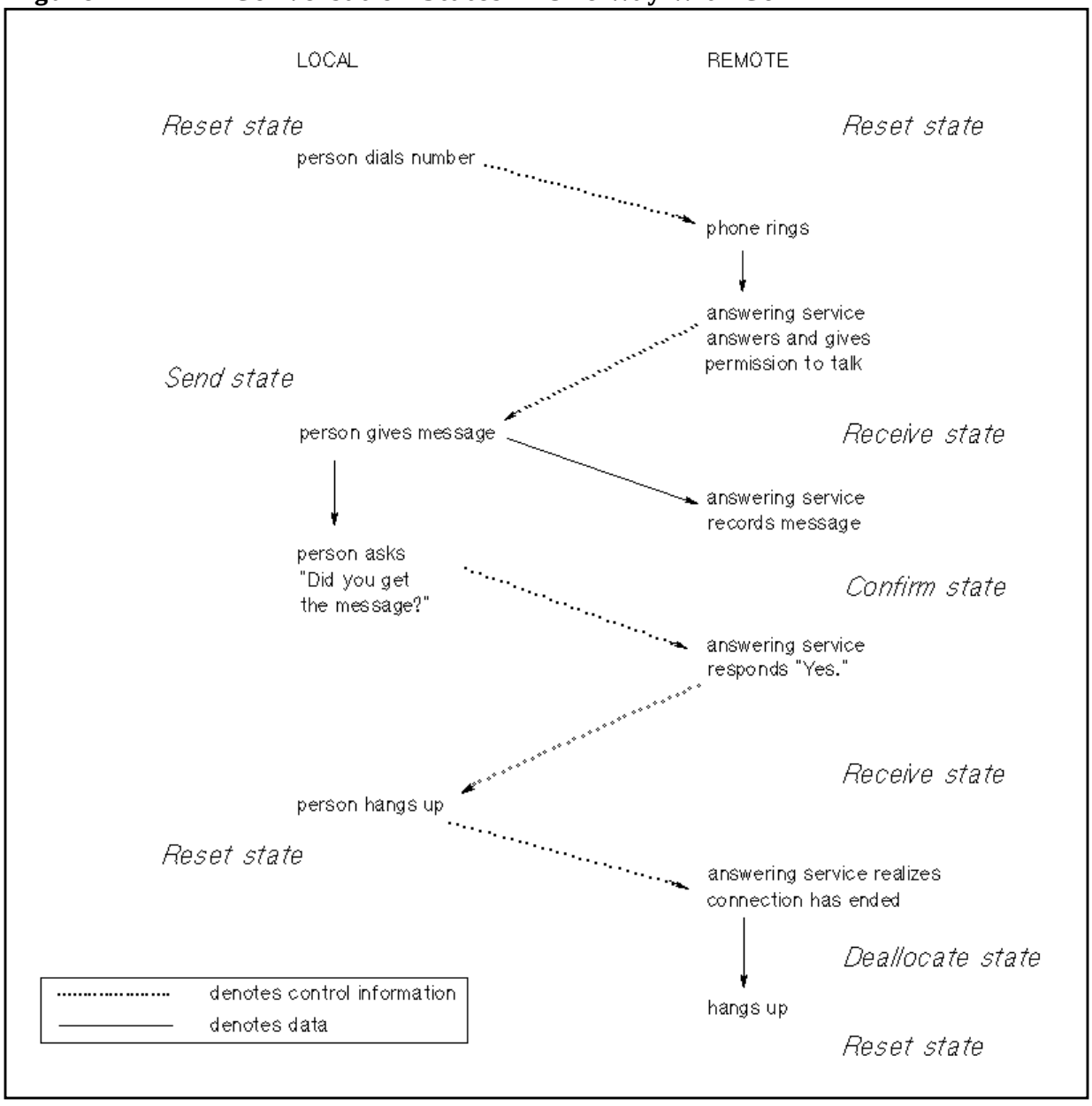
Figure 4-1 Conversation States — One-Way W/O Confirm



One-Way Conversation With Confirm

Figure 4-2 shows the state transitions for a one-way conversation with confirm.

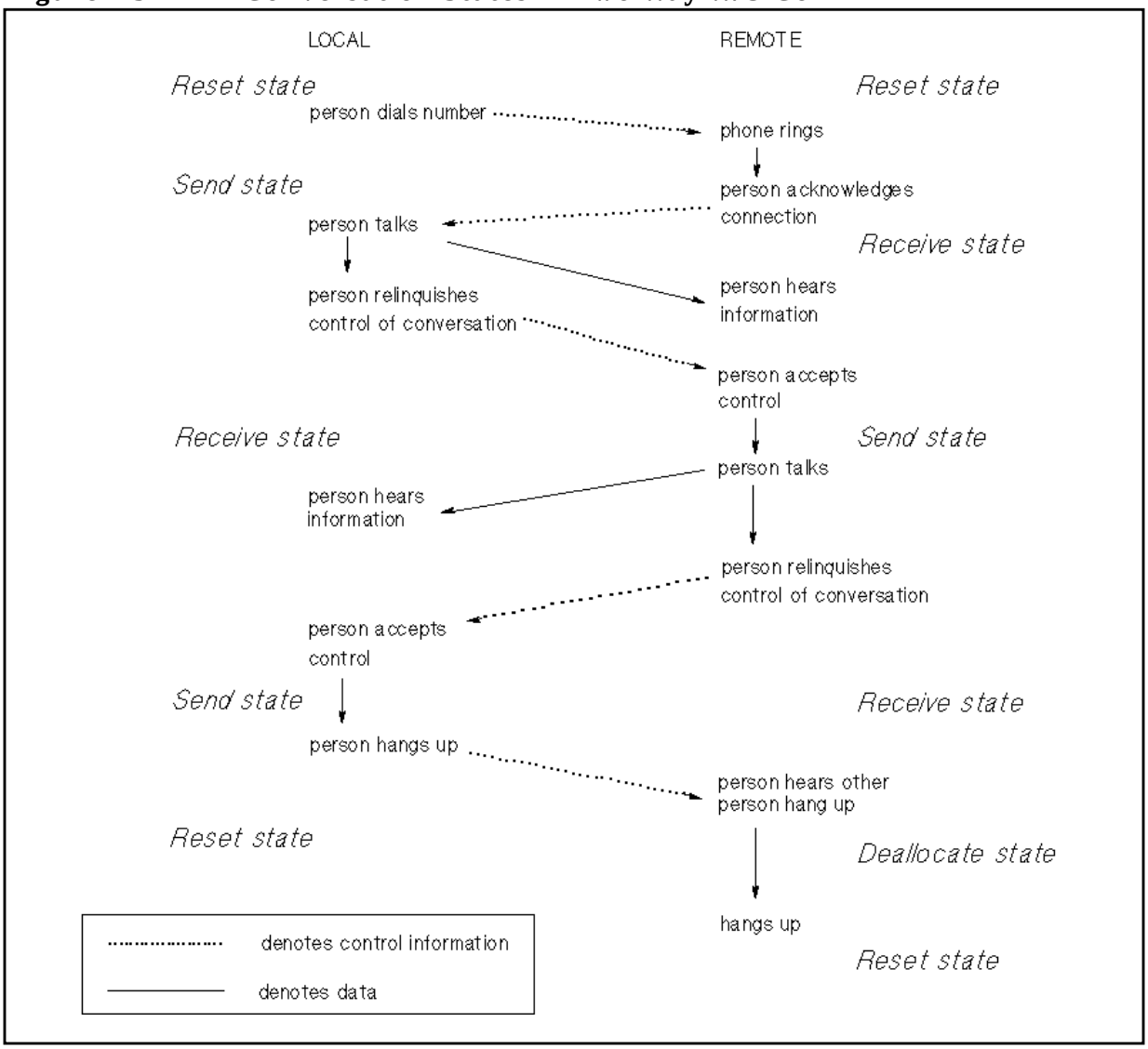
Figure 4-2 Conversation States — One-Way With Confirm



Two-Way Conversation Without Confirm

Figure 4-3 shows the state transitions for a two-way conversation without confirm.

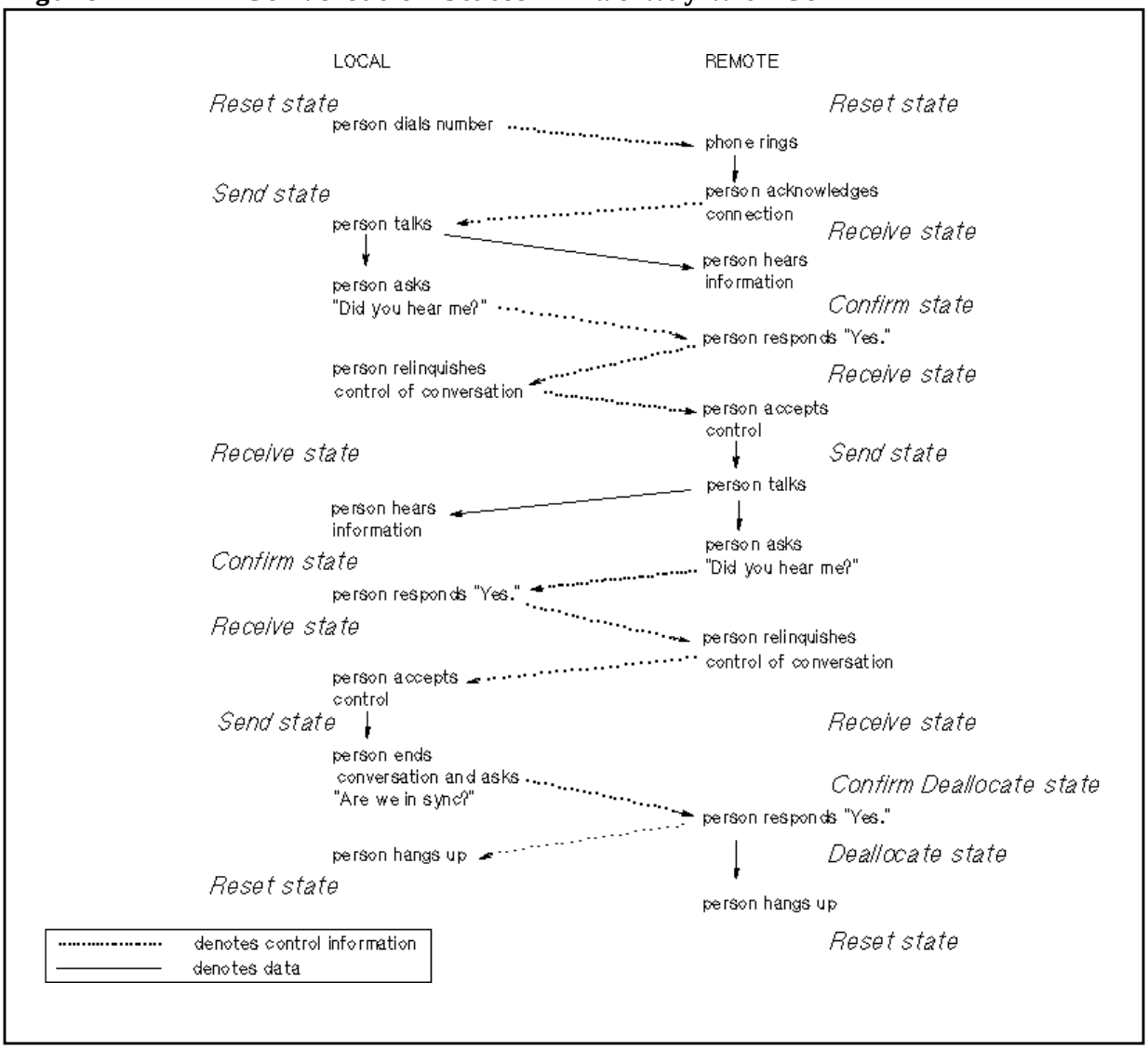
Figure 4-3 Conversation States — Two-Way W/O Confirm



Two-Way Conversation With Confirm

Figure 4-4 shows the state transitions for a two-way conversation with confirm.

Figure 4-4 Conversation States — Two-Way With Confirm



5 Intrinsic Descriptions

This chapter describes all the LU 6.2 API intrinsics. It is divided into the following sections:

The **Syntax Conventions** section explains how formatting and typefaces are used to describe the intrinsics and their parameters.

The **Parameter Data Types** section defines the mnemonics listed above the intrinsic parameters. These mnemonics indicate the data types of the parameters.

The **Status Parameter** section explains the fields of the *Status* parameter, used by all LU 6.2 API intrinsics.

The **TP Intrinsics** section describes the intrinsics used to start and stop a TP.

The **Conversation Intrinsics** section describes the intrinsics that can be used in a conversation.

CAUTION

For all LU 6.2 API intrinsics running on MPE V, the minimum stack size necessary is 4000 words.

Do not call any LU 6.2 API intrinsics in split stack mode on MPE V. If you do, a status info value of -1 will be returned.

Syntax Conventions

The syntax description for each intrinsic is given in the following form:

Syntax

```
                I32          I16V          I16
INTRINSIC NAME (parameter1, [parameter2], parameter3)
```

Optional parameters, like *parameter2*, are enclosed in square brackets. Required parameters, like *parameter1* and *parameter3*, are not enclosed in brackets.

NOTE Any optional parameters that are not used in an intrinsic call must have place holders. In most languages, a comma serves as a place holder.

Output parameters (parameters whose values are returned to the program after intrinsic execution) are underlined. In the example above, *parameter3* is an output parameter.

Input parameters (parameters whose values are passed to the intrinsic in the intrinsic call) are not underlined. In the example above, *parameter1* and *parameter2* are input parameters.

Input/output parameters are underlined. Input/output parameters are used to pass a value to the intrinsic and then to return a value to the program after intrinsic execution.

The mnemonics that appear over the parameters indicate their data type and whether they are passed by reference (the default) or by value. The mnemonics are defined in Table 5-1 and Table 5-2.

A parameter passed by value will have a V next to the mnemonic. For example, I16V indicates a 16-bit integer passed by value. A parameter without a V next to the mnemonic is passed by reference. All arrays are passed by reference.

Parameter Data Types

Above each intrinsic parameter is a mnemonic that indicates the data type of the parameter. Data types are defined generically; they are not language-specific. Table 5-1 maps each mnemonic to its generic definition, to its definition in COBOL II, and to its definition in Transact.

Table 5-1 Data Types for COBOL II and Transact

Mnemonic	Generic type	COBOL II	Transact
I16	16-bit signed integer	computational 1–4 digits	integer 1–4 digits
I32	32-bit signed integer	computational 5–9 digits	integer 5–9 digits
C	character	USAGE DISPLAY or group item	ASCII character
A	array	USAGE DISPLAY, USAGE COMP-3, or group item	compound item

Table 5-2 maps each mnemonic to its generic definition, to its definition in Pascal, and to its definition in C.

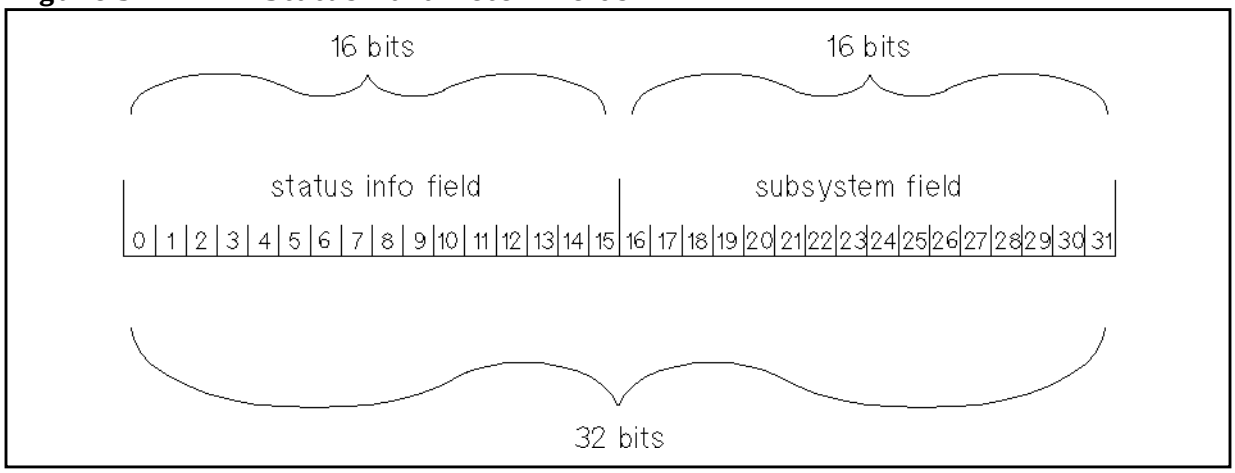
Table 5-2 Data Types for Pascal and C

Mnemonic	Generic type	Pascal	C (MPE XL only)
I16	16-bit signed integer	shortint or range type (user-defined)	short
I32	32-bit signed integer	integer or range type (user-defined)	int
C	character	char	char
A	array (any type)	array (any type)	array (any type)

Status Parameter

When a TP calls an intrinsic, information about the execution of the intrinsic is returned to the TP in the *Status* parameter. The *Status* parameter is 32 bits long and is divided into two 16-bit fields: the **status info field** and **subsystem** field, as shown in Figure 5-1.

Figure 5-1 Status Parameter Fields



The status info field is a 16-bit integer that indicates whether an error has occurred or a message has been generated. The status info field can be positive, negative, or zero.

- A value of zero indicates that the intrinsic executed successfully, and no messages were generated.
- A positive value indicates that the intrinsic executed successfully, and further information is available. The positive number in the status info field is the number of a status info message.
- A negative value indicates that the intrinsic did not complete successfully. The negative number in the status info field is the number of a status info message describing the error that has occurred.

All status info messages are listed in Appendix A , “Status Info,” in this manual, along with their causes and any actions you should take to resolve problems.

The subsystem field is a 16-bit integer that represents the subsystem from which the status info message was returned.

- A value of zero indicates that the intrinsic executed successfully and no other information is necessary. A zero is returned in the subsystem field when the status info field is zero.
- A value of 732 indicates that the message in the status info field was returned by the APPC subsystem.

Programs should be able to reference the *Status* parameter as a full 32-bit integer and as two 16-bit fields.

After executing an intrinsic, always compare the 32-bit value in the *Status* parameter to zero (successful completion).

If *Status* is not zero, compare the 16-bit value in the status info field with the numbers of any messages that can be generated by the intrinsic. At the end of every intrinsic description in this chapter is a list of the important status info messages that can be generated by that intrinsic.

Work with the application programmers at the remote site to determine the procedures you will follow if you encounter errors.

See Appendix B , “Sample Programs,” for programming examples using LU 6.2 intrinsics with the *Status* parameter.

TP Intrinsics

In Hewlett-Packard's implementation of the LU 6.2 architecture, the TPStarted intrinsic initializes access to LU 6.2 API and sets up the resources necessary to establish conversations. The TPEnded intrinsic terminates access to LU 6.2 API and releases all the resources used by the TP. The TP intrinsics and their descriptions are given in Table 5-3.

Table 5-3 LU 6.2 API TP Instrinsics

TPStarted	Initializes access to LU 6.2 API and allocates resources.
TPEnded	Terminates access to LU 6.2 API and releases resources.

TPStarted

Initializes access to LU 6.2 API and allocates resources.

Syntax

```
                CA          I16     I32
TPStarted(LocalTPName, TPID, Status,

              I16          I16V           CA              CA
          [TraceOn], [TraceSize], [TraceFile], [DefaultFile]);
```

Parameters

LocalTPName **Required**; character array; input. This parameter is an 8-character array, left justified and padded with blanks. It identifies the name of the transaction program being executed. For remotely initiated TPs on MPE XL, this parameter must match the *LocalTPName* of the MCGetAllocate intrinsic.

TPID **Required**; 16-bit signed integer; output. This number is assigned to the specific execution instance of the TP. (More than one instance of the same TP may be executing at once, and the *TPID* uniquely identifies a single instance of a TP.)

Status **Required**; 32-bit signed integer; output. Indicates the result of intrinsic execution. See the "Status Parameter" section, earlier in this chapter, for more information.

TraceOn 16-bit signed integer; input. Indicates the type of intrinsic tracing, if any, to be enabled. Possible values are as follows:

0 = no tracing

1 = API intrinsic tracing

2 = APPC subsystem intrinsic tracing

3 = API and APPC subsystem intrinsic tracing

Default: 0 (no tracing)

NOTE Specify *TraceOn* values of 2 and 3 only when asked to do so by your HP representative.

TraceSize 16-bit signed integer by value; input. This is a number

from 1 through 32767 that specifies the maximum number of logical records in the user trace file. When the maximum number of records is reached, the first record is overwritten.

Default: 1024 logical records.

TraceFile character array; input. This character array contains an actual file designator of the trace file to be used. The *TraceFile* parameter is used only when tracing is turned on with the *TraceOn* parameter. *TraceFile* can contain a fully qualified 35-character file name, with lockword, in the following form:

```
filename/lockword.groupname.acctname
```

The *TraceFile* array must be terminated with a blank. If the *TraceFile* parameter is used, the specified trace file is overwritten every time the TP is invoked. If more than one active TP specifies the same trace file name, or if you try to execute more than one instance of the same TP that references the file name, a status info value of -1033 will be returned.

Default: The default trace file name is PSTRAC*nn*, where *nn* is a number from 00 through 49. The default trace file is created in the user's logon group and account. A new trace file is created every time TPStarted is called with tracing enabled.

DefaultFile character array; output. This is a 28-character ASCII array, padded with blanks. It returns the name of the default trace file in the form PSTRAC*nn*.group.account, where *nn* is a number from 00 through 49, and group.account is the user's logon group and account.

When user tracing is enabled, but the *TraceFile* parameter is not passed, you can use the *DefaultFile* parameter to get the name of the current trace file.

Description

The TPStarted intrinsic is executed only once within a TP. It allocates resources for the TP and allows the TP to establish conversations. When TPStarted is called, the *LocalTPName* is passed to LU 6.2 API. LU 6.2 API then assigns a *TPID* to the instance of the TP that called TPStarted. The *TPID* uniquely identifies each execution instance of a TP. If several users execute the same TP at the same time, every instance of that TP will have the same *LocalTPName*, but each instance will have its own *TPID*.

NOTE For remotely initiated TPs on MPE XL, the *LocalTPName* parameter of the TPStarted intrinsic must match the *LocalTPName* parameter of the MCGetAllocate intrinsic.

The *TraceOn* parameter can have four possible values. When you are writing and debugging programs, it is useful to turn API intrinsic tracing on by setting the *TraceOn* parameter to 1. API intrinsic tracing is documented in Chapter 7 , “Debugging.” It will help you diagnose errors in your TP. Use *TraceOn* values of 2 and 3 only when asked to do so by your HP representative.

Since tracing can be turned on or off only during the execution of TPStarted, you might want to write your TP so that it will accept an info string that specifies the type of tracing to be performed. This would save having to recompile the program to change the type of tracing.

Status Info Values

```
 0     Successful Completion.
-1     Intrinsic called with parameter out of bounds.
-19    APPC subsystem is inactive.
-20    Not enough stack space for intrinsic to run.
-21    Insufficient memory space to allocate a conversation.
-90    An internal error in Presentation Services has occurred.
-95    Internal Error: Unable to create Transaction Program port.(MPE XL)
-1002  An internal error at the mapped conversation level has occurred.
-1003  Required parameter missing.
-1005  Insufficient Heap Space. (MPE V)
-1030  TPStarted request rejected.
-1033  Unable to open file specified in the 'TraceFile' parameter.
-1034  Out of range 'TraceSize' parameter specified in intrinsic call.
-1036  Out of range 'TraceOn' parameter specified in intrinsic call.
-1044  Multiple calls made to TPStarted.
```

TPEnded

Terminates access to LU 6.2 API and releases resources.

Syntax

```
 I16V   I32
TPEnded(TPID, Status);
```

Parameters

TPID **Required**; 16-bit signed integer by value; input. This number is assigned to the specific instance of the TP during the execution of the TPStarted intrinsic. (More than one instance of the same TP may be executing at once, and the *TPID* uniquely identifies a single instance of a TP.)

Status **Required**; 32-bit signed integer; output. Indicates the result of intrinsic execution. See the "Status Parameter" section, earlier in this chapter, for more information.

Description

The TPEnded intrinsic is used to release all resources allocated for the execution of this TP. If the TPStarted intrinsic has been called within a TP, the TPEnded intrinsic should be called before the TP finishes executing.

If a TP terminates abnormally, and TPEnded cannot execute successfully, LU 6.2 API will attempt to release the resources allocated for the TP.

NOTE Always call TPEnded to end your TPs. If all conversations have been deallocated, the TP is still executing, and TPEnded has not been called, the node manager cannot bring down the APPC subsystem without first aborting the TP. Call TPEnded as soon as all conversations have been deallocated.

Status Info Values

```
 0     Successful Completion.
-1     Intrinsic called with parameter out of bounds.
-15    Invalid 'TPID' parameter specified in intrinsic call.
-19    APPC subsystem is inactive.
-20    Not enough stack space for intrinsic to run.
-90    An internal error in Presentation Services has occurred.
-1002  An internal error at the mapped conversation level has occurred.
-1003  Required parameter missing.
-1040  Conversation(s) not deallocated before calling TPEnded.
```

Conversation Intrinsics

This section describes the intrinsics used to manage a conversation between TPs on different processors. Table 5-4 lists the LU 6.2 API conversation intrinsics and their descriptions.

Table 5-4 LU 6.2 API Conversation Intrinsics

Intrinsic	Definition
MCAllocate	Establishes a mapped conversation between TPs.
MCConfirm	Sends a confirmation request to the remote TP and waits for a reply.
MCConfirmed	Sends a confirmation response to a remote TP that has issued a confirmation request.
MCDeallocate	Ends a mapped conversation between TPs.
MCErrMsg	Provides the message corresponding to a given status info value.
MCGetAllocate	Receives the request from a remote TP to start a conversation and then establishes the conversation.
MCGetAttr	Returns information about a mapped conversation.
MCFlush	Flushes the LU's send buffer.
MCPostOnRcpt	Allows the LU to check the contents of the receive buffer for the specified conversation.
MCPrepToRcv	Informs the remote TP that the local TP is now ready to receive data.
MCRcvAndWait	Waits for information to arrive on the mapped conversation and then receives the information. The information can be data, conversation status, or request for confirmation.
MCRcvNoWait	Similar to MCRcvAndWait, this intrinsic receives any information that has arrived on the conversation but will not wait if no data has arrived.
MCReqToSend	Notifies the remote TP that the local TP is requesting to send data.
MCSendData	Sends data to the remote TP.
MCSendError	Informs the remote TP that the local TP has detected an error.
MCTest	Tests the conversation for the receipt of information.
MCWait	Waits for the receipt of information on one or more conversations.

MCAllocate

Establishes a conversation initiated by a local TP.

Syntax

```
              I16V      CA             CA           I16V
MCAllocate (TPID, SessionType, RemoteTPName, RemoteTPLen,

                 I16       I32          I16V           I16V
              ResourceID, Status, [ReturnControl], [SyncLevel],

                I16V      I16V       I16V        I16A
              [Timer], [Security], [NumPIPs], [PIPLengths],

                CA      CA             CA
              [PIP1,] [PIP2,] . . . [PIP16]);
```

Parameters

TPID

Required; 16-bit signed integer by value; input. This number is assigned to the specific instance of the TP during the execution of the TPStarted intrinsic. (More than one instance of the same TP may be executing at once, and the *TPID* uniquely identifies a single instance of a TP.)

SessionType

Required; character array; input. This is an 8-character ASCII array, left justified and padded with blanks. It contains the name of a session type that is configured for the APPC subsystem. For more information on session types and configuration of the APPC subsystem, see the *LU 6.2 API/V Node Manager's Guide* or the *APPC Subsystem on MPE XL Node Manager's Guide*.

RemoteTPName

Required; character array; input; EBCDIC. This is an array of up to 64 EBCDIC characters. It contains the name of the remote TP to be connected at the other end of the conversation. The remote TP must be written to use mapped conversations.

Because LU 6.2 API performs no translation on this array, the local TP must convert it from ASCII to

EBCDIC. The MPE CTRANSLATE intrinsic, or the NLTRANSLATE intrinsic on MPE XL, may be used.

RemoteTPLen

Required; 16-bit signed integer by value; input. This parameter contains the length, in characters, of the *RemoteTPName*. It must be an integer from 1 through 64.

ResourceID

Required; 16-bit signed integer; output. This number identifies the conversation being allocated. It must be used in all subsequent intrinsic calls, so that LU 6.2 API can determine which conversation the intrinsic calls belong to.

Status

Required; 32-bit signed integer; output. Indicates the result of intrinsic execution. See the "Status Parameter" section, earlier in this chapter, for more information.

ReturnControl

6-bit signed integer by value; input. Specifies when the local LU will return control to the local TP after allocating or attempting to allocate a session. Possible values are as follows:

0 = IMMEDIATE

If an active session is immediately available, it will be allocated, and control will be returned to the calling program. If no session is immediately available, no session will be allocated, and control will be returned to the calling program.

1 = WHEN_SESSION_ALLOCATED

The requested session will be allocated before control is returned to the calling program. If no session is immediately available, the request will be queued, and the calling program will be suspended until a session is activated or freed, or until the allocation request fails.

Default: 0 (IMMEDIATE)

SyncLevel

16-bit signed integer by value; input. This parameter determines the synchronization level (whether or not confirmation will be used) for this conversation. Possible values are as follows:

0 = CONFIRM — Denotes that the MCConfirm and MCConfirmed intrinsics can be called. It also means that the confirm request option of any intrinsic may be used.

2 = NONE — Denotes that no confirmation will be used. If a *SyncLevel* of 2 is specified, the MCConfirm and MCConfirmed intrinsics cannot be called during this conversation, nor can the confirm request option of any intrinsic be used during this conversation. If any confirmation is attempted with *SyncLevel* set to 2, a status info value of -31 is returned.

Default: 2 (NONE)

Timer

16-bit signed integer by value; input. This is an integer from 0 through 28800 that indicates the maximum number of seconds LU 6.2 API will wait after executing an intrinsic before returning control to the TP. (28800 seconds = 8 hours.) For example, if the local TP sets its *Timer* to 600 (10 minutes) and issues MCRcvAndWait, and no data arrives within 10 minutes, LU 6.2 API will issue a status info of +80 to the local TP, which indicates that the allotted time has expired.

A *Timer* value of zero indicates that no timer is to be used, which means that the program will wait indefinitely for an intrinsic call to complete.

The intrinsics that use the *Timer* are:

MCConfirm

MCDeallocate *DeallocateType* = CONFIRM)

MCPrepToRcv (*PrepToRcvType* = CONFIRM)

MCRcvAndWait

MCWait

See the intrinsic descriptions in this chapter for more information.

Default: 0

Security

16-bit signed integer by value; input. Reserved for future use.

NumPIPs

16-bit signed integer by value; input. This is the number (from 0 through 16) of Program Initialization Parameters (PIPs) to be sent to the remote TP. A *NumPIPs* value of 0 indicates that no PIP data will be sent.

Default: 0

PIPLengths

16-bit signed integer array; input. This is an array of up to 16 integers that indicate the lengths, in bytes, of the Program Initialization Parameters (*PIP1...PIP16*). The combined length of all the PIPs must not be greater than 1980 bytes.

NOTE If *NumPIPs* is greater than 0, the *PIPLengths* parameter is required. If *NumPIPs* is zero, the *PIPLengths* parameter is ignored.

PIP1, PIP2,...PIP16

character array; input; EBCDIC. Each PIP is a character array containing a Program Initialization Parameter for the remote TP. PIPs are used to transmit any special information the local TP wants to send to the remote TP at conversation initiation. The combined length of all the PIPs must not be greater than 1980 bytes.

Because LU 6.2 API performs no translation on this array, the local TP must convert it from ASCII to EBCDIC. The MPE CTRANSLATE intrinsic, or the NLTRANSLATE intrinsic on MPE XL, may be used.

NOTE If *NumPIPs* is greater than 0, the specified number of PIPs must be supplied. If *NumPIPs* is zero, all PIPs are ignored.

Description

The MCAllocate intrinsic establishes a conversation between the local TP that calls it and the remote TP specified in the *RemoteTPName* parameter. Once the MCAllocate intrinsic has executed successfully, the local TP is in Send state and the remote TP is in Receive state.

When the local TP initiates a conversation, the MCAllocate intrinsic must be the first LU 6.2 API intrinsic called after TPStarted, unless the TPEnded intrinsic is called and the program terminated. See MCGetAllocate, later in this chapter, for information about conversations initiated by the remote TP.

When the MCAllocate intrinsic executes successfully, a *ResourceID* is assigned to the allocated conversation. Just as the *TPID* uniquely identifies one among many possible instances of the same TP, the *ResourceID* uniquely identifies one among many possible conversations conducted by one instance of a TP. Every time MCAllocate is called, a new conversation is established, and a unique *ResourceID* is assigned to it.

Every conversation requires one APPC session. On MPE V, only 8 sessions may be active on the APPC subsystem at once. On MPE XL, the APPC subsystem can support a maximum of 256 active sessions. The available active sessions must be shared among all the TPs running on the APPC subsystem, so your TP can allocate only as many conversations as there are available sessions.

In addition to the session limit for the whole APPC subsystem, there is also a session limit configured for each session type. For more information on session limits and session type configuration, see the *LU 6.2 API/V Node Manager's Guide* or the *APPC Subsystem on MPE XL Node Manager's Guide*.

The *ReturnControl* parameter determines what your program will do if no session is available when it calls MCAllocate. If you specify 0 (IMMEDIATE) in the *ReturnControl* parameter, your program will not wait for a session to be activated or freed; control will be returned to your program immediately. If you specify 1 (WHEN_SESSION_ALLOCATED) in the *ReturnControl* parameter, your program will be suspended until a session is activated or until a conversation is deallocated, freeing a session.

NOTE

If you specify 1 (WHEN_SESSION_ALLOCATED) in the *ReturnControl* parameter, your program can be suspended indefinitely. If no session becomes available, or if there is an error with the session type, your program will be suspended until the APPC subsystem shuts down or discovers the error.

LU 6.2 API does not wait for a response from the host before it returns a status info value. Therefore, allocation errors due to host problems do not appear in the *Status* parameter of the MCAllocate intrinsic. (See Chapter 6 , "Buffer Management.") To verify that a conversation has been allocated successfully, call MCAllocate with a *SyncLevel* of 0 (CONFIRM), and then call MCConfirm. See MCConfirm, later in this chapter, for more information.

MCAllocate allows you to send initialization information to the remote TP in the Program Initialization Parameters (PIPs). These parameters can be used, for example, to indicate whether a TP is processing daily information or an end-of-month report. They can also be used to inform the remote TP of the type of data it will receive, the size of the records, etc. PIPs can be used to transmit any special information the local TP must send to the remote TP before it executes.

Status Info Values

```
 0    Successful Completion.
-1    Intrinsic called with parameter out of bounds.
-4    Out of range 'ReturnControl' parameter specified in intrinsic call.
-5    Out of range 'SyncLevel' parameter specified in intrinsic call.
-6    PIP data length is out of range.
-7    Out of range 'Timer' parameter specified in intrinsic call.
-15   Out of range 'TPID' parameter specified in intrinsic call.
-19   APPC subsystem is inactive.
-20   Not enough stack space for intrinsic to run.
-21   Insufficient memory space to allocate a conversation.
-22   Internal error: Unable to create Presentation Services port.
-23   Unable to allocate a conversation.
-24   Unable to obtain an LU-LU session.
-90   An internal error in Presentation Services has occurred.
-91   An internal error in the APPC subsystem has occurred.
-1002 An internal error at the mapped conversation level has occurred.
-1003 Required parameter missing.
-1005 Insufficient Heap Space.  (MPE V)
-1006 Out of range 'RemoteTPLen' parameter specified in intrinsic call.
-1007 Out of range 'NumPIPs' parameter specified in intrinsic call.
-1009 Combined length of PIPs is out of range.
```

MCConfirm

Sends a confirmation request to the remote program and waits for a reply. A status info value of 0 indicates that the remote TP has returned a positive confirmation response (the equivalent of an MCConfirmed).

Syntax

```
                I16V              I16              I32
MCConfirm (ResourceID, RequestToSendReceived, Status);
```

Parameters

ResourceID

Required; 16-bit signed integer by value; input. This is the unique resource ID number assigned to this conversation when it was allocated. See MCAllocate or MCGetAllocate, in this chapter, for more information.

RequestToSendReceived

Required; 16-bit signed integer; output. This is a flag that indicates whether a RequestToSend notification has been received from the remote TP.

1 = YES

A RequestToSend notification has been received from the remote TP. The remote TP has issued the equivalent of MCReqToSend, requesting that the local TP enter Receive state and place the remote TP in Send state. See the description of the MCReqToSend intrinsic, later in this chapter.

0 = NO

No RequestToSend notification has been received.

Status

Required; 32-bit signed integer; output. Indicates the result of intrinsic execution. See the "Status Parameter" section, earlier in this chapter, for more information.

Description

The MCConfirm intrinsic is used to determine whether the two sides of a conversation are synchronized. It can be used only if the conversation is allocated with a synchronization level of CONFIRM (that is, if the

MCAllocate or MCGetAllocate intrinsic was called with the *SyncLevel* parameter set to 0). It is used to request confirmation from the remote TP and wait for a reply. A TP must be in Send state to call MCConfirm.

How confirmation is used in a conversation is up to the TP programmers. It can be used, for example, to verify that a conversation has been allocated properly and that the remote TP is ready to receive data. It can also be used after data is sent, to verify that the remote TP received everything the local TP sent.

When MCConfirm executes, the send buffer is flushed, and then a confirmation request is sent to the remote TP. The remote TP is placed in Confirm state. It may respond to the confirmation request with the equivalent of any of the intrinsics listed in Table 5-5. Table 5-5 also lists the conversation state of the remote TP after issuing each intrinsic. See Chapter 4 , “Conversation States,” for more information on states and the intrinsics that may be called from them.

Table 5-5 Intrinsics With Confirmation Responses

Intrinsic Callable from Confirm State	State of Remote TP After Execution
MCConfirmed	Receive state
MCDeallocate (*DeallocateType* = ABEND)	Reset state
MCSendError	Send state

The conversation is suspended until the MCConfirm intrinsic completes.

Because this intrinsic causes the send buffer to be flushed, it may be called after MCAllocate to determine whether the conversation was successfully allocated on the remote side. See Chapter 6 , “Buffer Management,” for more information on receiving allocation errors.

Some status info values commonly returned to the MCConfirm intrinsic are described below:

```
0     Successful Completion.
```

The remote TP has responded with the equivalent of MCConfirmed.

```
31    Confirm not allowed.
```

The conversation was not allocated with a synchronization level of CONFIRM.

```
50    Allocation Error.
```

The conversation was not allocated properly by the remote TP.

```
60    Program Error:  Data may have been purged.
```

The remote TP has sent the equivalent of MCSendError.

Following is a complete list of status info values that may be returned to the MCConfirm intrinsic.

Status Info Values

```
 0     Successful Completion.
-1     Intrinsic called with parameter out of bounds.
-2     Invalid 'ResourceID' parameter specified in intrinsic call.
-20    Not enough stack space for intrinsic to run.
-31    Confirm not allowed.
-40    Intrinsic called in invalid state.
-50    Allocation Error.
-51    Resource Failure:  No retry possible.
-52    Resource Failure:  Retry possible.
-60    Program Error:  Data may have been purged.
+80    Timer has expired.
-90    An internal error in Presentation Services has occurred.
-91    An internal error in the APPC subsystem has occurred.
-1002  An internal error at the mapped conversation level has occurred.
-1003  Required parameter missing.
-1020  Deallocate Abend.
```

MCConfirmed

Sends a positive confirmation response to the remote TP.

Syntax

```
                I16V      I32
MCConfirmed(ResourceID, Status);
```

Parameters

ResourceID **Required**; 16-bit signed integer by value; input. This is the unique resource ID number assigned to this conversation when it was allocated. See MCAllocate or MCGetAllocate, in this chapter, for more information.

Status **Required**; 32-bit signed integer; output. Indicates the result of intrinsic execution. See the "Status Parameter" section, earlier in this chapter, for more information.

Description

The MCConfirmed intrinsic sends a positive response to a confirmation request that was issued by the remote. The remote TP can send a confirmation request with the equivalent of any of the intrinsics listed in Table 5-6. Table 5-6 also lists the conversation state of the local TP after the remote TP issues each intrinsic.

The MCConfirmed intrinsic can be used only if the conversation was allocated with a synchronization level of CONFIRM (that is, if the MCAllocate or MCGetAllocate intrinsic was called with the *SyncLevel* parameter set to 0).

Table 5-6 Intrinsics With Confirmation Requests

Intrinsic Requesting Confirmation	State of Local TP After Remote Calls Intrinsic
MCConfirm	Confirm state
MCPrepToRcv (*PrepToRcvType* = CONFIRM or *PrepToRcvType* = CONVERSATION_SYNC_LEVEL, if conversation was allocated with *SyncLevel* of CONFIRM)	Confirm Send state
MCDeallocate (*DeallocateType* = CONFIRM or *DeallocateType* = CONVERSATION_SYNC_LEVEL, if conversation was allocated with *SyncLevel* of CONFIRM)	Confirm Deallocate state

The local side of the conversation must be in one of the Confirm states to issue the MCConfirmed intrinsic.

The MCConfirmed intrinsic can be used only to send a positive response to a confirmation request. Use the MCSendError intrinsic to send a negative response to a confirmation request.

Status Info Values

```
 0     Successful Completion.
-1     Intrinsic called with parameter out of bounds.
-2     Invalid 'ResourceID' parameter specified in intrinsic call.
-20    Not enough stack space for intrinsic to run.
-40    Intrinsic called in invalid state.
-51    Resource Failure: No retry possible.
-52    Resource Failure:  Retry possible.
-90    An internal error in Presentation Services has occurred.
-91    An internal error in the APPC subsystem has occurred.
-1002  An internal error at the mapped conversation level has occurred.
-1003  Required parameter missing.
```

MCDeallocate

Deallocates the specified conversation.

Syntax

```
                I16V          I16V          I32
MCDeallocate(ResourceID, [DeallocateType], Status);
```

Parameters

ResourceID

Required; 16-bit signed integer by value; input. This is the unique resource ID number assigned to this conversation when it was allocated. See MCAllocate or MCGetAllocate, in this chapter, for more information.

DeallocateType

16-bit signed integer by value; input. The type of deallocation to be performed. Possible values are as follows:

0 = CONVERSATION_SYNC_LEVEL

Denotes that the conversation should be deallocated with the synchronization level specified by the *SyncLevel* parameter of the MCAllocate or MCGetAllocate intrinsic. The *SyncLevel* parameter can specify synchronization levels of CONFIRM and NONE.

If the conversation was allocated with a *SyncLevel* of CONFIRM, then the conversation is deallocated as if CONFIRM were given as the *DeallocateType*. See the discussion of CONFIRM (*DeallocateType* = 6).

If the conversation was allocated with a *SyncLevel* of NONE, then the conversation is deallocated as if FLUSH were given as the *DeallocateType*. See the discussion of FLUSH (*DeallocateType* = 1).

1 = FLUSH

Causes the local LU to empty its send buffer and release the conversation resources normally. The conversation must be in Send state to use a *DeallocateType* of FLUSH. A *DeallocateType* of FLUSH may be specified no matter what the synchronization level of the conversation is.

2 = ABEND

Allows the conversation to deallocate in any state except Deallocate state. All buffers are flushed. If the conversation is in Receive state, loss of data can occur.

5 = LOCAL

Deallocates the conversation from Deallocate state.

6 = CONFIRM

Causes an internal execution of the MCConfirm intrinsic. The remote TP must respond with positive confirmation before the conversation can be deallocated. This *DeallocateType* can be used only if the synchronization level of the conversation is CONFIRM. The conversation must be in Send state to use a *DeallocateType* of CONFIRM.

Default: 0 (CONVERSATION_SYNC_LEVEL) Note that the default cannot be used in all cases, because the conversation must be in Send state to use a *DeallocateType* of CONVERSATION_SYNC_LEVEL.

Status

Required; 32-bit signed integer; output. Indicates the result of intrinsic execution. See the "Status Parameter" section, earlier in this chapter, for more information.

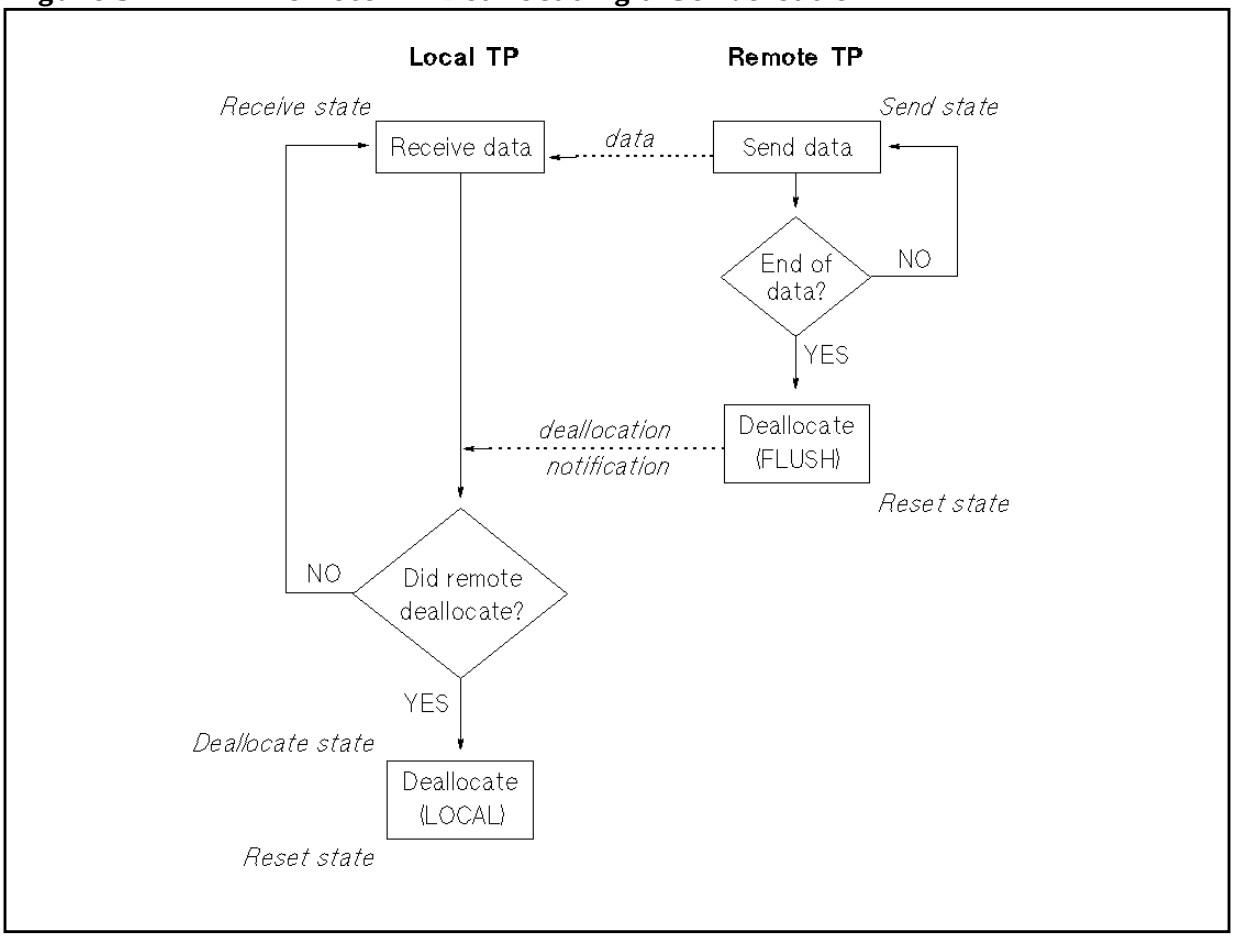
Description

The MCDeallocate intrinsic releases the resources used for a conversation. Before the TPEnded intrinsic can be called to stop the TP, MCDeallocate must be called for each conversation the TP is engaged in.

MCDeallocate with *DeallocateType* of ABEND ends posting. See the MCPostOnRcpt intrinsic description, later in this chapter, for more information about posting.

A TP in Deallocate state calls MCDeallocate with a *DeallocateType* of LOCAL. Unless there is an error, a *DeallocateType* of LOCAL is used when the remote TP deallocates first. Figure 5-2 shows the remote TP deallocating a conversation. When the remote calls the equivalent of MCDeallocate, the local TP is placed in Deallocate state. From Deallocate state, the local TP deallocates its side of the conversation with a *DeallocateType* of LOCAL.

Figure 5-2 Remote TP Deallocating a Conversation



Status Info Values

```
 0     Successful Completion.
-1     Intrinsic called with parameter out of bounds.
-2     Invalid 'ResourceID' parameter specified in intrinsic call.
-8     Out of range 'DeallocateType' specified in intrinsic call.
-20    Not enough stack space for intrinsic to run.
-31    Confirm not allowed.
-40    Intrinsic called in invalid state.
-50    Allocation Error.
-51    Resource Failure:  No retry possible.
-52    Resource Failure:  Retry possible.
-60    Program Error:  Data may have been purged.
+80    Timer has expired.
-90    An internal error in Presentation Services has occurred.
-91    An internal error in the APPC subsystem has occurred.
-1002  An internal error at the mapped conversation
       level has occurred.
-1003  Required parameter missing.
-1020  Deallocate Abend.
```

MCErrMsg

Provides the message corresponding to a status info value that was returned in a previous intrinsic call.

Syntax

```
              I32V          CA              I16         I32
MCErrMsg(OldStatus, MessageBuffer, MessageLength, Status);
```

Parameters

OldStatus

Required; 32-bit signed integer by value; input. The status info value for which a corresponding message is desired. This is a value that was returned in the *Status* parameter of a previous intrinsic call. MCErrMsg returns the message that corresponds to the value in this parameter.

MessageBuffer

Required; character array; output. A 256-byte character array in which the error message is returned.

MessageLength

Required; 16-bit signed integer; output. An integer representing the length, in bytes, of the message returned in *MessageBuffer*.

Status

Required; 32-bit signed integer; output. This status info value contains information about the execution of the MCErrMsg intrinsic. See the "Status Parameter" section, earlier in this chapter, for more information.

Description

The MCErrMsg intrinsic gives you the message that corresponds to the status info value you received in a previous intrinsic call. The MCErrMsg intrinsic obtains only local information; it will not return messages generated on the remote side.

MCErrMsg can be called from any conversation state, and it does not change the state of the conversation.

Status Info Values

```
 0     Successful Completion.
-1     Intrinsic called with parameter out of bounds.
-16    Unable to open catalog file.
-17    GENMESSAGE failed. (MPE V)
-17    CATREAD failed. (MPE XL)
-20    Not enough stack space for intrinsic to run.
-28    Invalid 'OldStatus' passed to error message intrinsic.
-90    An internal error in Presentation Services has occurred.
-91    An internal error in the APPC subsystem has occurred.
-1002  An internal error at the mapped conversation level has occurred.
-1003  Required parameter missing.
```

MCFlush

Flushes the local LU's send buffer, sending everything in it to the remote LU's receive buffer.

Syntax

```
            I16V      I32
MCFlush(ResourceID, Status);
```

Parameters

ResourceID **Required**; 16-bit signed integer by value; input. This is the unique resource ID number assigned to this conversation when it was allocated. See MCAllocate or MCGetAllocate, in this chapter, for more information.

Status **Required**; 32-bit signed integer; output. Indicates the result of intrinsic execution. See the "Status Parameter" section, earlier in this chapter, for more information.

Description

The MCFlush intrinsic can be issued only in Send state. It causes the contents of the send buffer to be sent immediately to the remote LU's receive buffer. If the send buffer is empty, no transmission takes place.

On the HP 3000 side, each conversation has a send buffer. On MPE V, the size of the send buffer is always 2044 bytes. On MPE XL, the send buffer is the same size as the maximum RU size for that session. (The maximum RU size is a configured value associated with the session type. See the *APPC Subsystem on MPE XL Node Manager's Guide*.) Data sent from the HP 3000 side accumulates in the send buffer until the buffer is full or until an intrinsic is called that causes all the data in the buffer to be sent.

The following intrinsics cause the send buffer to be flushed:

MCFlush
MCConfirm
MCDeallocate
MCPrepToRcv
MCRcvAndWait (when called from Send state)
MCSendError (when called from Send state)

Status Info Values

```
 0     Successful Completion.
-1     Intrinsic called with parameter out of bounds.
-2     Invalid 'ResourceID' parameter specified in intrinsic call.
-20    Not enough stack space for intrinsic to run.
-40    Intrinsic called in invalid state.
-90    An internal error in Presentation Services has occurred.
-91    An internal error in the APPC subsystem has occurred.
-1002  An internal error at the mapped conversation level has occurred.
-1003  Required parameter missing.
```

MCGetAllocate

Establishes a conversation initiated by a remote TP.

Syntax

```
                 I16V       CA              CA          I16
MCGetAllocate(TPID, SessionType, LocalTPName, ResourceID,

               I32         I16         I16V       I16V
              Status, [SyncLevel], [Timer], [Security],

                I16V         I16A         CA       CA                CA
              [NumPIPs], [PIPLengths], [PIP1,] [PIP2,] . . . [PIP16]);
```

Parameters

TPID **Required**; 16-bit signed integer by value; input. This number is assigned to the specific instance of the TP during the execution of the TPStarted intrinsic. (More than one instance of the same TP may be executing at once, and the *TPID* uniquely identifies a single instance of a TP.)

SessionType **Required**; character array; output. This is an 8-character ASCII array, left justified and padded with blanks. It contains the name of a session type that is configured for the APPC subsystem. For more information on session types and configuration of the APPC subsystem, see the *LU 6.2 API/V Node Manager's Guide* or the *APPC Subsystem on MPE XL Node Manager's Guide*.

LocalTPName **Required**; character array; output in EBCDIC on MPE V; input in ASCII on MPE XL.;

NOTE If you are migrating a TP to LU 6.2 API/XL from LU 6.2 API/V or from a version of LU 6.2 API/XL prior to the Node Type 2.1 version, you must change the *LocalTPName* parameter from an output parameter to an input parameter.

On **MPE V**, *LocalTPName* is an output parameter, received in EBCDIC from the remote TP. It contains the name of the job file sent by the remote TP. The job file name must be 8 characters long; if it is shorter than 8 characters long, it must be padded with blanks. The job file must be located in the APPC.SYS group and account. When the HP 3000 receives the job file name, it streams the job, which runs the local TP.

Because LU 6.2 API performs no translation on the *LocalTPName* array,

the local TP must convert it from EBCDIC to ASCII. The MPE CTRANSLATE intrinsic, or the intrinsic on MPE XL, may be used.

On **MPE XL**, *LocalTPName* is an input ASCII parameter. It must match the value in the *LocalTPName* parameter of the TPStarted intrinsic. The *LocalTPName* parameter contains the TP name sent by the remote TP. This TP name must be configured in the APPC configuration branch of NMMGR. The configuration file associates each TP name with a job file, which may be located in any group and account. When LU 6.2 API receives the TP name from the remote TP, it does one of two things:

1. If the local TP is already active, and if it is configured to receive multiple allocate requests from remote TPs, LU 6.2 API waits until any conversation with the local TP process has been deallocated, and then it allocates a conversation with the local TP process.
2. If the local TP is not active, or if it is configured to receive only one allocate request per execution instance, LU 6.2 API initiates a new local TP process by streaming the job file associated with the TP name. Then, it allocates a conversation with the local TP process.

A remotely initiated TP on MPE XL is configured to conduct either single or multiple conversations with the remote TP. A TP that conducts a single conversation must call MCGetAllocate only once; another instance of the TP will be started each time a remote TP requests a conversation. A TP that conducts multiple conversations can call MCGetAllocate many times; only one instance of it may be running at once. See the *APPC Subsystem on MPE XL Node Manager's Guide* for more information on TP configuration.

ResourceID **Required**; 16-bit signed integer; output. This number identifies the conversation being allocated. It must be used in all subsequent intrinsic calls, so that LU 6.2 API can determine which conversation the intrinsic calls belong to.

Status **Required**; 32-bit signed integer; output. Indicates the result of intrinsic execution. See the "Status Parameter" section, earlier in this chapter, for more information.

SyncLevel 16-bit signed integer; output. This parameter determines the synchronization level (whether or not confirmation will be used) for this conversation. Possible values are as follows:

0 = CONFIRM

Denotes that the MCConfirm and MCConfirmed intrinsics can be called. It also means that the confirm request option of any intrinsic may be used.

2 = NONE

Denotes that no confirmation will be used. If a *SyncLevel* of 2 is specified, the MCConfirm and MCConfirmed intrinsics cannot be called during this conversation, nor can the confirm request option of any intrinsic be used during this conversation. If any confirmation is attempted with *SyncLevel* set to 2, a status info value of -31 is returned.

Timer 16-bit signed integer by value; input. This is an integer from 0 through 28800 that indicates the maximum number of seconds LU 6.2 API will wait after executing an intrinsic before returning control to the TP. (28800 seconds = 8 hours.) For example, if the local TP sets its *Timer* to 600 (10 minutes) and issues MCRcvAndWait, and no data arrives within 10 minutes, LU 6.2 API will issue a status info of +80 to the local TP, which indicates that the allotted time has expired.

A *Timer* value of zero indicates that no timer is to be used, which means that the program will wait indefinitely for an intrinsic call to complete.

The following intrinsics use the *Timer*:

```
MCConfirm
MCDeallocate (DeallocateType = CONFIRM)
MCPrepToRcv (PrepToRcvType = CONFIRM)
MCRcvAndWait
MCWait
```

Default: 0

Security 16-bit signed integer by value; input. Reserved for future use.

NumPIPs 16-bit signed integer by value; input/output. Indicates the number of Program Initialization Parameters (PIPs) to be used for this conversation.

Input:

The *NumPIPs* value that the local TP passes to the intrinsic specifies the maximum number of PIPs the local TP can receive from the remote TP on this conversation.

Output:

The *NumPIPs* value that the intrinsic returns to the local TP indicates the number of PIPs that the remote TP actually sent to the local TP.

NOTE If the remote TP sends more PIPs than the value specified in the *NumPIPs* parameter, the *PIPLengths* parameter and all PIPs are ignored. A status info value of -1010 is returned.

PIPLengths 16-bit signed integer array; input/output. This is an array of up to 16 integers that indicate the lengths, in bytes, of the Program Initialization Parameters (*PIP1*...*PIP16*). The combined length of all the PIPs must not be greater than 1980 bytes.

Input:

The values the local TP passes in *PIPLengths* indicate the maximum number of characters each PIP can receive.

Output:

The values the intrinsic returns in *PIPLengths* indicate the actual lengths of the PIPs sent by the remote to the local TP.

NOTE If *NumPIPs* is greater than 0, the *PIPLengths* parameter is required. If *NumPIPs* is zero, the *PIPLengths* parameter is ignored.

PIP1, PIP2,...PIP16

character array; output; EBCDIC. Each PIP is a character array containing a Program Initialization Parameter received from the remote TP. PIPs are used to transmit any special information the remote TP wants to send to the local TP at conversation initiation. The combined length of all the PIPs must not be greater than 1980 bytes.

Because LU 6.2 API performs no translation on this array, the local TP must convert it from EBCDIC to ASCII. The MPE CTRANSLATE intrinsic, or the intrinsic on MPE XL, may be used

NOTE If *NumPIPs* is greater than 0, the specified number of PIPs must be supplied. If *NumPIPs* is zero, all PIPs are ignored.

Description

MCGetAllocate is called to receive a conversation allocation request from the remote TP and initialize all the resources the local TP needs to conduct a conversation. When the remote TP issues the equivalent of the MCAllocate intrinsic, it sends a TP name to the HP 3000, indicating the local TP with which it wants a conversation. The

HP 3000 then runs the local TP, which calls MCGetAllocate to allocate the local side of the conversation.

Once the MCGetAllocate intrinsic has executed successfully, the local TP is in Receive state and the remote TP is in Send state.

When the MCGetAllocate intrinsic executes successfully, a *ResourceID* is assigned to the allocated conversation. Just as the *TPID* uniquely identifies one among many possible instances of the same TP, the *ResourceID* uniquely identifies one among many possible conversations conducted by one instance of a TP.

Every conversation requires one session on the APPC subsystem. On MPE V, only 8 sessions may be active on the APPC subsystem at once. On MPE XL, the APPC subsystem can support a maximum of 256 active sessions. The available active sessions must be shared among all the TPs running on the APPC subsystem. For more information on session limits, see the *APPC Subsystem on MPE V Node Manager's Guide* or the *APPC Subsystem on MPE XL Node Manager's Guide*.

MCGetAllocate allows you to receive initialization information from the remote TP in the Program Initialization Parameters (PIPs). These parameters can be used, for example, to indicate whether a TP is processing daily information or an end-of-month report. They can also be used to inform the local TP of the type of data it will receive, the size of the records, etc. PIPs can be used to transmit any special information the local TP must receive from the remote TP before it executes.

Remotely Initiated Conversations on MPE V

On MPE V, MCGetAllocate can be called only once during the execution of a TP. That is, each execution instance of a TP can have only one conversation initiated by the remote TP.

On MPE V, when a conversation is initiated by the remote TP, the MCGetAllocate intrinsic must be the first LU 6.2 API intrinsic called after the TPStarted intrinsic, unless the TPEnded intrinsic is called and the program terminated.

On MPE V, the TP name sent by the remote TP is the name of a job file located in the APPC.SYS group and account. This TP name must match the name of the job file exactly. The job file contains the MPE RUN command that starts up the local TP. The executable TP file can reside in any group and account. When LU 6.2 API receives the TP name from the remote TP, it streams the job, which in turn runs the local TP. The value returned in the *LocalTPName* parameter is the TP name sent by the remote TP.

Remotely Initiated Conversations on MPE XL

On MPE XL, the TP name sent by the remote TP is configured in the APPC subsystem branch of NMMGR. The TP name is associated with a

job file through configuration. When the HP 3000 receives the TP name from the remote TP, it does one of two things:

1. If the local TP is already active, and if it is configured to receive multiple allocate requests from remote TPs, LU 6.2 API waits for any current conversation with the local TP process to be deallocated, and then it allocates a conversation with the local TP process.
2. If the local TP is not active, or if it is configured to receive only one allocate request per execution instance, LU 6.2 API initiates a new local TP process by streaming the job file associated with the TP name. Then, it allocates a conversation with the local TP process.

A remotely initiated TP on MPE XL is configured to conduct either single or multiple conversations with the remote TP. A TP that conducts a single conversation must call `MCGetAllocate` only once; another instance of the TP will be started each time a remote TP requests a conversation. A TP that conducts multiple conversations can call `MCGetAllocate` many times; only one instance of it may be running at once. See the *APPC Subsystem on MPE XL Node Manager's Guide* for more information on TP configuration.

On MPE XL, a timer for the `MCGetAllocate` intrinsic may be configured. The timer prevents the local TP from being suspended indefinitely if no allocate request arrives from the remote TP. The timer is a value from 0 (no timer) to 480 minutes (8 hours), configured through NMMGR/XL. If your program calls `MCGetAllocate`, and no allocate request arrives from the remote TP before the timer expires, control is returned to your program. See the *APPC Subsystem on MPE XL Node Manager's Guide* for more information.

The *`LocalTPName`* passed in the `MCGetAllocate` intrinsic must be configured in the APPC subsystem branch of NMMGR. It must match the TP name sent by the remote TP, and it must match the *`LocalTPName`* passed in the `TPStarted` intrinsic.

NOTE

If you are migrating a TP to LU 6.2 API/XL from LU 6.2 API/V or from a version of LU 6.2 API/XL prior to the Node Type 2.1 version, you must change the *`LocalTPName`* parameter from an output parameter to an input parameter.

Figure 5-3 shows how a local TP is started when a conversation allocation request is received from a remote TP.

Figure 5-3 Remotely Initiated TP on the HP 3000

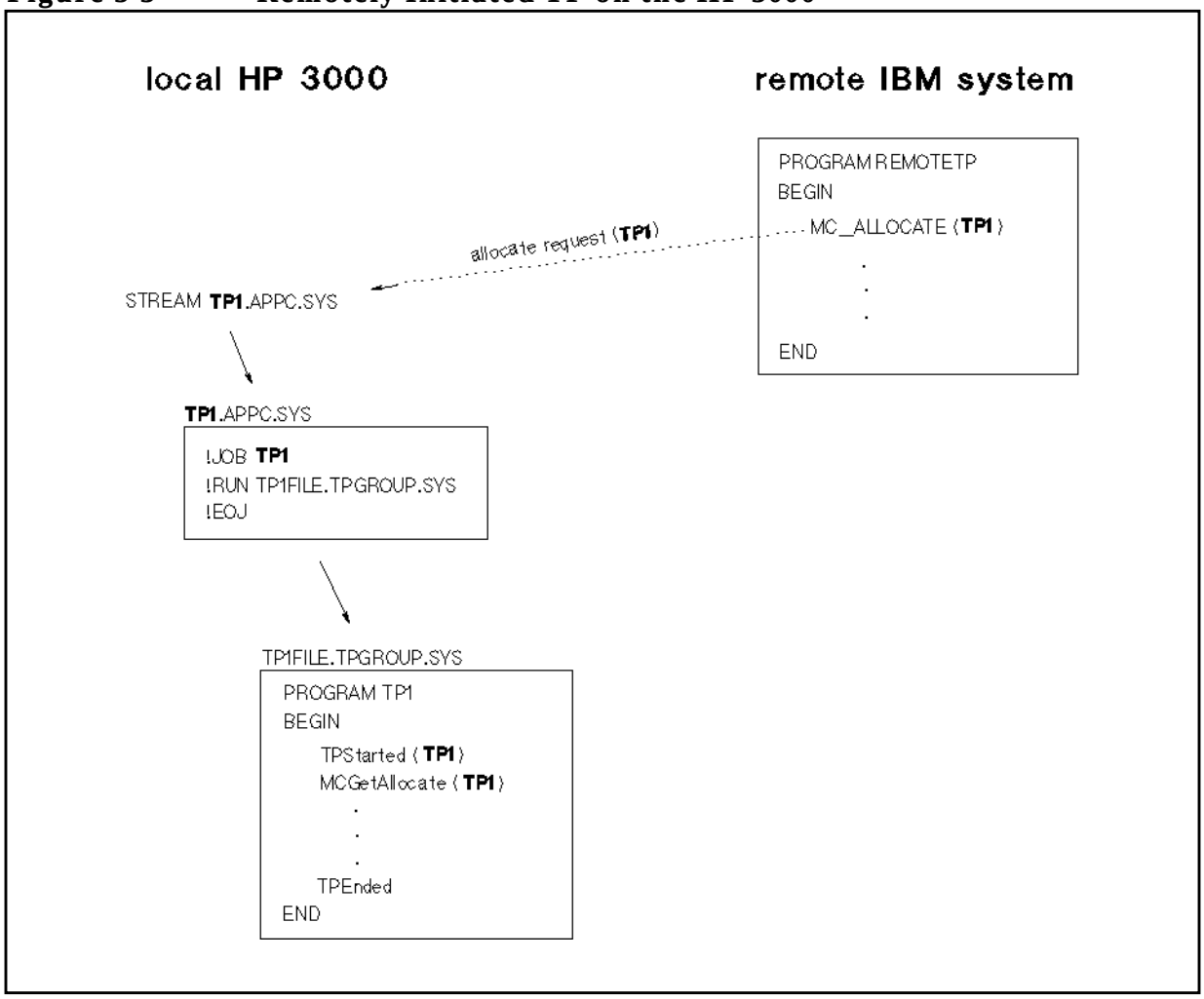
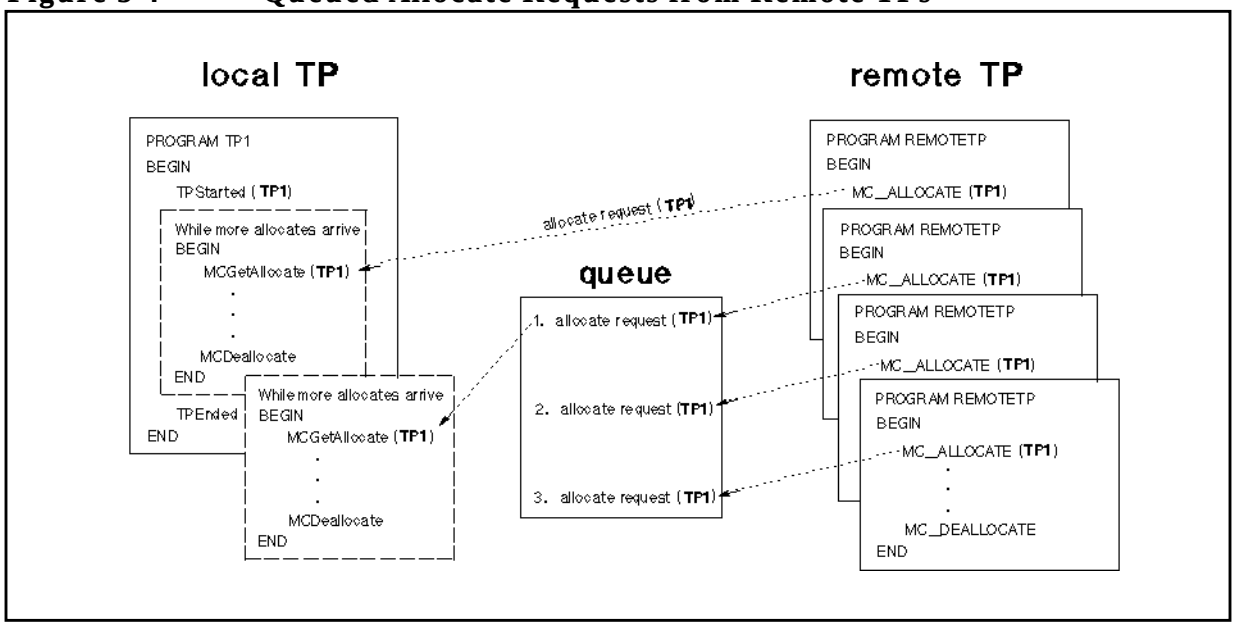


Figure 5-4 shows how, on MPE XL, allocate requests from the remote TP can be queued to wait until the local TP calls the `MCGetAllocate` intrinsic. The TP in Figure 5-4 loops through the same conversation, beginning with `MCGetAllocate` and ending with `MCDeallocate`, until it receives the last allocate request.

The TP could be written to handle a predetermined number of allocate requests, or it could loop through the conversation until the last `MCGetAllocate` call timed out because the queue was empty. When `MCGetAllocate` times out, +29 is returned in the *`Status`* parameter. (The `MCGetAllocate` time-out value must be configured through NMMGR/XL. See the *APPC Subsystem on MPE XL Node Manager's Guide*.)

Figure 5-4 Queued Allocate Requests from Remote TPs



Status Info Values

```
0    Successful Completion.
-1    Intrinsic called with parameter out of bounds.
-5    Out of range 'SyncLevel' parameter specified in intrinsic call.
-6    PIP data length is out of range.
-7    Out of range 'Timer' parameter specified in intrinsic call.
-15   Invalid 'TPID' parameter specified in intrinsic call.
-19   APPC subsystem is inactive.
-20   Not enough stack space for intrinsic to run.
-21   Insufficient memory space to allocate a conversation.
-22   Internal error: Unable to create Presentation Services port.
-23   Unable to allocate a conversation.
-24   Unable to obtain an LU-LU session.
-25   Could not find a conversation for this transaction program.
+29   Could not find a conversation for this transaction program -
       timer popped.  (MPE XL)
-51   Resource failure: No retry possible.
-65   Received an invalid attach from the remote LU.
-90   An internal error in Presentation Services has occurred.
-91   An internal error in the APPC subsystem has occurred.
-1002 An internal error at the mapped conversation level has occurred.
-1003 Required parameter missing.
-1005 Insufficient Heap Space.  (MPE V)
-1007 Out of range 'NumPIPs' parameter specified in intrinsic call.
-1008 Invalid 'LocalTPName' parameter specified in intrinsic call.
     (MPE XL)
-1009 Combined length of PIPs is out of range.
-1010 Too many PIP subfields.
```

MCGetAttr

Returns information pertaining to the specified conversation.

Syntax

```
              I16V       I32             CA
MCGetAttr(ResourceID, Status, [OwnFullyQualifiedLUName],

              CA                    CA
          [PartnerLUName], [PartnerFullyQualifiedLUName],


             CA         I16
          [ModeName], [SyncLevel]);
```

Parameters

ResourceID

Required; 16-bit signed integer by value; input. This is the unique resource ID number assigned to this conversation when it was allocated. See MCAllocate or MCGetAllocate, in this chapter, for more information.

Status

Required; 32-bit signed integer; output. Indicates the result of intrinsic execution. See the "Status Parameter" section, earlier in this chapter, for more information.

OwnFullyQualifiedLUName

character array; output. If the specified conversation is using a dependent LU to communicate with a Type 5 node (like an IBM mainframe), this parameter is not implemented and returns a 17-character array of blanks.

If the specified conversation is using an independent LU on MPE XL to communicate with a Type 2.1 node (like an IBM AS/400), this is a 17-character array that returns the fully qualified LU name of the local LU. It has the form *NetID.LUName*, where *NetID* and *LUName* are the names of the local SNA network and the local LU configured in the APPC subsystem configuration. See the *APPC Subsystem on MPE XL Node Manager's Guid*e for information on APPC subsystem configuration.

PartnerLUName

character array; output. This is an 8-character ASCII array, left justified and padded with blanks. It returns the name of the remote LU used by the remote TP. *PartnerLUName* is the name by which the local LU knows the remote LU. *PartnerLUName* is returned only on a conversation using a dependent LU to communicate with a Type 5 node (like an IBM mainframe).

PartnerFullyQualifiedLUName

character array; output. If the specified conversation is using a dependent LU to communicate with a Type 5 node (like an IBM mainframe), this parameter is not implemented and returns a 17-character array of blanks.

If the specified conversation is using an independent LU on MPE XL to communicate with a Type 2.1 node (like an IBM AS/400), this is a 17-character array that returns the fully qualified LU name of the LU used by the remote TP. It has the form *NetID.LUName*, where *NetID* is the name of the SNA network where the remote node resides, and *LUName* is the local name for the remote LU configured in the APPC subsystem configuration. See the *APPC Subsystem on MPE XL Node Manager's Guide* for information on APPC subsystem configuration. *PartnerFullyQualifiedLUName* is returned only on conversations using independent LUs.

ModeName

character array; output. This is an 8-character array, left justified and padded with blanks. It returns the mode name configured for the APPC session on which the conversation is allocated. This parameter matches the mode name configured for the *SessionType* specified in the call to MCAllocate or MCGetAllocate. For more information on mode configuration, see the *LU 6.2 API/V Node Manager's Guide* or the *APPC Subsystem on MPE XL Node Manager's Guide*.

SyncLevel

16-bit signed integer; output. The synchronization level (whether or not confirmation is used) established for this conversation. Possible values are as follows:

0 = CONFIRM

Denotes that the MCConfirm and MCConfirmed intrinsics can be called. It also means that the confirm request option of any intrinsic may be used.

2 = NONE

Denotes that no confirmation will be used. If the *SyncLevel* is 2, the MCConfirm and MCConfirmed intrinsics cannot be called during this conversation, nor can the confirm request option of any intrinsic be used during this conversation.

Description

This intrinsic allows the local TP to get information about the conversation specified in the *ResourceID* parameter.

Status Info Values

```
 0    Successful Completion.
-1    Intrinsic called with parameter out of bounds.
-2    Invalid 'ResourceID' parameter specified in intrinsic call.
-20   Not enough stack space for intrinsic to run.
-90   An internal error in Presentation Services has occurred.
-91   An internal error in the APPC subsystem has occurred.
-1002 An internal error at the mapped conversation level has occurred.
-1003 Required parameter missing.
```

MCPostOnRcpt

Allows the LU to check the contents of the receive buffer for the specified conversation.

Syntax

```
                I16V       I16V   CA    I32
MCPostOnRcpt(ResourceID, Length, Data, Status);
```

Parameters

ResourceID **Required**; 16-bit signed integer by value; input. This is the unique resource ID number assigned to this conversation when it was allocated. See MCAllocate or MCGetAllocate, in this chapter, for more information.

Length **Required**; 16-bit signed integer by value; input. This parameter specifies the minimum amount of data (in bytes) that must be received in the receive buffer before the TP is to be notified. If the logical record length is less than the number specified in this parameter, the TP will be notified when a complete logical record is received. A value of 0 or 1 in this parameter indicates that the TP is to be notified when any amount of data is received. The value of *Length* may not exceed 4092 bytes.

Data **Required**; character array; input. This is the character array into which data will be received after it arrives in the receive buffer. Its length must be greater than or equal to the *Length* parameter.

Status **Required**; 32-bit signed integer; output. Indicates the result of intrinsic execution. See the "Status Parameter" section, earlier in this chapter, for more information.

Description

The MCPostOnRcpt intrinsic causes LU 6.2 API to set up resources to check the contents of the receive buffer for the specified conversation. The MCTest or MCWait intrinsic can then be called to interrogate these resources and find out if any data has arrived for the conversation. See the descriptions of MCTest and MCWait, later in this chapter.

Once MCPostOnRcpt has executed successfully, posting is active. It remains active until something is received into the receive buffer or an

intrinsic is called that ends posting and releases the resources. The following intrinsics end posting:

```
MCDeallocate
MCRcvAndWait
MCRcvNoWait
MCSendError
```

When the MCTest or MCWait intrinsic indicates that something is waiting in the receive buffer, call MCRcvAndWait or MCRcvNoWait with the same *Data* parameter you used in the last call to MCPostOnRcpt.

The MCPostOnRcpt intrinsic can be called only from Receive state. It can be called many times during the execution of a TP. If the *Length* parameter is changed from one call to the next, the *Length* value from the last call will be used to determine the minimum amount of data that must arrive before the TP is notified.

Status Info Values

```
 0    Successful Completion.
-1    Intrinsic called with parameter out of bounds.
-2    Invalid 'ResourceID' parameter specified in intrinsic call.
-11   Out of range 'Length' parameter specified in intrinsic call.
-20   Not enough stack space for intrinsic to run.
-40   Intrinsic called in invalid state.
-90   An internal error in Presentation Services has occurred.
-91   An internal error in the APPC subsystem has occurred.
-1002 An internal error at the mapped conversation level has occurred.
-1003 Required parameter missing.
-1050 Invalid 'Data' parameter specified in intrinsic call.
```

MCPrepToRcv

Changes the conversation state of the local TP from Send to Receive, and changes the conversation state of the remote TP from Receive to Send.

Syntax

```
                I16V      I32          I16V          I16V
MCPrepToRcv(ResourceID, Status, [PrepToRcvType], [Locks]);
```

Parameters

ResourceID

Required; 16-bit signed integer by value; input. This is the unique resource ID number assigned to this conversation when it was allocated. See MCAllocate or MCGetAllocate, in this chapter, for more information.

Status

Required; 32-bit signed integer; output. Indicates the result of intrinsic execution. See the "Status Parameter" section, earlier in this chapter, for more information.

PrepToRcvType

16-bit signed integer by value; input. The type of MCPrepToRcv to be executed. Possible values are as follows:

0 = CONVERSATION_SYNC_LEVEL

Denotes that the MCPrepToRcv intrinsic will be executed with the synchronization level specified by the *SyncLevel* parameter of the MCAllocate or MCGetAllocate intrinsic. The *SyncLevel* parameter can specify synchronization levels of CONFIRM and NONE.

If the conversation was allocated with a *SyncLevel* of CONFIRM, then MCPrepToRcv is executed as if CONFIRM were given as the *MCPrepToRcvType*. See the discussion of CONFIRM *MCPrepToRcvType* = 2).

If the conversation was allocated with a *SyncLevel* of NONE, then MCPrepToRcv is executed as if FLUSH were given as the *MCPrepToRcvType*. See the discussion of FLUSH (*MCPrepToRcvType* = 1).

1 = FLUSH

Causes the local LU to empty its send buffer and the local TP to enter Receive state.

6 = CONFIRM

Causes the local LU to flush its send buffer and immediately send a confirmation request to the remote TP. The remote TP is then placed in Confirm Send state and must respond with positive confirmation before it can enter Send state. If the MCPrepToRcv intrinsic does not execute successfully, the state of the local TP is determined by the value returned in the *Status* parameter, as follows:

status info value	state of local TP
-50 Allocation Error	Deallocate state
-51 Resource Failure – No Retry	Deallocate state
-52 Resource Failure – Retry	Deallocate state
-60 Program Error	Receive state
+80 Timer Has Expired	Receive state
-1020 Deallocate Abend	Deallocate state

Default: 0 (CONVERSATION_SYNC_LEVEL)

Locks

16-bit signed integer by value; input. This parameter specifies when control is to be returned to the local TP after it calls MCPrepToRcv with a *PrepToRcvType* of CONFIRM (or CONVERSATION_SYNC_TYPE, if the conversation has a *SyncLevel* of CONFIRM). If MCPrepToRcv is called with a *PrepToRcvType* of FLUSH, the *Locks* parameter has no meaning and is ignored. Possible values for this parameter are as follows:

0 = SHORT

Control will be returned to the local TP as soon as the remote TP sends back an appropriate reply. The equivalent of any of the following LU 6.2 API intrinsics will generate an appropriate reply:

```
MCConfirmed
MCDeallocate (DeallocateType = ABEND)
MCSendError
```

1 = LONG

Control will not be returned to the local TP until information, such as data, is received from the remote TP following a positive confirmation response.

Default: 0 (SHORT)

Description

The MCPrepToRcv intrinsic flushes the local TP's send buffer, changes the conversation state of the local TP from Send to Receive, and changes the state of the remote TP from Receive to Send. No data can be received through this intrinsic. To receive data, you must call either MCRcvAndWait or MCRcvNoWait.

Status Info Values

```
0    Successful Completion.
-1    Intrinsic called with parameter out of bounds.
-2    Invalid 'ResourceID' parameter specified in intrinsic call.
-20   Not enough stack space for intrinsic to run.
-26   Out of range 'PrepToRcvType' parameter specified in intrinsic call.
-27   Out of range 'Locks' parameter specified in intrinsic call.
-40   Intrinsic called in invalid state.
-50   Allocation Error.
-51   Resource Failure:  Retry possible.
-52   Resource Failure:  No retry possible.
-56   Program Error: No data truncation has occurred.
-60   Program Error:  Data may have been purged.
+80   Timer has expired.
-90   An internal error in Presentation Services has occurred.
-91   An internal error in the APPC subsystem has occurred.
-1002 An internal error at the mapped conversation level has occurred.
-1003 Required parameter missing.
-1020 Deallocate Abend.
-1105 Internal Error: Conversation deallocated.
```

MCRcvAndWait

Waits for information to arrive on the specified conversation and then receives the information.

Syntax

```
                I16V        I16             I16
MCRcvAndWait(ResourceID, Length, RequestToSendReceived,

              CA       I16        I32
            Data, WhatReceived, Status);
```

Parameters

ResourceID

Required; 16-bit signed integer by value; input. This is the unique resource ID number assigned to this conversation when it was allocated. See MCAllocate or MCGetAllocate, in this chapter, for more information.

Length

Required; 16-bit signed integer by value; input/output.

Input:

The *Length* value that the local TP passes to the remote TP indicates the maximum amount of data, in bytes, that the local TP can receive in its *Data* parameter. The *Length* value must not exceed 4092 bytes on MPE V, or 32763 bytes on MPE XL. A *Length* of 0 means that only control or error information can be received (no data).

Output:

If data is received (*WhatReceived* = DATA_COMPLETE or DATA_INCOMPLETE), the *Length* value that the remote TP returns to the local TP is the actual length of the data received. If control information is received (*WhatReceived* = SEND, CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE), the remote TP does not change the value in the *Length* parameter, so it contains whatever was supplied as input.

RequestToSendReceived

Required; 16-bit signed integer; output. Indicates whether the remote TP has issued a RequestToSend. Possible values are as follows:

1 = YES

Indicates a RequestToSend has been received from the remote TP. The remote TP has issued the equivalent of the MCReqToSend intrinsic, requesting that the local TP enter Receive state and place the remote TP in Send state.

0 = NO

No RequestToSend has been received.

Data

Required; character array; output; EBCDIC. The character array into which the local TP will receive data sent by the remote TP. The length of the *Data* array must be greater than or equal to the value in the *Length* parameter.

If the data comes from an EBCDIC application, the local TP must convert it from EBCDIC to ASCII. The MPE CTRANSLATE intrinsic, or the NLTRANSLATE intrinsic on MPE XL, may be used.

WhatReceived

Required; 16-bit signed integer; output. If the value returned in the *Status* parameter is 0, then the *WhatReceived* parameter contains a value indicating the type of information received. Possible values are as follows:

1 = DATA_COMPLETE

Indicates that a complete data record, or the final portion of a data record, has been received. The *Length* parameter determines the amount of data that can be received in a single call to MCRcvAndWait. If a data record is larger than the value in the *Length* parameter, you must call MCRcvAndWait more than once to receive a complete record.

2 = DATA_INCOMPLETE

Indicates that less than a complete record has been received, and you must call MCRcvAndWait again to receive the next portion of it. Incomplete data records are received when the size of a record exceeds the value

in the *Length* parameter. When the final portion of a data record is received, the *WhatReceived* parameter returns 1 (DATA_COMPLETE).

4 = SEND

Indicates that the remote TP has issued the equivalent of MCPrepToRcv or MCRcvAndWait and has entered Receive state. The local TP is now in Send state and can issue only those intrinsics that are callable from Send state.

5 = CONFIRM

Indicates that the remote TP has issued the equivalent of MCConfirm, placing the local TP in Confirm state. Unless it detects an error, the local TP must respond with a call to MCConfirmed.

6 = CONFIRM_SEND

Indicates that the remote TP has issued the equivalent of MCPrepToRcv with a synchronization level of CONFIRM. The local TP is placed in Confirm Send state. Unless it detects an error, it must send a confirmation response to the remote and enter Send state.

7 = CONFIRM_DEALLOCATE

Indicates that the remote TP has issued the equivalent of MCDeallocate with a synchronization level of CONFIRM. The local TP is placed in Confirm Deallocate state. Unless it detects an error, it must send a confirmation response to the remote and call MCDeallocate with a *DeallocateType* of LOCAL. When the remote TP deallocates normally, a status info value of +100 is returned to the local TP.

Status

Required; 32-bit signed integer; output. Indicates the result of intrinsic execution. See the “Status Parameter” section, earlier in this chapter, for more information.

Description

The MCRcvAndWait intrinsic waits for information to arrive on the conversation specified in the *ResourceID* parameter, then it receives the information from the receive buffer into the *Data* parameter. MCRcvAndWait is used to receive data and control information.

NOTE A TP cannot receive both data and control information in the same call to MCRcvAndWait. If both data and control information have been received in the receive buffer, a TP must make separate calls to MCRcvAndWait for each one.

The only control information that can be received in the same MCRcvAndWait call with data is the *RequestToSendReceived* notification.

If posting has been set on the specified conversation, MCRcvAndWait ends posting.

The local TP does not have to call MCPrepToRcv to enter Receive state before it calls MCRcvAndWait. A TP can call MCRcvAndWait directly from Send state. The send buffer will be flushed, the remote TP will be placed in Send state, and the local TP will be placed in Receive state.

When logical records sent by the remote are larger than the receive buffer (4092 bytes on MPE V, or 32763 bytes on MPE XL), the local TP must call MCRcvAndWait more than once to receive each record.

NOTE If the local TP will be receiving data records larger than the receive buffer, it must allocate more than one location for storing data. Data received in the *Data* parameter during the first call to MCRcvAndWait will be overwritten during the second call unless it is moved to another location.

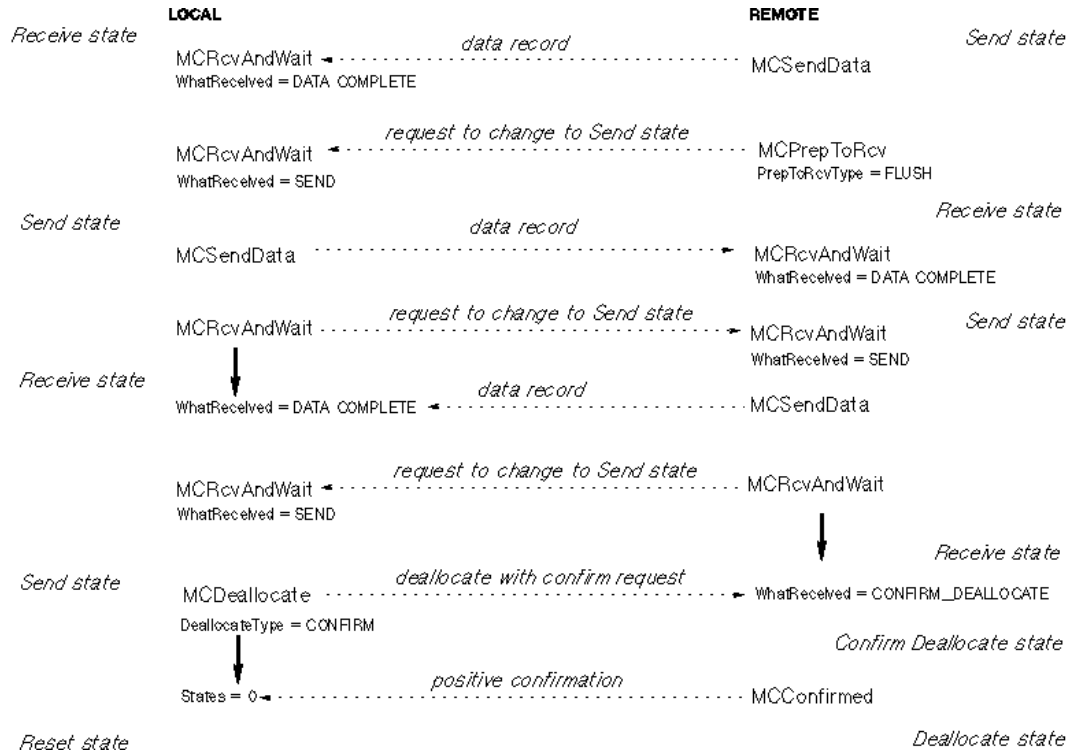
The MCRcvAndWait intrinsic differs from the MCRcvNoWait intrinsic in the following ways:

1. MCRcvAndWait can be used to change the conversation state of the local TP from Send state to Receive state. MCRcvNoWait can be called only from Receive state.
2. MCRcvAndWait waits for information from the remote before returning control to the calling TP. MCRcvNoWait does not wait for information to arrive. It checks to see if any information is available in the receive buffer, and if the buffer is empty, it returns a status info value of +38 (Not Posted) and returns control to the TP.
3. MCRcvAndWait can receive 4092 bytes on MPE V, or 32763 bytes on MPE XL, in a single intrinsic call. MCRcvNoWait can receive a maximum of 4092 bytes in a single intrinsic call, on MPE V or MPE XL.

Figure 5-5 shows a conversation in which both sides call MCRcvAndWait. Notice that, when MCRcvAndWait is called from Send state, the TP that called it is placed in Receive state and must wait for data to arrive before it can continue processing. The MCRcvAndWait intrinsic does not finish executing until information arrives for it to receive (or until the

Timer expires. See MCAllocate or MCGetAllocate for information on the *Timer*).

Figure 5-5 Conversation with Calls to MCRcvAndWait



Status Info Values

```
 0    Successful Completion.
-1    Intrinsic called with parameter out of bounds.
-2    Invalid 'ResourceID' parameter specified in intrinsic call.
-11   Out of range 'Length' parameter specified in intrinsic call.
-13   Data buffer specified in intrinsic call is out of bounds.
-20   Not enough stack space for intrinsic to run.
-40   Intrinsic called in invalid state.
-50   Allocation Error.
-51   Resource Failure:  Retry possible.
-52   Resource Failure:  No retry possible.
-56   Program Error:  No data truncation has occurred.
-60   Program Error:  Data may have been purged.
+80   Timer has expired.
-90   An internal error in Presentation Services has occurred.
-91   An internal error in the APPC subsystem has occurred.
+100  Deallocate Normal received from the remote TP.
-1002 An internal error at the mapped conversation level has occurred.
-1003 Required parameter missing.
-1020 Deallocate Abend.
-1050 Invalid 'Data' parameter specified in intrinsic call.
-1105 Internal Error: Conversation deallocated.
```

MCRcvNoWait

Receives any information available for the specified conversation but does not wait for information to arrive before returning control to the calling TP.

Syntax

```
                I16V        I16             I16
MCRcvNoWait(ResourceID, Length, RequestToSendReceived,

              CA       I16        I32
            Data, WhatReceived, Status);
```

Parameters

ResourceID

Required; 16-bit signed integer by value; input. This is the unique resource ID number assigned to this conversation when it was allocated. See MCAllocate or MCGetAllocate, in this chapter, for more information.

Length

Required; 16-bit signed integer by value; input/output.

Input:

The *Length* value that the local TP passes to the remote TP indicates the maximum amount of data, in bytes, that the local TP can receive in its *Data* parameter. The *Length* value must not exceed 4092 bytes. A *Length* of 0 means that only control or error information can be received (no data).

Output:

If data is received (*WhatReceived* = DATA_COMPLETE or DATA_INCOMPLETE), the *Length* value that the remote TP returns to the local TP is the actual length of the data received. If control information is received (*WhatReceived* = SEND, CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE), the remote TP does not change the value in the *Length* parameter, so it contains whatever was supplied as input.

RequestToSendReceived

Required; 16-bit signed integer; output. Indicates whether the remote TP has issued a RequestToSend. Possible values are as follows:

1 = YES

Indicates a RequestToSend has been received from the remote TP. The remote TP has issued the equivalent of the MCReqToSend intrinsic, requesting that the local TP enter Receive state and place the remote TP in Send state.

0 = NO

No RequestToSend has been received.

Data

Required; character array; output; EBCDIC. The character array into which the local TP will receive data sent by the remote TP. The length of the *Data* array must be greater than or equal to the value in the *Length* parameter.

If the data comes from an EBCDIC application, the local TP must convert it from EBCDIC to ASCII. The MPE CTRANSLATE intrinsic, or the NLTRANSLATE intrinsic on MPE XL, may be used.

WhatReceived

Required; 16-bit signed integer; output. If the value returned in the *Status* parameter is 0, then the *WhatReceived* parameter contains a value indicating the type of information received. Possible values are as follows:

1 = DATA_COMPLETE

Indicates that a complete data record, or the final portion of a data record, has been received. The *Length* parameter determines the amount of data that can be received in a single call to MCRcvAndWait. If a data record is larger than the value in the *Length* parameter, you must call MCRcvAndWait more than once to receive a complete record.

2 = DATA_INCOMPLETE

Indicates that less than a complete record has been received, and you must call MCRcvAndWait again to receive the next portion of it. Incomplete data records are received when the size of a record exceeds the value

in the *Length* parameter. When the final portion of a data record is received, the *WhatReceived* parameter returns 1 (DATA_COMPLETE).

4 = SEND

Indicates that the remote TP has issued the equivalent of MCPrepToRcv or MCRcvAndWait and has entered Receive state. The local TP is now in Send state and can issue only those intrinsics that are callable from Send state.

5 = CONFIRM

Indicates that the remote TP has issued the equivalent of MCConfirm, placing the local TP in Confirm state. Unless it detects an error, the local TP must respond with a call to MCConfirmed.

6 = CONFIRM_SEND

Indicates that the remote TP has issued the equivalent of MCPrepToRcv with a synchronization level of CONFIRM. The local TP is placed in Confirm Send state. Unless it detects an error, it must send a confirmation response to the remote and enter Send state.

7 = CONFIRM_DEALLOCATE

Indicates that the remote TP has issued the equivalent of MCDeallocate with a synchronization level of CONFIRM. The local TP is placed in Confirm Deallocate state. Unless it detects an error, it must send a confirmation response to the remote TP and call MCDeallocate with a DeallocateType of LOCAL. When the remote TP deallocates normally, a status info value of +100 is returned to the local TP.

Status

Required; 32-bit signed integer; output. Indicates the result of intrinsic execution. See the “Status Parameter” section, earlier in this chapter, for more information.

Description

The MCRcvNoWait intrinsic receives any data that has arrived in the receive buffer for the specified conversation. It differs from the MCRcvAndWait intrinsic in the following ways:

1. MCRcvNoWait can be called only from Receive state. Unlike MCRcvAndWait, it cannot be used to change the conversation state of the local TP from Send state to Receive state.
2. Unlike MCRcvAndWait, which waits for information from the remote before returning control to the calling TP, MCRcvNoWait does not wait for information to arrive. It checks to see if any information is available in the receive buffer, and if the buffer is empty, it returns a status info value of +38 (Not Posted) and returns control to the TP.
3. The MCRcvNoWait intrinsic can receive a maximum of 4092 bytes in a single intrinsic call. The MCRcvAndWait intrinsic can receive 4092 bytes on MPE V, or 32763 bytes on MPE XL, in a single intrinsic call.

MCRcvNoWait may not be used in conjunction with posting. Posting ends as soon as MCRcvNoWait is called. For more information on posting, see the description of MCPostOnRcpt, earlier in this chapter.

When logical records sent by the remote processor are larger than 4092 bytes, the local TP must call MCRcvNoWait more than once to receive each record.

NOTE If the local TP will be receiving data records larger than 4092 bytes, it must allocate more than one location for storing data. Data received in the *Data* parameter during the first call to MCRcvNoWait will be overwritten during the second call unless it is moved to another location.

Status Info Values

```
 0    Successful Completion.
-1    Intrinsic called with parameter out of bounds.
-2    Invalid 'ResourceID' parameter specified in intrinsic call.
-11   Out of range 'Length' parameter specified in intrinsic call.
-13   Data buffer specified in intrinsic call is out of bounds.
-20   Not enough stack space for intrinsic to run.
+38   Not Posted.
-40   Intrinsic called in invalid state.
-50   Allocation Error.
-51   Resource Failure:  Retry possible.
-52   Resource Failure:  No retry possible.
-56   Program Error:  No data truncation has occurred.
-60   Program Error:  Data may have been purged.
-90   An internal error in Presentation Services has occurred.
-91   An internal error in the APPC subsystem has occurred.
+100  Deallocate Normal received from the remote TP.
-1002 An internal error at the mapped conversation level has occurred.
-1003 Required parameter missing.
-1020 Deallocate Abend.
-1050 Invalid 'Data' parameter specified in intrinsic call.
-1105 Internal Error: Conversation deallocated.
```

MCReqToSend

Notifies the remote TP that the local TP wants to enter Send state.

Syntax

```
                I16V        I32
MCReqToSend(ResourceID, Status);
```

Parameters

ResourceID **Required**; 16-bit signed integer by value; input. This is the unique resource ID number assigned to this conversation when it was allocated. See MCAllocate or MCGetAllocate, in this chapter, for more information.

Status **Required**; 32-bit signed integer; output. Indicates the result of intrinsic execution. See the "Status Parameter" section, earlier in this chapter, for more information.

Description

The MCReqToSend intrinsic can be called from either Receive state or Confirm state. It does not cause a state change; it requests that the remote TP, which is in Send state and therefore controls the conversation, enter Receive state and place the local TP in Send state.

After calling MCReqToSend, the local TP must call MCRcvAndWait or MCRcvNoWait to find out whether the remote TP performed the requested state change. If the state change was successful, the *WhatReceived* parameter of the MCRcvAndWait or MCRcvNoWait intrinsic will return 4 (SEND). See the descriptions of MCRcvAndWait and MCRcvNoWait, earlier in this chapter.

Status Info Values

```
 0    Successful Completion.
-1    Intrinsic called with parameter out of bounds.
-2    Invalid 'ResourceID' parameter specified in intrinsic call.
-20   Not enough stack space for intrinsic to run.
-40   Intrinsic called in invalid state.
-51   Resource failure: No retry possible.
-52   Resource failure: Retry possible.
-90   An internal error in Presentation Services has occurred.
-91   An internal error in the APPC subsystem has occurred.
-1002 An internal error at the mapped conversation level has occurred.
-1003 Required parameter missing.
```

MCSendData

Sends one data record to the remote TP.

Syntax

```
                I16V       CA     I16V             I16
MCSendData(ResourceID, Data, Length, RequestToSendReceived,

                I32
              Status);
```

Parameters

ResourceID

Required; 16-bit signed integer by value; input. This is the unique resource ID number assigned to this conversation when it was allocated. See MCAllocate or MCGetAllocate, in this chapter, for more information.

Data

Required; character array; input; EBCDIC. This byte array holds the data that is to be sent to the remote TP. Data that is to be sent to an EBCDIC application must be translated from ASCII to EBCDIC. The MPE CTRANSLATE intrinsic, or the NLTRANSLATE intrinsic on MPE XL, may be used.

Length

Required; 16-bit signed integer by value; input. This is an integer from 0 through 32763 that specifies the length, in bytes, of the data record to be sent. If *Length* = 0, a null record is sent.

RequestToSendReceived

Required; 16-bit signed integer; output. Indicates whether the remote TP has issued a RequestToSend. Possible values are as follows:

1 = YES

Indicates a RequestToSend has been received from the remote TP. The remote TP has issued the equivalent of the MCReqToSend intrinsic, requesting that the local TP enter Receive state and place the remote TP in Send state.

0 = NO

No RequestToSend has been received.

Status

Required; 32-bit signed integer; output. Indicates the result of intrinsic execution. See the "Status Parameter" section, earlier in this chapter, for more information.

Description

The MCSendData intrinsic sends data to the remote TP. When it is called, the data in the *Data* parameter is moved to the send buffer. When the send buffer is full, or when an intrinsic is called that flushes the send buffer, all the data in the send buffer is transmitted to the remote TP.

On MPE V, the send buffer is 2044 bytes for every conversation. On MPE XL, the send buffer is the same size as the maximum RU size for the session. The maximum RU size is a configured value from 256 through 2048 associated with the session type. For more information on configuring RU sizes, see the *APPC Subsystem on MPE XL Node Manager's Guide*.

If you call MCSendData with a *Data* parameter smaller than the send buffer, the data might not be transmitted immediately to the remote TP. To empty the send buffer and transmit all data immediately to the remote TP, call the MCFlush intrinsic. See the description of MCFlush, earlier in this chapter. See Chapter 6 , "Buffer Management," for more information on the send and receive buffers.

Status Info Values

```
0    Successful Completion.
-1    Intrinsic called with parameter out of bounds.
-2    Invalid 'ResourceID' parameter specified in intrinsic call.
-11   Out of range 'Length' parameter specified in intrinsic call.
-13   Data buffer specified in intrinsic call is out of bounds.
-20   Not enough stack space for intrinsic to run.
-40   Intrinsic called in invalid state.
-50   Allocation Error.
-51   Resource Failure:  Retry possible.
-52   Resource Failure:  No retry possible.
-60   Program Error:  Data may have been purged.
-90   An internal error in Presentation Services has occurred.
-91   An internal error in the APPC subsystem has occurred.
-1002 An internal error at the mapped conversation level has occurred.
-1003 Required parameter missing.
-1020 Deallocate Abend.
-1105 Internal Error: Conversation deallocated.
```

MCSendError

Informs the remote TP that the local TP has detected an error in an application. Places the remote TP in Receive state and the local TP in Send state.

Syntax

```
                I16V              I16             I32
MCSendError(ResourceID, RequestToSendReceived, Status);
```

Parameters

ResourceID

Required; 16-bit signed integer by value; input. This is the unique resource ID number assigned to this conversation when it was allocated. See MCAllocate or MCGetAllocate, in this chapter, for more information.

RequestToSendReceived

Required; 16-bit signed integer; output. Indicates whether the remote TP has issued a RequestToSend. Possible values are as follows:

1 = YES

Indicates a RequestToSend has been received from the remote TP. The remote TP has issued the equivalent of the MCReqToSend intrinsic, requesting that the local TP enter Receive state and place the remote TP in Send state.

0 = NO

No RequestToSend has been received.

Status

Required; 32-bit signed integer; output. Indicates the result of intrinsic execution. See the “Status Parameter” section, earlier in this chapter, for more information.

Description

The MCSendError intrinsic informs the remote TP that the local TP has detected an application error and is unable to receive any further information. It can be called from any state except Deallocate state or Reset state. Successful execution of this intrinsic places the remote TP

in Receive state (if it is not already in Receive state) and the local TP in Send state (if it is not already in Send state).

The most common use of MCSendError is to respond negatively to a confirmation request.

A call to MCSendError ends posting. For more information on posting, see the description of MCPostOnRcpt, earlier in this chapter.

The MCSendError intrinsic operates differently depending on the current state of the TP that calls it. Following are descriptions of what MCSendError does in each conversation state:

Send state

When MCSendError is called from Send state, it flushes the send buffer.

Receive state

When MCSendError is called from Receive state, the states of the local and remote TPs are immediately reversed: The local TP is placed in Send state and the remote TP is placed in Receive state. All data in the receive buffer of the local TP is purged. However, if a RequestToSend indicator is waiting in the receive buffer, it is preserved for the next call to an intrinsic that uses the *RequestToSendReceived* parameter.

All Confirm states

When MCSendError is called from one of the Confirm states, all data from the remote TP has already been received, so the buffers are already empty. The local TP is placed in Send state, and the remote TP is placed in Receive state. The call to MCSendError serves as a negative response to the remote TP's confirmation request, so the request is no longer pending.

Status Info Values

```
 0    Successful Completion.
-1    Intrinsic called with parameter out of bounds.
-2    Invalid 'ResourceID' parameter specified in intrinsic call.
-20   Not enough stack space for intrinsic to run.
-40   Intrinsic called in invalid state.
-50   Allocation Error.
-51   Resource Failure:  Retry possible.
-52   Resource Failure:  No retry possible.
-56   Program Error:  No data truncation has occurred.
-60   Program Error:  Data may have been purged.
-90   An internal error in Presentation Services has occurred.
-91   An internal error in the APPC subsystem has occurred.
+100  Deallocate Normal received from the remote TP.
-1002 An internal error at the mapped conversation level has occurred.
-1003 Required parameter missing.
-1020 Deallocate Abend.
-1105 Internal Error: Conversation deallocated.
```

MCTest

Tests the specified for the receipt of information.

Syntax

```
          I16V      I16V    I32         I16
MCTest(ResourceID, [Test], Status, [PostedType]);
```

Parameters

ResourceID Required; 16-bit signed integer by value; input. This is the unique resource ID number assigned to this conversation when it was allocated. See MCAllocate or MCGetAllocate, in this chapter, for more information.

Test 16-bit signed integer by value; input. This parameter specifies whether to look for information waiting in the receive buffer or to check for a RequestToSend indicator sent from the remote TP.

0 = POSTED

Tests for information waiting in the receive buffer. The value returned in the *Status* parameter indicates the result of the test, as follows:

status info value	meaning
0	Something is waiting in the receive buffer.
-37	Posting is not active. (See MCPostOnRcpt.
+38	Nothing is waiting in the receive buffer.

1 = REQUEST_TO_SEND_RECEIVED

Tests for the receipt of a RequestToSend indicator from the remote TP. The value returned in the *Status* parameter indicates the result of the test, as follows:

status info value	meaning
0	A RequestToSend indicator has been received.
+36	No RequestToSend indicator has bee received.

Default: 0 (POSTED)

Status **Required**; 32-bit signed integer; output. Indicates the result of intrinsic execution. See the "Status Parameter" section, earlier in this chapter, for more information.

PostedType 16-bit signed integer; output. This parameter indicates the kind of information received. It is valid only when *Test* = 0 (POSTED) and *Status* = 0 (something is waiting in the receive buffer). Possible values are as follows:

0 = DATA

Indicates that the receive buffer contains at least the amount of data specified in the *Length* parameter of the MCPostOnRcpt intrinsic.

1 = NOT_DATA

Indicates that the receive buffer contains control information, not data.

Description

The MCTest intrinsic tests the receive buffer of the specified conversation to see whether any information has been received and what kind of information it is (data or control information). The MCTest intrinsic can also indicate whether the remote TP has issued the equivalent of MCReqToSend, requesting that the local TP enter Receive state and place the remote TP in Send state.

Before calling MCTest, you must call MCPostOnRcpt to set up the resources necessary to check the contents of the receive buffer. See the description of MCPostOnRcpt, earlier in this chapter.

The MCWait intrinsic, described later in this chapter, is similar to the MCTest intrinsic; however, MCTest and MCWait differ in the following ways:

1. The MCTest intrinsic can test only one conversation at a time, while the MCWait intrinsic can monitor the receive buffers of many conversations at once.
2. The MCWait intrinsic can return only information about the contents of the receive buffer and, unlike the MCTest intrinsic, cannot check whether a RequestToSend has been received from the remote TP.
3. The MCTest intrinsic does not wait for information to arrive. After it tests for the specified information, no matter what it finds, it returns control to the conversation. The MCWait intrinsic waits until information arrives in the receive buffer of one of the conversations it is monitoring before it returns control to the conversation.

Status Info Values

```
 0    Successful Completion.
-1    Intrinsic called with parameter out of bounds.
-2    Invalid 'ResourceID' parameter specified in intrinsic call.
-20   Not enough stack space for intrinsic to run.
-35   Out of range 'Test' parameter specified in intrinsic call.
+36   Request To Send Not Received.
-37   Posting Not Active.
+38   Not Posted.
-40   Intrinsic called in invalid state.
-50   Allocation Error.
-51   Resource Failure:  Retry possible.
-52   Resource Failure:  No retry possible.
-56   Program Error:  No data truncation has occurred.
-60   Program Error:  Data may have been purged.
-90   An internal error in Presentation Services has occurred.
-91   An internal error in the APPC subsystem has occurred.
+100  Deallocate Normal received from the remote TP.
-1002 An internal error at the mapped conversation level has occurred.
-1003 Required parameter missing.
-1020 Deallocate Abend.
-1105 Internal Error: Conversation deallocated.
```

MCWait

Waits for information to arrive for any of a list of conversations.

Syntax

```
          IA16           I16V           I16
MCWait(ResourceList, NumResources, ResourcePosted,

          I32       I16
       Status, [PostedType]);
```

Parameters

ResourceList

Required; 16-bit integer array; input. This array is a list of *ResourceIDs* for current conversations. It specifies the conversations that MCWait will monitor for incoming information. At least one of the conversations specified must have posting active; that is, MCPostOnRcpt must have been called. MCWait can monitor only conversations for which posting is active.

On MPE V, up to 8 *ResourceIDs* can be listed in the *ResourceList*, because MPE V supports a maximum of 8 active conversations.

On MPE XL, up to 256 *ResourceIDs* can be listed in the *ResourceList*, because MPE XL supports a maximum of 256 active conversations.

NumResources

Required; 16-bit signed integer by value; input. This parameter specifies the number of *ResourceIDs* listed in the *ResourceList* array. On MPE V, *NumResources* must be from 1 through 8. On MPE XL, *NumResources* must be from 1 through 256.

ResourcePosted

Required; 16-bit signed integer; output. When the *Status* parameter returns 0, the *ResourcePosted* parameter contains the *ResourceID* of the conversation for which information has arrived.

Status

Required; 32-bit signed integer; output. Indicates the result of intrinsic execution. See the "Status Parameter" section, earlier in this chapter, for more information.

PostedType

16-bit signed integer; output. This parameter indicates the kind of information received. It is valid only when *Status* = 0 (something is waiting in the receive buffer for one of the posted conversations). Possible values are as follows:

0 = DATA

Indicates that the receive buffer contains at least the amount of data specified in the *Length* parameter of the MCPostOnRcpt intrinsic.

1 = NOT_DATA

Indicates that the receive buffer contains control information, not data.

Description

The MCWait intrinsic waits for information to arrive in the receive buffer of any in a list of active conversations. When it executes successfully, it returns the *ResourceID* of the conversation for which information has arrived. It may also indicate the type of information that has arrived: control information or data.

Before calling MCWait, you must call MCPostOnRcpt for each of the conversations you want to monitor, to set up the resources necessary to check the contents of their receive buffers. See the description of MCPostOnRcpt, earlier in this chapter.

The MCTest intrinsic, described earlier in this chapter, is similar to the MCWait intrinsic; however, MCTest and MCWait differ in the following ways:

1. The MCTest intrinsic can test only one conversation at a time, while the MCWait intrinsic can monitor the receive buffers of many conversations at once.
2. The MCWait intrinsic can return only information about the contents of the receive buffer, while the MCTest intrinsic can check whether a RequestToSend has been received from the remote TP.
3. The MCTest intrinsic does not wait for information to arrive. After it tests for the specified information, no matter what it finds, it returns control to the conversation. The MCWait intrinsic waits until

information arrives in the receive buffer of one of the conversations it is monitoring before it returns control to the conversation.

Status Info Values

```
 0    Successful Completion.
-1    Intrinsic called with parameter out of bounds.
-2    Invalid 'ResourceID' parameter specified in intrinsic call.
-20   Not enough stack space for intrinsic to run.
-34   Out of range 'NumResources' parameter specified in intrinsic call.
-37   Posting Not Active.
-40   Intrinsic called in invalid state.
-50   Allocation Error.
-51   Resource Failure:  Retry possible.
-52   Resource Failure:  No retry possible.
-56   Program Error:  No data truncation has occurred.
-60   Program Error:  Data may have been purged.
+80   Timer has expired.
-90   An internal error in Presentation Services has occurred.
-91   An internal error in the APPC subsystem has occurred.
+100  Deallocate Normal received from the remote TP.
-1002 An internal error at the mapped conversation level has occurred.
-1003 Required parameter missing.
-1020 Deallocate Abend.
-1105 Internal Error: Conversation deallocated.
```

6 Buffer Management

LU 6.2 API optimizes the use of the data communications line by buffering the data that is sent and received. This chapter explains how LU 6.2 API handles control information and how it uses the send and receive buffers to manage the data traffic between transaction programs.

Control Information

For every conversation allocated, LU 6.2 API establishes and maintains a set of flags or indicators that keep track of certain control information necessary to manage the conversation. Whenever an intrinsic is called, LU 6.2 API checks these flags before executing the intrinsic to see if any information must be relayed to the TP. The following flags are associated with each conversation:

- **Error flag**. The error flag is used to inform the TP that some type of error has occurred.
- **RequestToSendReceived flag**. The `RequestToSendReceived` flag tells the TP whether a `RequestToSend` notification has been received from the remote TP. A `RequestToSend` is the only control information that can be received in the same intrinsic call with data.
- **State indicator**. The state indicator keeps track of the conversation state of the local TP. If the state changes, the state indicator changes. When an intrinsic is called, the state indicator is checked to ensure that the intrinsic can be called from the current state.
- **Synchronization level indicator**. The synchronization level indicator records the synchronization level that is established when a conversation is allocated. If an intrinsic is called that can be executed only at a certain synchronization level, the synchronization level indicator is checked to verify that the conversation was allocated with the appropriate synchronization level.

Send Buffer

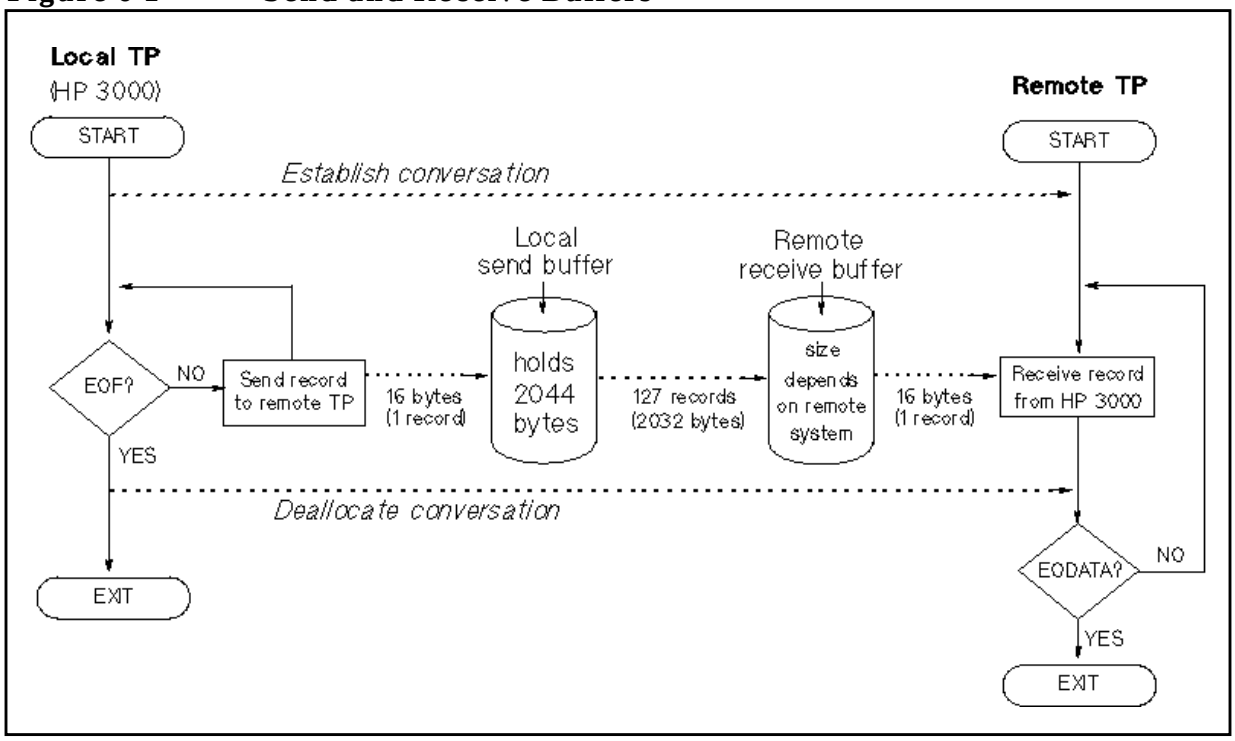
For each conversation allocated, LU 6.2 API establishes a send buffer. On MPE V, the send buffer is always 2044 bytes. On MPE XL, the send buffer is the same size as the maximum RU size for the session. The maximum RU size is a configured value from 256 through 2048 associated with the session type. For more information on configuring RU sizes, see the *APPC Subsystem on MPE XL Node Manager's Guide*.

Whenever the `MCSendData` intrinsic is called, data is transferred to the send buffer from the *`Data`* parameter specified in the intrinsic call. When the send buffer is full, LU 6.2 API flushes the buffer and transmits the data to the remote TP. No data is transmitted until the send buffer is full or until an intrinsic is called that flushes the buffer.

Example 1: Sending Small Data Records

Figure 6-1 shows how data flows from the send buffer of the local TP to the receive buffer of the remote TP in a one-way file transfer application. This example application sends 16-byte data records. The send buffer in this example holds 2044 bytes.

Figure 6-1 Send and Receive Buffers



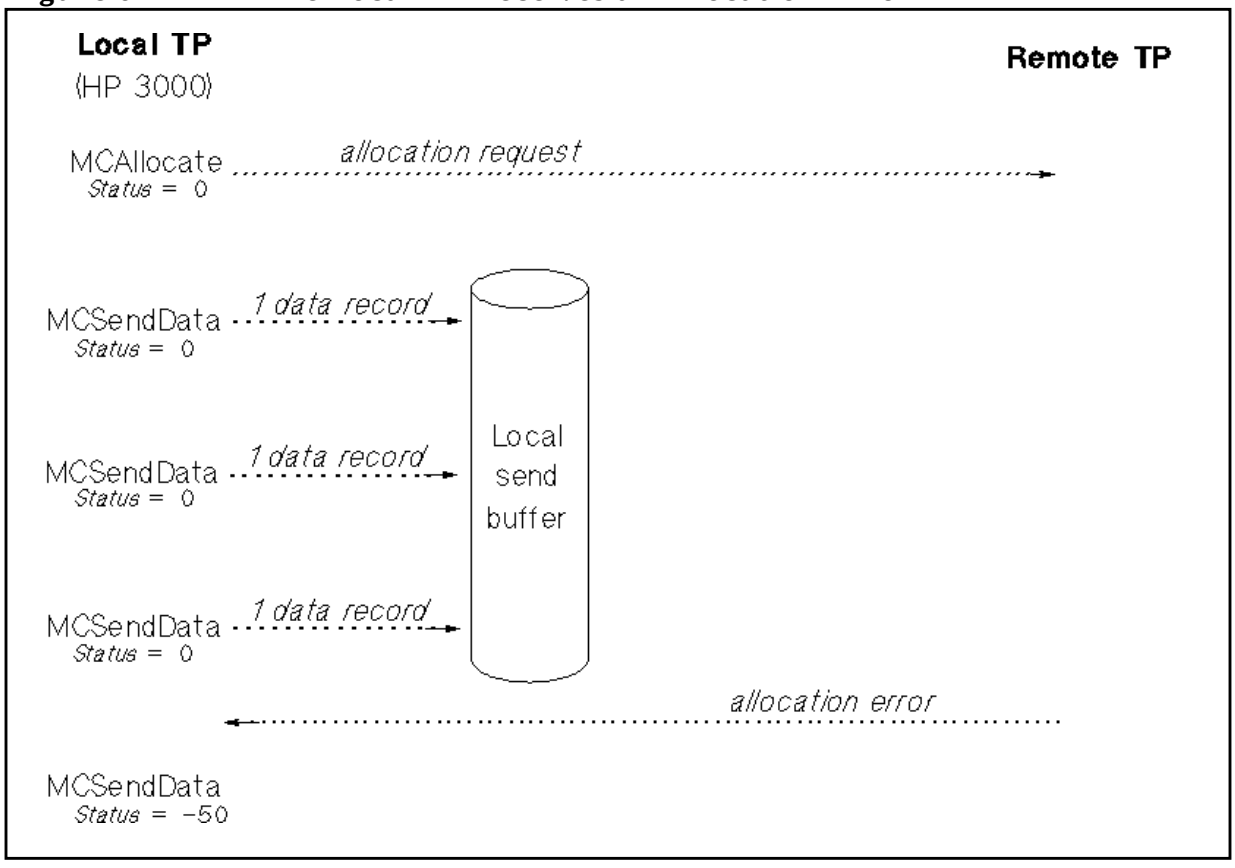
Because each data record in figure 6-1 is only 16 bytes long, LU 6.2 API can store 127 data records in the 2044-byte send buffer (127 records of 16 bytes = 2032 bytes). When `MCSendData` is called for the 128th time,

LU 6.2 API checks to see if 16 more bytes will fit in the send buffer. Only 12 more bytes will fit, so LU 6.2 API transmits the 127 records in the send buffer and then stores the 128th record in the send buffer.

Example 2: An Allocation Error

In example 2, a local TP receives an allocation error from the remote TP. After executing the `MCAllocate` intrinsic, LU 6.2 API does not wait for a response from the remote TP before it starts executing calls to `MCSendData`. If the conversation could not be allocated on the remote side, the allocation error could be received on any of the calls to `MCSendData`. In Figure 6-2, the allocation error does not reach the local TP until the local TP has made three calls to `MCSendData`. The TP is informed of the error through the *`Status`* parameter in the fourth call to `MCSendData`.

Figure 6-2 The Local TP Receives an Allocation Error



To verify that a conversation is allocated successfully, call `MCAllocate` with the *`SyncLevel`* parameter set to 0 (CONFIRM), and then call `MCConfirm`. `MCConfirm` flushes the send buffer and requests that the remote TP send a reply, confirming that the conversation was allocated successfully.

Receive Buffer

Unless the data records exchanged by TPs are very large, they are transmitted and received in groups of more than one at a time. However, a TP can logically send and receive only one record at a time. So each group of records that a TP receives is stored in its receive buffer, and the TP takes the records out of the receive buffer one at a time.

LU 6.2 API allocates a receive buffer for every active conversation. On MPE V, the size of the receive buffer is 4092 bytes. On MPE XL, the receive buffer is a different size depending on which intrinsic you call to receive data. On MPE XL, the `MCRcvAndWait` intrinsic uses a 32763-byte receive buffer, and the `MCRcvNoWait` intrinsic uses a 4092-byte receive buffer.

Because `MCRcvAndWait` and `MCRcvNoWait` can receive control information as well as data, an additional area is assigned with the receive buffer. This area contains the state the TP is to enter once it receives all the data from the receive buffer.

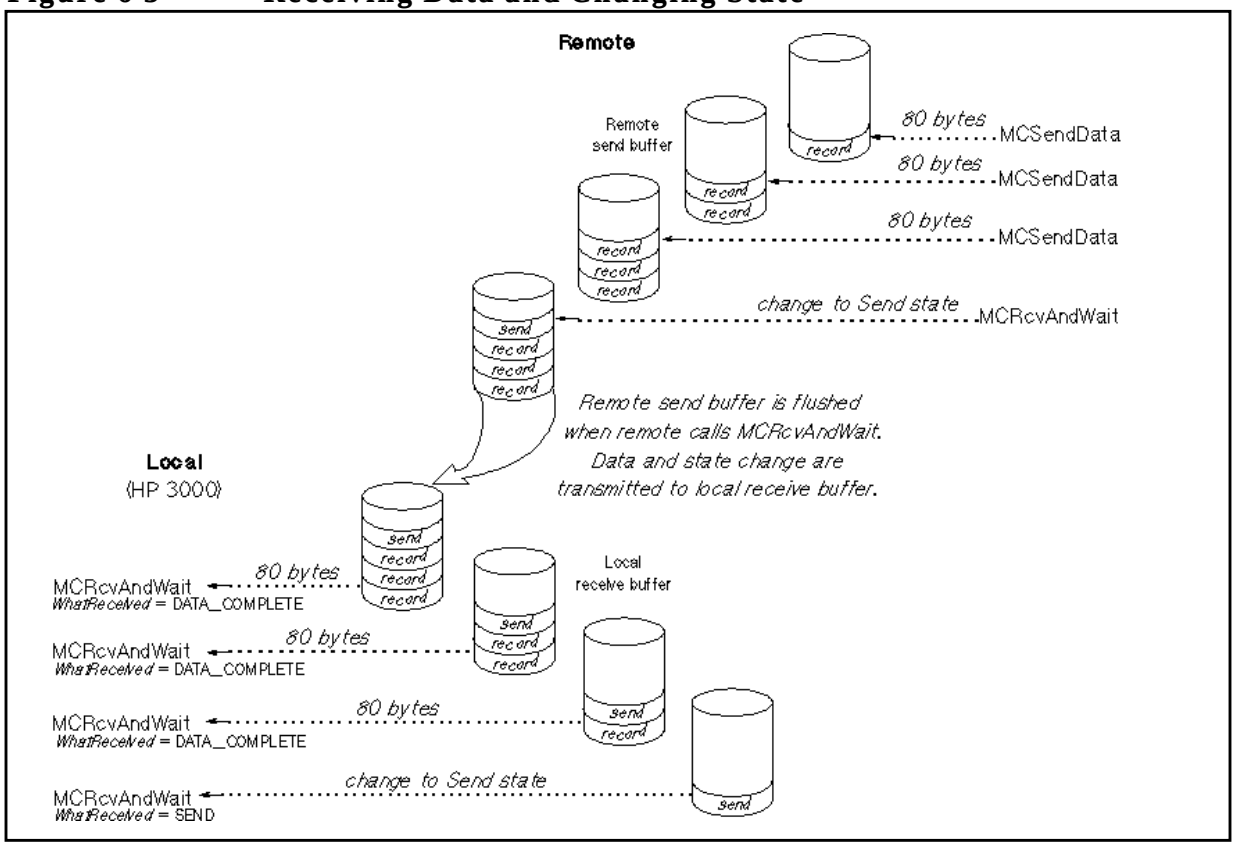
NOTE Hewlett-Packard's implementation of LU 6.2 API does not allow data and control information to be received on the same intrinsic call. Other LU 6.2 implementations may allow data and control information to be received at the same time. This could affect how the TP must be designed at the remote location.

Example 3: Receiving Data and Changing State

In example 3, the remote TP is sending 80-byte data records during a two-way conversation without confirm. When the remote TP is finished sending data, it issues the equivalent of the `MCRcvAndWait` intrinsic, which flushes its send buffer and tells the local TP to enter Send state.

The data and the state change instruction are transmitted to the local receive buffer. The local TP receives the data from the receive buffer through calls to `MCRcvAndWait` or `MCRcvNoWait`. After the local TP receives all the data, one record at a time, it calls `MCRcvAndWait` once more to receive the instruction to change to Send state. When the `MCRcvAndWait` intrinsic returns, the state indicator flag is updated to reflect the new state. Figure 6-3 illustrates this example.

Figure 6-3 Receiving Data and Changing State



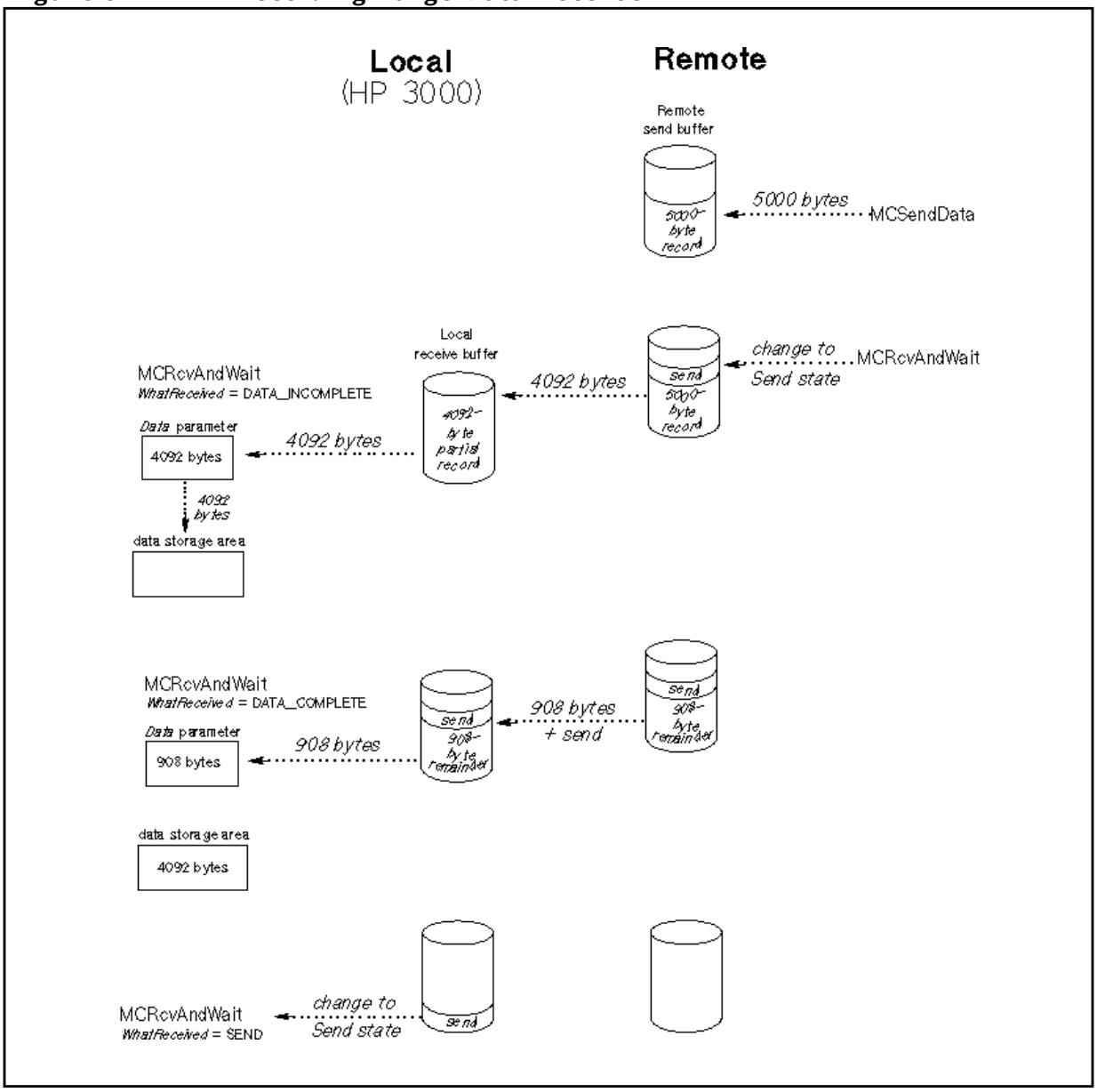
Example 4: Receiving Large Data Records

In example 4, the remote TP has a large send buffer, and the data records that it sends to the local TP are 5000 bytes long. The local receive buffer will hold 4092 bytes. It is too small to receive a complete data record, so the local TP must call `MCRcvAndWait` or `MCRcvNoWait` twice to receive a complete record: once to receive the first 4092 bytes, and a second time to receive the remaining 908 bytes of the 5000-byte record.

When the local TP calls `MCRcvAndWait` or `MCRcvNoWait` the first time, the *`WhatReceived`* parameter returns DATA_INCOMPLETE, telling the local TP that it has received an incomplete data record and must make another intrinsic call to receive the rest of the record. When the local TP receives the last 908 bytes, the *`WhatReceived`* parameter returns DATA_COMPLETE, telling the local TP that it has received the end of a data record. Figure 6-4 illustrates this example.

NOTE If the local TP will be receiving data records larger than the receive buffer, it must allocate more than one location for storing data. Data received in the *`Data`* parameter during the first call to `MCRcvAndWait` or `MCRcvNoWait` will be overwritten during the second call unless it is moved to another location. (See Figure 6-4.)

Figure 6-4 Receiving Large Data Records



7 Debugging

This chapter shows you how to isolate problems in your TPs. It contains the following sections:

- Debugging Steps. This section tells you the steps to follow to minimize the time you spend debugging.
- The User Trace. This section explains how to enable user tracing, how to identify and format the trace files, and how to read a user trace.

Debugging Steps

To minimize the time spent isolating TP problems, follow the procedure below.

1. Format and analyze the user trace. The user trace is the most useful point of departure when debugging. If you have not enabled user tracing, you will have to recreate the problem with user tracing enabled. See the description of the `TPStarted` intrinsic, in Chapter 5 , “Intrinsic Descriptions,” for more information on enabling user tracing.

   Scan the user trace for status info values other than 0 (successful completion). Determine the meaning of any non-zero status info values.

2. If you cannot find any problems with your code, verify that the remote TP name passed through `MCAllocate` is the correct name for the remote TP that will communicate with the local TP.

3. If you do not find a problem with the local TP, and you have verified that the local TP is communicating with the correct remote TP, talk with the remote programmer to determine whether the problem lies with the remote TP. Many problems occur because the local and remote programs are out of synchronization.

The User Trace

User tracing records intrinsic calls and data for a single transaction program, including all conversations in which the transaction program is engaged. Tracing can be turned on using the *TraceOn* parameter of the TPStarted intrinsic. Once the trace has been turned on, it remains on until TPEnded is called. Entries are written to the trace file at the completion of each intrinsic call. See Chapter 5 , "Intrinsic Descriptions," for descriptions of the TPStarted and TPEnded intrinsics.

User tracing should be enabled during program development so that the trace will always be available for debugging. It is common practice to disable user tracing after a program has been debugged. However, if a problem arises after the program is in use, enabling tracing again may help to diagnose the problem.

You can enable or disable user tracing without recompiling the program by creating a control file that passes the desired tracing value to the TPStarted intrinsic, or by coding your program to accept an info string from the MPE RUN command that contains the *TraceOn* value.

NOTE

Develop each TP in a separate group on the HP 3000 to avoid trace file confusion.

You can specify a name for your trace file in the *TraceFile* parameter of the TPStarted intrinsic, or you can allow the trace file name to default.

If you specify a name for the trace file, the file will be overwritten each time the program is run with tracing enabled. Separate processes of the same TP will try to open and overwrite the same trace file. The first process started will overwrite the old trace file, but subsequent processes will be unable to start, because the trace file they need to open is in use.

If you use the default name, each TP process will open a separate trace file. You can use the *DefaultFile* parameter of the TPStarted intrinsic to find out which trace file is associated with which process, or you can distinguish between trace files of different processes or different TPs by reading the time stamp at the beginning of the trace.

Collecting the User Trace

The user trace file is created in the group and account from which the TP is run. The default name is PSTRAC*nn*, where *nn* is a number from 00 through 49 that increments each time a new file is opened.

After 25 user trace files are created, the system purges one trace file each time a new one is opened. Trace files are created and numbered as follows:

1. Trace files PSTRAC00 through PSTRAC24 are created.
2. PSTRAC25 is created and file PSTRAC00 is purged.
3. The next trace file created is PSTRAC26. When PSTRAC26 is created, PSTRAC01 is purged.
4. When PSTRAC27 is created, PSTRAC02 is purged, and so on.
5. After trace file PSTRAC49 is created, the numbering wraps and the next file created is PSTRAC00.

There are never more than twenty-five trace files in the group and account where the TP is running, because, after the twenty-fifth file is created, an old file is purged for each new one created. Because the numbering wraps to 00 after it reaches 49, the youngest trace file could be numbered either low or high. The key to finding the youngest file is determining whether the file numbers have wrapped.

If the file numbers have not wrapped, the numbering will be contiguous; that is, no gaps will exist between numbers. In the following example, PSTRAC00 to PSTRAC21 have been purged, and PSTRAC47 to PSTRAC49 do not exist. Therefore, the files have not wrapped, and the file with the highest number is the youngest.

```
:listf pstrac@.testpgms.api,2
<F100P12>ACCOUNT=   API        GROUP=   TESTPGMS

FILENAME  CODE  ------------LOGICAL RECORD-----------  ----SPACE----

                  SIZE  TYP        EOF      LIMIT R/B  SECTORS #X MX

PSTRAC22  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC23  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC24  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC25  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC26  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC27  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC28  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC29  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC30  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC31  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC32  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC33  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC34  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC35  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC36  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC37  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC38  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC39  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC40  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC41  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC42  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC43  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC44  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC45  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC46  NTRAC   128W  FB        1000       1000   8     1008 16 16

:
```

The youngest trace file is PSTRACE46, because the file numbers are contiguous, and 46 is the highest number.

If the file numbers have wrapped, the file with the highest number is not the youngest. In the following example, notice that the numbering jumps from PSTRAC02 to PSTRAC28. There is a gap of 26 file numbers after PSTRAC02. The youngest file will always be at the lower end of the gap. In this example, the youngest file is PSTRAC02. The files PSTRAC03 through PSTRAC27 have been purged, and the numbering has wrapped.

```
:listf pstrac@.testpgms.api,2
<F100P12>ACCOUNT=   TESTPGMS      GROUP=   API

FILENAME  CODE  ------------LOGICAL RECORD-----------  ----SPACE----

                  SIZE  TYP        EOF      LIMIT R/B  SECTORS #X MX

PSTRAC00  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC01  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC02  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC28  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC29  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC30  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC31  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC32  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC33  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC34  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC35  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC36  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC37  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC38  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC39  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC40  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC41  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC42  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC43  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC44  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC45  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC46  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC47  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC48  NTRAC   128W  FB        1000       1000   8     1008 16 16
PSTRAC49  NTRAC   128W  FB        1000       1000   8     1008 16 16

:
```

NOTE When you delete some of the trace files from a group, you no longer have a complete set, and finding the youngest file becomes difficult. If you must clean up trace files, copy the files that you need to another group and purge all the trace files at once.

Formatting the User Trace

The trace file must be formatted before you can read it. On MPE V, use the utility APPCDUMP.APPC.SYS to format the user trace. APPCDUMP requires that you specify an input file and an output file. The syntax for running APPCDUMP is as follows:

```
:FILE TRACEIN  = trace file name
:FILE TRACEOUT = { $STDLIST } (terminal)
                 { *LP }      (printer)
                 { file designator }

:RUN APPCDUMP.APPC.SYS
```

On MPE XL, use the NMS utility NMDUMP.PUB.SYS to format the user trace. To run NMDUMP, type the following command at the MPE colon prompt:

```
:RUN NMDUMP.PUB.SYS
```

NMDUMP will respond with a menu. You use the menu to indicate the type of file you want to format and which subsystem generated the file. NMDUMP will prompt you for an input file and an output file. The default output file is $STDLIST. You can use a file equation to back-reference the name of an input or output file. For more information on NMDUMP, see *Using the Node Management Services (NMS) Utilities*.

Reading the User Trace

The user trace records information for each intrinsic the local TP calls. This section explains how to read the information in the user trace.

Figure 7-1 is an example trace of a two-way conversation.

Figure 7-1 User Trace of a Two-Way Conversation

```
NMDUMP output of data file:  PSTRAC26.USERTPS.API
Time of output:  MON, SEP 16, 1991,  2:27 PM
Subsystems being formatted:  16

*******************************************************************************
TPSTARTED               RetCode:  0

                        TPID           = 14
                        TPName         = LOCALTP
                        TraceSize      = 1024
                        TraceFile      = PSTRAC26.USERTPS.API

*******************************************************************************
MCALLOCATE              RetCode:  0  Rsrc: 15  ConvState: SEND

                        TPID           = 14
                        SessionType    = STYPE01
                        ReturnControl  = WHEN_SESSION_ALLOCATED
                        SyncLevel      = CONFIRM
                        Timer          = 0
                        TPNameLength   = 8
                        TPName         = D9C5D4D6E3C5E3D7

*******************************************************************************
MCSENDDATA              RetCode:  0  Rsrc: 15  ConvState: SEND

                        RTS            = NO
                        Length         = 12
                        Offset         = 328.4033BDA8

    E2 C5 D5 C4 C9 D5 C7 40 C4 C1 E3 C1                    SENDING DATA

*******************************************************************************
MCPREPTORCV             RetCode:  0  Rsrc: 15  ConvState: RECEIVE

                        Locks          = SHORT
                        Type           = FLUSH

*******************************************************************************
MCRCVANDWAIT            RetCode:  0  Rsrc: 15  ConvState: RECEIVE

                        RTS            = NO
                        Length         = 14
                        Offset         = 328.4056DE96
                        WhatRcvd       = DATA_COMPLETE

    D9 C5 C3 C5 C9 E5 C9 D5 C7 40 C4 C1 E3 C1              RECEIVING DATA<$&figure 7-1>
@COMPUTERSMALL =
*******************************************************************************
MCRCVANDWAIT            RetCode:  0  Rsrc: 15  ConvState: CONF_DEALL

                        RTS            = NO
                        Length         = 256
                        Offset         = 328.4056DE96
                        WhatRcvd       = CONFIRM_DEALLOCATE

*******************************************************************************

MCCONFIRMED             RetCode:  0  Rsrc: 15  ConvState: DEALLOCATE

*******************************************************************************

MCDEALLOCATE            RetCode:  0  Rsrc: 15

                        DeallocateType  = LOCAL

*******************************************************************************

TPENDED                 RetCode:  0

                        TPID           = 14

*******************************************************************************
```

Field Descriptions

Figure 7-1 is an example of a trace of a two-way conversation. This section explains the meaning of each field in the user trace.

Intrinsic name

The name of each LU 6.2 API intrinsic the local TP calls appears on the left side of the display.

RetCode

(Return Code) This is the status info value returned in the *Status* parameter of the intrinsic. The status info is the most useful piece of diagnostic information in the user trace. It directs you to areas of investigation that help you identify problems with your TP. Chapter 5 , "Intrinsic Descriptions," lists the values that each intrinsic can return, and Appendix A , "Status Info," lists the causes and recommended actions for all status info values.

Rsrc

(Resource ID) This is the conversation identifier returned by `MCAllocate` or `MCGetAllocate`. It identifies the conversation for which each intrinsic was called. In the example, only one conversation (with Resource ID of 15) was allocated, so all intrinsics were called for that conversation. If more than one conversation had been allocated (that is, if `MCAllocate` had been called more than once by the TP), the different Resource IDs would tell you which intrinsic calls belonged to which conversations.

ConvState

This is the conversation state of the local TP after the intrinsic has executed.

Intrinsic parameters

The rest of the fields in the trace are the parameters specific to each intrinsic. Different parameters are traced for different intrinsics. The parameters are listed here by intrinsic. See Chapter 5 , "Intrinsic Descriptions," for more information on intrinsics and their parameters.

`TPStarted:`

TPID	Transaction program identifier
TPName	Name of transaction program
TraceSize	Trace file size in records

TraceFile	Name of trace file for user tracing

TPEnded:

TPID	Transaction program identifier

MCAllocate:

TPID	Transaction program identifier
SessionType	Configured session type used by the conversation
SyncLevel	Synchronization level for the conversation (CONFIRM or NONE)
ReturnControl	When control is returned to your program if no session is available (IMMEDIATE or WHEN_SESSION_ALLOCATED)
Timer	How long your program will wait on an intrinsic call for data or confirmation from the remote TP before control is returned to your program
NameLength	Length (in bytes) of remote TP name
RemoteTPName	Name of remote TP (in hex)
PIPxx	PIP data (xx = 1 through 16). On the left, the PIP data is displayed in hexadecimal. On the right, it is displayed in EBCDIC.

MCGetAllocate:

TPID	Transaction program identifier
SessionType	Configured session type used by the conversation
SyncLevel	Synchronization level for the conversation (CONFIRM or NONE)
Timer	How long your program will wait on an intrinsic call for data or confirmation from the remote TP before control is returned to your program
TPName	Name of local transaction program
PIPxx	PIP data (xx = 1 through 16). On the left, the PIP data is displayed in hexadecimal. On the right, it is displayed in EBCDIC.

MCConfirm:

RTS	Request to send received

MCDeallocate:

DeallocateType	Type of deallocation for the conversation (CONVERSATION_SYNC_LEVEL, FLUSH, ABEND, LOCAL, or CONFIRM)

MCPostOnRcpt:

Length	Length of data to be received before local TP is notified
Offset	Stack offset of data to be posted (This value is in decimal on MPE V and hexadecimal on MPE XL.)

MCPrepToRcv:

Locks	Lock specifying when control is returned to the conversation
Type	Type of PrepToRcv (FLUSH, CONFIRM or CONVERSATION_SYNC_LEVEL)

MCRcvAndWait, MCRcvNoWait:

RTS	Request to send received
Length	Length of data received
Offset	Stack offset of data received (This value is in decimal on MPE V and hexadecimal on MPE XL.)
WhatRcvd	What was received (data or control information)
Data	Data received. On the left, the data is displayed in hexadecimal. On the right, it is displayed in EBCDIC.

MCSendData:

RTS	Request to send received
Length	Length of data sent
Offset	Stack offset of data sent (This value is in decimal on MPE V and hexadecimal on MPE XL.)

	Data	Data sent. On the left, the data is displayed in hexadecimal. On the right, it is displayed in EBCDIC.
`MCSendError:`		
	RTS	Request to send received
`MCTest:`		
	PostedType	Type of information posted (DATA or NOT_DATA)
	Test	Condition being tested (POSTED or REQUEST_TO_SEND_RECEIVED)
`MCWait:`		
	PostedType	Type of information posted (DATA or NOT_DATA)
	ResourcePosted	Resource ID of the conversation being posted.

A Status Info

This appendix contains all of the status info values that are returned to LU 6.2 API intrinsics through the *Status* parameter.

The status info values are listed here in order by absolute value. No status info value has both a positive and a negative meaning. A positive value indicates that an intrinsic executed successfully and further information is available. A negative value indicates that an intrinsic did not execute successfully.

0 **MESSAGE:** Intrinsic called with parameter out of bounds.

CAUSE: Intrinsic executed successfully.

ACTION: None.

-1 **MESSAGE: Successful Completion.**

CAUSE: (**MPE V only**) The address of a parameter is outside of the DL to S stack boundary. (That is, it is either less than DL or greater than S).

ACTION: Verify that the stack data has not been corrupted by overwriting.

Verify that parameters are being passed to the intrinsic as specified in Chapter 7 , “Debugging.”

CAUSE: (**MPE V only**) Intrinsic called in split stack mode.

ACTION: Change your program so that it does not call an intrinsic in split stack mode.

CAUSE: (**MPE XL only**) The address of a parameter is outside the process data area.

ACTION: Verify that the data has not been corrupted.

CAUSE: (**MPE XL or V**) The value of the *Length* parameter is greater than the length of the *Data* array specified in the intrinsic call.

ACTION: Verify that the value of the *Length* parameter is less than or equal to the length of the *Data* array.

-2 **MESSAGE: Invalid 'ResourceID' parameter specified in intrinsic call.**

CAUSE: A *ResourceID* has been specified that does not correspond to any value returned by the MCAllocate intrinsic for an active conversation.

ACTION: Verify that the value specified in the intrinsic call is the same value returned by the MCAllocate or MCGetAllocate intrinsic for an active conversation.

-4 **MESSAGE: Out of range 'ReturnControl' parameter specified in intrinsic call.**

CAUSE: (MPE V only) The *ReturnControl* parameter has been set to a value other than 0.

ACTION: Because this parameter is not currently implemented, the value must always be 0. The parameter can be explicitly set to 0 in the intrinsic call, or it can be omitted from the intrinsic call and allowed to default to 0.

CAUSE: (MPE XL only) The *ReturnControl* parameter has been set to a value other than 0 or 1.

ACTION: The *ReturnControl* parameter of the MCAllocate intrinsic can be either 0 (IMMEDIATE) or 1 (WHEN_SESSION_ALLOCATED). Check to see that it is set to a valid value.

-5 **MESSAGE: Out of range 'SyncLevel' parameter specified in intrinsic call.**

CAUSE: The value of the *Synclevel* parameter is not valid.

ACTION: Verify that *Synclevel* is either 0 or 2.

-6 **MESSAGE: PIP data length is out of range.**

CAUSE: The length of an individual PIP is out of range.

ACTION: Verify that the length of each PIP is greater than or equal to 0 bytes and less than or equal to 1980 bytes.

-7 **MESSAGE: Out of range 'Timer' parameter specified in intrinsic call.**

CAUSE: The value for the *Timer* parameter is not within its valid range.

ACTION: Verify that the *Timer* parameter is a value from 0 through 28800 seconds.

-8 **MESSAGE: Out of range 'DeallocateType' parameter specified in intrinsic call.**

CAUSE: The *DeallocateType* parameter is not a valid value.

ACTION: Verify that the *DeallocateType* parameter is 0, 1, 2, 5 or 6.

-8 **MESSAGE: Out of range 'DeallocateType' parameter specified in intrinsic call.**

CAUSE: The *DeallocateType* parameter is not a valid value.

ACTION: Verify that the *DeallocateType* parameter is 0, 1, 2, 5 or 6.

-11 **MESSAGE: Out of range 'Length' parameter specified in intrinsic call.**

CAUSE: The value of the *Length* parameter is out of its valid range.

ACTION: Verify that the *Length* parameter is a value from 0 through 4092 bytes (or from 0 through 32763 bytes for the MCRcvAndWait intrinsic on MPE XL).

CAUSE: The value of the *Length* parameter is greater than the length of the *Data* array specified in the intrinsic call.

ACTION: Verify that the value of the *Length* parameter is less than or equal to the length of the *Data* array.

CAUSE: The *Length* parameter may be corrupted.

ACTION: Verify that addresses are not being corrupted.

-13 **MESSAGE: Data buffer specified in intrinsic call is out of bounds.**

CAUSE: The address of the *Data* array in the MCRcvAndWait, MCRcvNoWait, MCPostOnRcpt, or MCSendData intrinsic is invalid or corrupted.

ACTION: Verify that the *Data* array address is not being modified accidentally before it is passed to the intrinsic.

-15 **MESSAGE: Invalid 'TPID' parameter specified in intrinsic call.**

CAUSE: The *TPID* value specified in the intrinsic call does not correspond to the value returned by the TPStarted intrinsic.

ACTION: Verify that the *TPID* specified in the intrinsic call contains the value that was returned by TPStarted.

-16 **MESSAGE: Unable to open catalog file.**

CAUSE: The catalog files for LU 6.2 API do not exist.

ACTION: Verify that the files CATAPI.PUB.SYS and CATAPPC.APPC.SYS exist.

CAUSE: The catalog files for LU 6.2 API have not been released for system use.

ACTION: Verify that the files CATAPI.PUB.SYS and CATAPPC.APPC.SYS have been released for general system use.

-17 **MESSAGE: GENMESSAGE failed. (MPE V only)**

CAUSE: GENMESSAGE file is corrupted.

ACTION: Verify that GENMESSAGE file is not corrupted.

CAUSE: The MPE GENMESSAGE intrinsic did not execute properly.

ACTION: Notify your HP representative.

-17 **MESSAGE: CATREAD failed. (MPE XL only)**

CAUSE: CATREAD file is corrupted.

ACTION: Verify that CATREAD file is not corrupted.

CAUSE: The MPE CATREAD intrinsic did not execute properly.

ACTION: Notify your HP representative.

-19 **MESSAGE: APPC subsystem is inactive.**

CAUSE: The APPC subsystem has not been started or is no longer active.

ACTION: Request that the node manager start the APPC subsystem.

-20 **MESSAGE: Not enough stack space for intrinsic to run.**

CAUSE: The minimum amount of stack space required to call and execute an intrinsic is not available.

ACTION: (**MPE V only**) Increase the maximum stack size using the *Stack* parameter of the MPE RUN command. Increase the maximum stack size using the *Stack* or *MaxData* parameter of the MPE PREP command. For information on the RUN and PREP commands, refer to the *MPE V Command Reference Manual* or the online help facility.

ACTION: (**MPE XL only**) Increase the maximum stack size using the *NMStack* parameter of the MPE RUN command. Increase the maximum stack size using the *NMStack* parameter of the MPE LINK command. For information on the RUN and LINK commands, refer to the *MPE XL Command Reference Manual* or the online help facility.

-21 **MESSAGE:** Insufficient memory space to allocate a conversation.

CAUSE: Not enough space for the TP process to allocate an Available File Table (AFT) entry for the conversation.

ACTION: (**MPE V only**) Allocate more space for the process using the MPE PREP command (DL=*DLsize*) or the MPE RUN command (DL=*DLsize*). For information on the RUN and PREP commands, refer to the *MPE V Command Reference Manual* or the online help facility.

ACTION: (**MPE XL only**) Allocate more space for the process using the MPE RUN command (DL=*DLsize*). For information on the RUN command, refer to the *MPE XL Command Reference Manual* or the online help facility.

-22 **MESSAGE: APPC subsystem is inactive.**

CAUSE: Presentation Services was unable to build the InterProcess Communication (IPC) port.

ACTION: Notify your HP representative.

-23 **MESSAGE: Unable to allocate a conversation.**

CAUSE: The *SessionType* specified in the intrinsic call was not found in the APPC subsystem configuration.

ACTION: Verify that the correct *SessionType* is specified in the intrinsic call.

Verify that the specified *SessionType* has been configured in the NMCONFIG file for the APPC subsystem. For information on APPC subsystem configuration, see the *LU 6.2 API/V Node Manager's Guide* or the *APPC Subsystem on MPE XL Node Manager's Guide*.

CAUSE: No conversations are available because the maximum number of conversations has been allocated.

ACTION: Run programs at a time when resources are not in heavy demand.

-24 **MESSAGE: Unable to obtain an LU-LU session.**

CAUSE: All sessions for the specified session type are busy or unavailable.

ACTION: Have the node manager increase the number of active sessions using the APPCCONTROL SESSIONS command. See the *APPC Subsystem on MPE V Node Manager's Guide* or the *APPC Subsystem on MPE XL Node Manager's Guide* for information on APPCCONTROL commands.

Run programs at a time when resources are not in heavy demand.

Request that more sessions be configured for the specified session type. (The maximum number of sessions is 8 on MPE V, or 256 on MPE XL.)

-25 **MESSAGE: Could not find a conversation for this transaction program.**

CAUSE: LU 6.2 API was not able to find a request to start a conversation with the local TP.

ACTION: Check the stream file in the APPC.SYS account to make sure that the jobname in the MPE JOB command matches the file name of the stream file.

Make sure that MCGetAllocate is not being called by a locally initiated TP.

-26 **MESSAGE: Out of range 'PrepToRcvType' parameter specified in intrinsic call.**

CAUSE: The value of the *PrepToRcvType* parameter is not within its valid range.

ACTION: Verify that the *PrepToRcvType* value is 0, 1 or 2.

-27 **MESSAGE: Out of range 'Locks' parameter specified in intrinsic call.**

CAUSE: The value of the *Locks* parameter is not within its valid range.

ACTION: Verify that the *Locks* value is 0 or 1.

-28 **MESSAGE: Invalid Return Code 'oldstatus' passed to intrinsic.**

CAUSE: The *OldStatus* parameter specified in the MCErrMsg intrinsic call is undefined or was corrupted.

ACTION: Verify that the *OldStatus* value is a valid status info value and that it has not been corrupted.

+29 **MESSAGE: Could not find a conversation for this transaction program - timer popped. (MPE XL only)**

CAUSE: The configured Time-out value for the remotely initiated transaction program expired before an allocate request arrived from a remote TP.

ACTION: If this message indicates a problem, verify that the remote TP is sending the correct TP name in the allocate request. Verify that the remotely initiated TP is configured correctly through NMMGR. If necessary, raise the configured Time-out value.

-31 **MESSAGE: Confirm not allowed.**

CAUSE: A confirmation was requested and the synchronization level for the conversation was not set to CONFIRM.

ACTION: Set the *SyncLevel* parameter to CONFIRM in the MCAllocate or MCGetAllocate intrinsic if confirmation will be used during this conversation. This status info value can be returned when you call MCConfirm, MCPrepToRcv with *PrepToRcvType* = CONFIRM, or MCDeallocate with DeallocateType = CONFIRM.

-34 **MESSAGE: Out of range 'NumResources' parameter specified in intrinsic call.**

CAUSE: The value of the *NumResources* parameter is not within its valid range.

ACTION: Verify that the *NumResources* parameter is a value from 1 through 8 on MPE V, or from 1 through 256 on MPE XL.

-35 **MESSAGE: Out of range 'Test' parameter specified in intrinsic call.**

CAUSE: The value of the *Test* parameter is not within its valid range.

ACTION: Verify that the *Test* value is 0 or 1.

+36 **MESSAGE: Request To Send not received.**

CAUSE: This status info value is returned to the MCTest intrinsic (*Test* = REQUEST_TO_SEND_RECEIVED) when a RequestToSend has not been received.

ACTION: Informational message. No action necessary.

-37 **MESSAGE: Posting not active.**

CAUSE: This status info value is returned to the MCTest (*Test* = POSTED) or MCWait intrinsic when posting is not active (MCPostOnRcpt has not been called) for the specified mapped conversation.

ACTION: Before calling MCTest or MCWait, verify that the MCPostOnRcpt intrinsic has executed successfully for the specified conversation.

+38 **MESSAGE: Not Posted.**

CAUSE: No data or conversation status information has been received for the specified conversation.

ACTION: Informational message. No action necessary.

-40 **MESSAGE: Intrinsic called in invalid state.**

CAUSE: The program called an intrinsic from a state in which the intrinsic is not allowed.

ACTION: Refer to Appendix C , “State Transition Tables,” or Chapter 5 , “Intrinsic Descriptions,” to verify that the conversation is in the appropriate state for this intrinsic call.

-50 **MESSAGE: Allocation Error.**

CAUSE: Allocation could not be completed. Reasons may include the following:

CAUSE: The conversation is now in Deallocate state and can be deallocated.

CAUSE: A conversation type mismatch. The remote LU does not support mapped conversations, so it rejected an allocation request from the local TP.

ACTION: Verify that the remote LU supports mapped conversations.

CAUSE: An unrecognized *RemoteTPName*. The remote LU rejected the allocation request from the local TP because the TP name specified in the *RemoteTPName* parameter of the MCAllocate intrinsic was not recognized.

ACTION: Verify that the *RemoteTPName* matches the name of the correct program on the remote system.

Verify that the *RemoteTPName* has been converted from ASCII to EBCDIC correctly.

CAUSE: PIP data is not allowed. The remote TP does not support PIP data.

ACTION: Verify that the remote system accepts PIP data. If it does not, your program will have to be redesigned.

CAUSE: The remote system will not accept the PIP data as it has been specified.

ACTION: Verify that the PIP data is in the form that the remote system can accept.

CAUSE: An unavailable remote TP. The remote LU rejected the allocation request because the remote TP could not be started.

ACTION: Determine why the remote TP could not be started. Do not try to restart the TP until the problem has been resolved.

-51 **MESSAGE: Resource Failure: No retry possible.**

CAUSE: A conversation failure has occurred and the APPC subsystem is unable to reactivate the conversation.

Possible reasons include the following:

LU 6.2 API detected a violation of mapped conversation protocol.

A session failure has occurred.

The APPC subsystem has been stopped by the node manager.

The partner LU has deallocated the conversation, indicating a protocol error.

ACTION: Do not attempt the transaction again until the condition has been identified and corrected.

Verify that the APPC subsystem is active.

Verify that the partner LU is transmitting data that does not violate the LU 6.2 protocol.

-52 **MESSAGE: Resource Failure: Retry possible.**

CAUSE: The conversation has been prematurely terminated, probably due to a session failure because of a power outage or modem failure.

ACTION: The conversation is now in Deallocate state and must be deallocated. The program may be designed to reallocate the conversation and continue processing. The TP does not have to be restarted.

-56 **MESSAGE: Program Error: No data truncation has occurred.**

CAUSE: The remote TP has discovered an error. It flushed its send buffer, issued the equivalent of the `MCSendError` intrinsic, and is now in Send state. No truncation of received data has occurred at the mapped conversation protocol boundary.

ACTION: The required action depends on the error recovery procedure for the local program.

-60 **MESSAGE: Program Error: Data may have been purged.**

CAUSE: While in Receive or Confirm state, the remote TP discovered an error and issued the equivalent of the `MCSendError` intrinsic. Any data sent to the local TP may have been purged.

ACTION: The required action depends on the error recovery procedure for the local program.

-65 **MESSAGE: Received an invalid attach from the remote LU.**

CAUSE: The APPC subsystem rejected the remote TP's request to start a conversation, because it was invalid.

The APPC subsystem could not receive the PIP data successfully.

ACTION: Have remote programmer verify the parameters used to allocate the conversation.

PIP data length must be specified correctly by the remote TP.

+80 **MESSAGE: Timer has expired.**

CAUSE: No data or status information was received from the remote TP in the time limit specified in the *Timer* parameter of the MCAllocate intrinsic.

ACTION: This message is an event indicator. Any required action is program dependent.

-90 **MESSAGE: An internal error in Presentation Services has occurred.**

CAUSE:

ACTION: Note the circumstances and report them to your HP representative.

-91 **MESSAGE: An internal error in the APPC subsystem has occurred.**

CAUSE:

ACTION: Note the circumstances and report them to your HP representative.

-95 **MESSAGE: Internal Error: Unable to create Transaction Program port. (MPE XL only)**

CAUSE: Presentation Services was unable to build the InterProcess Communication (IPC) port for the transaction program.

ACTION: Notify your HP representative.

+100 **MESSAGE: Deallocate Normal received from the remote TP.**

CAUSE: The conversation has been deallocated by the remote TP, without any errors.

ACTION: Deallocate the local TP using the MCDeallocate intrinsic (*DeallocateType* = LOCAL).

-1002 **MESSAGE: An internal error at the mapped conversation level has occurred.**

CAUSE: The conversation has been deallocated by the remote TP, without any errors.

ACTION: Note the circumstances and report them to your HP representative.

-1003 **MESSAGE: Required parameter missing.**

CAUSE: A required parameter was not included in the call to an LU 6.2 API intrinsic.

ACTION: Verify that all the required parameters are included in the intrinsic call.

-1005 **MESSAGE: Insufficient Heap Space. (MPE V)**

CAUSE: The size of the DL to DB area was insufficient for the MCAllocate intrinsic to start a conversation.

ACTION: Increase the maximum stack size using the *Stack* parameter of the MPE RUN parameter of the MPE PREP command. For information on the RUN and PREP commands, refer to the *MPE V Command Reference Manual* or the online help facility.

-1006 **MESSAGE: Out of range 'RemoteTPLen' parameter specified in intrinsic call.**

CAUSE: The *RemoteTPLen* parameter is not within its valid range.

ACTION: Verify that the *RemoteTPLen* parameter is a value from 1 through 64, indicating the number of characters in the *RemoteTPName*.

-1007 **MESSAGE: Out of range 'NumPIPs' parameter specified in intrinsic call.**

CAUSE: The value of the *NumPIPs* parameter is not within its valid range.

ACTION: Verify that the *NumPIPs* parameter is a value from 0 through 16.

-1008 **MESSAGE: Invalid 'LocalTPName' parameter specified in intrinsic call. (MPE XL only)**

CAUSE: 1. The *LocalTPName* parameter of the MCGetAllocate intrinsic is coded as an output parameter when it should be passed as an input parameter

2. The *LocalTPName* parameter of the MCGetAllocate intrinsic does not match the *LocalTPName* parameter of the TPStarted intrinsic.

3. The *LocalTPName* is not configured as a remotely initiated TP.

ACTION: Verify that the *LocalTPName* parameter is being passed as an input parameter to the MCGetAllocate intrinsic. Verify that the *LocalTPName* parameter is the same in both the MCGetAllocate and TPStarted intrinsic calls. Verify that the *LocalTPName* is configured through NMMGR as a remotely initiated TP.

-1009 **MESSAGE: Combined length of PIPs is out of range.**

CAUSE: The combined length of *PIP1* through *PIP16* is greater than 1980 bytes.

ACTION: Reduce the lengths of the PIPs until their combined length is less than or equal to 1980 bytes.

-1010 **MESSAGE: Too many PIP subfields.**

CAUSE: The remote TP has sent more PIPs than the number specified in the *NumPIPs* parameter of the MCGetAllocate intrinsic.

ACTION: Consult with the remote programmer to make sure the number of PIPs sent matches the value in the *NumPIPs* parameter.

-1020 **MESSAGE: Deallocate Abend.**

CAUSE: The remote TP issued the equivalent of the MCDeallocate intrinsic, specifying the deallocate type ABEND.

ACTION: All buffers have been flushed. If the local TP was in Receive state, data may been lost. After receiving this status info value, the local TP is in Deallocate state and must issue the MCDeallocate intrinsic with *DeallocateType* = LOCAL.

-1030 **MESSAGE: TPStarted request rejected.**

CAUSE: The TPStarted intrinsic did not execute successfully because the maximum number of concurrently active TPs has been reached.

ACTION: Attempt program execution at a time when resources are not in heavy demand. The maximum number of TPs allowed to be executing at one time is 8 on MPE V, or 256 on MPE XL.

-1033 **MESSAGE: Unable to open file specified in the 'TraceFile' parameter.**

CAUSE: Invalid *TraceFile* parameter specified in intrinsic call. The disc file name specified in the *TraceFile* parameter is not a valid name, is not terminated with a blank, or specifies an unknown file name, lockword, group name, or account name.

ACTION: Verify that the disc file name specified in the *TraceFile* parameter is valid, is terminated with a blank, and specifies a valid file name, lockword, group name, and account name.

CAUSE: If two TP processes try to open the same trace file, the first process will be successful, and the second process will receive this status info value.

ACTION: Verify that different TPs specify different trace file names. If more than one instance of the same TP will be executing at the same time, allow the trace file name to default.

-1034 **MESSAGE: Out of range 'TraceSize' parameter specified in intrinsic call.**

CAUSE: The *TraceSize* parameter is not within its valid range.

ACTION: Verify that the *TraceSize* parameter is a value from 1 through 32767.

-1036 **MESSAGE: Out of range 'TraceOn' parameter specified in intrinsic call.**

CAUSE: The value of the *TraceOn* parameter is not within its valid range.

ACTION: Verify that the *TraceOn* parameter is a value from 0 through 3.

-1040 **MESSAGE: Conversation(s) not deallocated before calling TPENDED.**

CAUSE: One or more conversations were not deallocated before the TPEnded intrinsic was called.

ACTION: Verify that the MCDeallocate intrinsic was executed for any active conversations before calling TPEnded.

-1044 **MESSAGE: Multiple calls to TPSTARTED.**

CAUSE: The program attempted to call the TPStarted intrinsic more than once during program execution.

ACTION: Verify that TPStarted intrinsic is called only once.

-1050 **MESSAGE: Invalid 'Data' parameter specified in intrinsic call.**

CAUSE: The *Data* parameter specified in the MCRcvAndWait, MCRcvNoWait, or MCPostOnRcpt intrinsic does not match the *Data* parameter specified in a previous call to MCPostOnRcpt.

ACTION: Verify that the MCRcvAndWait or MCRcvNoWait intrinsic and the MCPostOnRcpt intrinsic are using the same variable in their *Data* parameters.

-1105 **MESSAGE: Internal Error: Conversation deallocated.**

CAUSE:

ACTION: Contact your HP representative.

B Sample Programs

This appendix contains an example LU 6.2 application. The application in this example enables a clerk in a retail store to check the credit of a buyer before allowing the buyer to charge a purchase to a credit card. The retail store has an HP 3000 running LU 6.2 API. The credit information is stored in a VSAM database on an IBM processor running CICS. The TP on the HP 3000 calls the `MCAllocate` intrinsic to allocate the conversation, and the TP on the IBM processor is started up in response to the allocate request.

The example TP on the HP 3000 is written in COBOL II and in Pascal. It performs the following tasks:

1. It calls the `TPStarted` intrinsic to initialize the TP, and then it calls the `MCAllocate` intrinsic to allocate a conversation with the CICS TP on the IBM processor.
2. It prompts a terminal user for a social security number and name, and then it reads the data from the terminal.
3. It translates the data to EBCDIC and sends it to the CICS TP on the IBM processor.
4. It receives a data record from the CICS TP and translates it to ASCII.
5. It interprets the information in the data record and displays “Credit Approved” or “Credit Denied” on the terminal screen.
6. It asks the user whether he or she wants to quit. If the user responds with “Y,” it deallocates the conversation; otherwise, it prompts the user for another social security number and name.

The example TP on the IBM processor is written in PL/1. It performs the following tasks:

1. It receives a social security number and name sent by the TP on the HP 3000.
2. Using the social security number as a key, it searches the VSAM database for credit information on the buyer.
3. It sends the information from the database to the TP on the HP 3000, or it reports an error. The remote TP can return any of 3 error codes:

   001 — The SS# is not in the database.

   002 — The SS# is in the database, but the name does not match the name sent by the HP 3000.

   003 — Miscellaneous system errors.
4. It waits for another data record or a deallocate request from the TP on the HP 3000.

Figure B-1 is a sample credit card verification report generated from the database on the IBM processor. This is the data set used by the example application in this appendix.

Figure B-1 is a sample credit card verification report generated from the database on the IBM processor. When the CICS TP receives a social security number and name from the TP on the HP 3000, it sends the data record associated with the social security number and name.

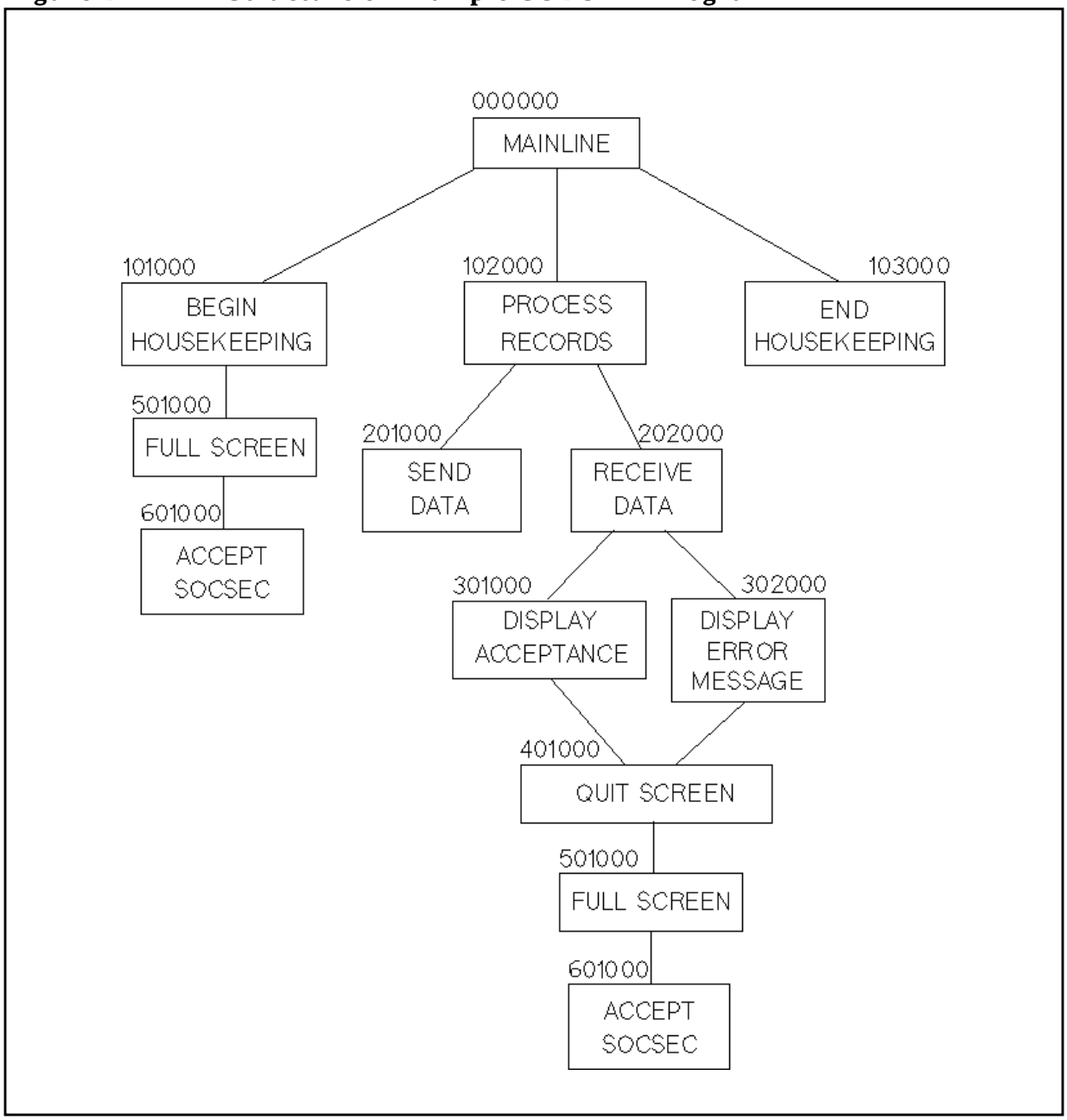
Figure B-1 Data Set for the Example Program

SOCIAL SECURITY NUMBER	NAME LAST	FIRST	MI	CREDIT CARD NAME	BALANCE	RISK CODE
567894321	REAGAN	RONALD	R	VICES INC	$9999.99	1
				MASTER BLASTER	$9999.99	
				PARTY HARDY	$9999.99	
				DESTITUTE PLUS	$9999.99	
568231942	REAGAN	NANCY	Q	VICES INC	$0100.00	5
568542874	APPLESEED	JOHNNY	P	RAPID SPEND	$0978.42	2
				PARTY HARDY	$0187.43	
				DESTITUTE PLUS	$0069.58	
				MASTER BLASTER	$0987.65	
569274321	HUBBARD	MOTHER	C	MASTER BLASTER	$1765.48	4
				VICES INC	$0895.38	
				RAPID SPEND	$0987.65	
				PARTY HARDY	$1069.53	
				DESTITUTE PLUS	$9872.13	
569358731	PAN	PETER	A	MASTER BLASTER	$0098.76	1
570743216	BAILEY	BEETLE	G	VICES INC	$0000.00	4
				MASTER BLASTER	$0000.00	
				PARTY HARDY	$0000.00	
				DESTITUTE PLUS	$9085.42	
573590451	CROCKER	BETTY	N	PARTY HARDY	$9816.54	3
577287453	RUBBLE	BARNEY	B	DESTITUTE PLUS	$0000.42	1
				VICES INC	$0010.10	
589022876	BEEBLEBROX	ZAPHOD	X	VICES INC	$1090.43	3
				RAPID SPEND	$0016.54	
				PARTY HARDY	$7612.01	
				DESTITUTE PLUS	$1087.94	

COBOL II Program

Figure B-2 is a chart of the program structure for the COBOL II TP that runs on the HP 3000.

Figure B-2 Structure of Example COBOL II Program



```
001000$CONTROL CROSSREF,SYMDEBUG
001100*-------------------------------------------------*
001200  IDENTIFICATION DIVISION.
001300*-------------------------------------------------*
001400 PROGRAM-ID.
001500 AUTHOR.
001600 INSTALLATION.
001700 DATE-WRITTEN.
001800 DATE-COMPILED.
001900*
002000 REMARKS.
002100*
002200*-------------------------------------------------*
002300  ENVIRONMENT DIVISION.
002400*-------------------------------------------------*
002500 CONFIGURATION SECTION.
002600 SOURCE-COMPUTER. HP 3000.
002700 OBJECT-COMPUTER. HP 3000.
002800 SPECIAL-NAMES.
002900     CONDITION-CODE IS CCODE.
003000*
003100*-------------------------------------------------*
003200  DATA DIVISION.
003300*-------------------------------------------------*
003400*
003500*-------------------------------------------------*
003600  WORKING-STORAGE SECTION.
003700*-------------------------------------------------*
003800*
003900 01  INTRINSIC-COMP.
004000     05  TPID                    PIC S9(4) COMP.
004100     05  TRACEON                 PIC S9(4) COMP  VALUE +1.
004200     05  LENGTH-REMOTE-TP-NAME   PIC S9(4) COMP  VALUE +4.
004300     05  RESOURCE-ID             PIC S9(4) COMP.
004400     05  TRANS-LENGTH            PIC S9(4) COMP  VALUE +30.
004500     05  RECEIVE-LENGTH          PIC S9(4) COMP.
004600     05  WHAT-RECEIVED           PIC S9(4) COMP.
004700     05  FULL-RECORD             PIC S9(4) COMP  VALUE +80.
004800     05  REQ-TO-SEND-REC         PIC S9(4) COMP.
004900     05  DATA-COMPLETE           PIC S9(4) COMP  VALUE +1.
005000     05  SEND-RECEIVED           PIC S9(4) COMP  VALUE +4.
005100     05  DEALLOCATE-TYPE         PIC S9(4) COMP  VALUE +0.
005200     05  TRANSLATE-TO-EBCDIC     PIC S9(4) COMP  VALUE +2.
005300     05  TRANSLATE-TO-ASCII      PIC S9(4) COMP  VALUE +1.
005400*
005500 01  INTRINSIC-STATUS            PIC S9(8) COMP.
005600 01  INTRINSIC-STATUS-ALL    REDEFINES INTRINSIC-STATUS.
005700     05  INTRINSIC-STATUS-INFO   PIC S9(4) COMP.
005800     05  INTRINSIC-STATUS-SUBSYS PIC S9(4) COMP.
005900*
006000 01  RETURN-CODE.
006100     05  ALLOCATE-RTRNCD         PIC X(5).
```

```
006200     05  DEALLOCATE-RTRNCD        PIC X(5).
006300     05  ENDED-RTRNCD             PIC X(5).
006400     05  SENDDATA-RTRNCD          PIC X(5).
006500     05  TPSTART-RTRNCD           PIC X(5).
006600     05  RCVANDWAIT-RTRNCD        PIC X(5).
006700*
006800 01  DISPLAY-WHAT-RECEIVED        PIC X(5).
006900*
007000 01  API-PARAMETERS.
007100     05  TPSTARTED-PARAMETERS.
007200         10  LOCAL-TP-NAME        PIC X(8)   VALUE "USERTP  ".
007300     05  ALLOCATE-PARAMETERS.
007400         10  SESSION-TYPE         PIC X(8)   VALUE "DISOSS1 ".
007500     05  REMOTE-TP-NAME.
007600         10  REMOTE-TP-NAME-EBCDIC  PIC X(4) VALUE SPACES.
007700         10  REMOTE-TP-NAME-ASCII   PIC X(4) VALUE "Z027".
007800*
007900 01  DEBUGGING-ERROR-MESSAGES.
008000     05  STARTED-ERR-MSG     PIC X(20)  VALUE 'TP STARTED ERROR'.
008100     05  ALLOCATE-ERR-MSG    PIC X(20)  VALUE 'ALLOCATE ERROR'.
008200     05  SENDDATA-ERR-MSG    PIC X(20)  VALUE 'SEND DATA ERROR'.
008300     05  DEALLOCATE-ERR-MSG  PIC X(20)  VALUE 'DEALLOCATE ERROR'.
008400     05  ENDED-ERR-MSG       PIC X(20)  VALUE 'ENDED ERROR'.
008500     05  CTRANSLATE-ERR-MSG  PIC X(20)  VALUE 'CTRANSLATE ERROR'.
008600     05  RCVANDWAIT-ERR-MSG  PIC X(20)  VALUE 'RCVANDWAIT ERROR'.
008700     05  WHAT-RECEIVED-MSG   PIC X(20)  VALUE 'WHAT RECEIVED ERROR'.
008800*
008900 01  CONTROL-FLAGS.
009000     05  QUIT-SW             PIC X.
009100*
009200 01  TRANSACTION-ERROR-CODES.
009300     05  SYSTEM-ERROR-CD     PIC 9(4)  VALUE 0003.
009400     05  SOCSEC-ERROR-CD     PIC 9(4)  VALUE 0001.
009500*
009600 01  CONTROL-VALUES.
009700     05  YES-SW              PIC X     VALUE 'Y'.
009800     05  NO-SW               PIC X     VALUE 'N'.
009900*
010000 01  CONSOLE-HEADING         PIC X(17) VALUE
010100     "CREDIT RISK CHECK".
010200*
010300 01  ACCEPT-CODE             PIC X     VALUE "3".
010400*
010500 01  MASTER-DATA.
010600     05  SOCSEC-MASTER.
010700         10  SOCSEC1-MASTER     PIC X(3).
010800         10  SOCSEC2-MASTER     PIC X(2).
010900         10  SOCSEC3-MASTER     PIC X(4).
011000     05  NAME-MASTER.
011100         10  LAST-NAME-MASTER   PIC X(10).
011200         10  FIRST-NAME-MASTER  PIC X(10).
011300         10  MI-NAME-MASTER     PIC X.
```

```
011400     05  CREDIT-INFO-MASTER   OCCURS 5 TIMES.
011500         10  CO-CODE-MASTER     PIC X.
011600         10  BALANCE-MASTER     PIC 9(4)V9(2).
011700     05  FILLER                 PIC X(14).
011800     05  RISK-CODE-MASTER       PIC X(1).
011900*
012000 01  ERROR-RECORD REDEFINES MASTER-DATA.
012100     05  ERROR-CODE             PIC 9(4).
012200     05  FILLER                 PIC X(76).
012300*
012400 01  TRANS-DATA.
012500     05  SOCSEC-TRANS.
012600         10  SOCSEC1-TRANS      PIC X(3).
012700         10  SOCSEC2-TRANS      PIC X(2).
012800         10  SOCSEC3-TRANS      PIC X(4).
012900     05  NAME-TRANS.
013000         10  LAST-NAME-TRANS    PIC X(10).
013100         10  FIRST-NAME-TRANS   PIC X(10).
013200         10  MI-NAME-TRANS      PIC X.
013300*
013400*
013500*-------------------------------------------------*
013600  PROCEDURE DIVISION.
013700*-------------------------------------------------*
013800*
013900*-------------------------------------------------*
014000  000000-MAINLINE                  SECTION.
014100*-------------------------------------------------*
014200*
014300     PERFORM 101000-BEGIN-HOUSEKEEPING.
014400*
014500     PERFORM 102000-PROCESS-RECORDS
014600       UNTIL QUIT-SW = YES-SW.
014700*
014800     PERFORM 103000-END-HOUSEKEEPING.
014900*
015000  000099-EXIT.
015100      STOP RUN.
015200*
```

```
015300*-------------------------------------------------*
015400  101000-BEGIN-HOUSEKEEPING        SECTION.
015500*-------------------------------------------------*
015600*  This section calls TPStarted to initialize resources
015700*  for the local TP, and then it calls MCAllocate to
015800*  allocate a conversation with the remote TP.
015900*
016000     MOVE NO-SW TO QUIT-SW.
016100*
016200     CALL INTRINSIC "TP'STARTED" USING LOCAL-TP-NAME,
016300                                       TPID,
016400                                       INTRINSIC-STATUS,
016500                                       TRACEON.
016600     IF INTRINSIC-STATUS IS NOT EQUAL TO ZERO
016700         MOVE YES-SW TO QUIT-SW
016800         MOVE INTRINSIC-STATUS-INFO TO TPSTART-RTRNCD
016900         DISPLAY STARTED-ERR-MSG,TPSTART-RTRNCD
017000         GO TO 101099-EXIT.
017100*
017200     CALL INTRINSIC "CTRANSLATE" USING TRANSLATE-TO-EBCDIC,
017300                                       REMOTE-TP-NAME-ASCII,
017400                                       REMOTE-TP-NAME-EBCDIC,
017500                                       LENGTH-REMOTE-TP-NAME.
017600     IF CCODE << ZERO
017700         DISPLAY CTRANSLATE-ERR-MSG,
017800                "CCL - REMOTE-TP-NAME NOT TRANSLATED"
017900         MOVE YES-SW TO QUIT-SW
018000         GO TO 101099-EXIT.
018100*
018200     CALL INTRINSIC "MCALLOCATE" USING TPID,
018300                                       SESSION-TYPE,
018400                                       REMOTE-TP-NAME-EBCDIC,
018500                                       LENGTH-REMOTE-TP-NAME,
018600                                       RESOURCE-ID,
018700                                       INTRINSIC-STATUS.
018800     IF INTRINSIC-STATUS IS NOT EQUAL TO ZERO
018900         MOVE YES-SW TO QUIT-SW
019000         MOVE INTRINSIC-STATUS-INFO TO ALLOCATE-RTRNCD
019100         DISPLAY ALLOCATE-ERR-MSG,ALLOCATE-RTRNCD
019200         GO TO 101099-EXIT.
019300*
019400     PERFORM 501000-FULL-SCREEN.
019500  101099-EXIT.
019600     EXIT.
019700*
019800*-------------------------------------------------*
019900  102000-PROCESS-RECORDS             SECTION.
020000*-------------------------------------------------*
020100*  This section calls SEND-DATA and RECEIVE-DATA.
020200*
020300     PERFORM 201000-SEND-DATA.
020400*
```

```
020500     IF QUIT-SW IS EQUAL TO YES-SW
020600        GO TO 102099-EXIT.
020700*
020800     PERFORM 202000-RECEIVE-DATA.
020900*
021000  102099-EXIT.
021100     EXIT.
021200*
021300*-------------------------------------------------*
021400  103000-END-HOUSEKEEPING          SECTION.
021500*-------------------------------------------------*
021600*  This section deallocates the conversation and calls
021700*  TPEnded to free the resources used by the local TP.
021800*
021900     CALL INTRINSIC "MCDEALLOCATE" USING RESOURCE-ID,
022000                                         DEALLOCATE-TYPE,
022100                                         INTRINSIC-STATUS.
022200     IF INTRINSIC-STATUS IS NOT EQUAL TO ZERO
022300         MOVE INTRINSIC-STATUS-INFO TO DEALLOCATE-RTRNCD
022400         DISPLAY DEALLOCATE-ERR-MSG,DEALLOCATE-RTRNCD.
022500*
022600     CALL INTRINSIC "TPENDED" USING TPID,
022700                                    INTRINSIC-STATUS.
022800     IF INTRINSIC-STATUS IS NOT EQUAL TO ZERO
022900         MOVE INTRINSIC-STATUS-INFO TO ENDED-RTRNCD
023000         DISPLAY ENDED-ERR-MSG,ENDED-RTRNCD.
023100*
023200  103099-EXIT.
023300     EXIT.
023400*
023500*-------------------------------------------------*
023600  201000-SEND-DATA                    SECTION.
023700*-------------------------------------------------*
023800*  This section translates the data received from the
023900*  user's screen into EBCDIC and sends it to the remote TP.
024000*
024100     CALL INTRINSIC "CTRANSLATE" USING TRANSLATE-TO-EBCDIC,
024200                                       TRANS-DATA,
024300                                       TRANS-DATA,
024400                                       TRANS-LENGTH.
024500     IF CCODE << ZERO
024600         DISPLAY CTRANSLATE-ERR-MSG,
024700                "CCL - TRANS-DATA NOT TRANSLATED"
024800         MOVE YES-SW TO QUIT-SW
024900         GO TO 201099-EXIT.
025000*
025100     CALL INTRINSIC "MCSENDDATA" USING RESOURCE-ID,
025200                                       TRANS-DATA,
025300                                       TRANS-LENGTH,
025400                                       REQ-TO-SEND-REC,
025500                                       INTRINSIC-STATUS.
025600     IF INTRINSIC-STATUS IS NOT EQUAL TO ZERO
```

```
025700         MOVE YES-SW TO QUIT-SW
025800         MOVE INTRINSIC-STATUS-INFO TO SENDDATA-RTRNCD
025900         DISPLAY SENDDATA-ERR-MSG,SENDDATA-RTRNCD.
026000*
026100  201099-EXIT.
026200     EXIT.
026300*
026400*-----------------------------------------------------*
026500  202000-RECEIVE-DATA                SECTION.
026600*-----------------------------------------------------*
026700*  This section calls MCRcvAndWait twice:  once to
026800*  receive a data record from the remote TP and once
026900*  to receive the instruction to change to Send state.
027000*  If this section receives a complete data record,
027100*  it calls CTranslate to translate it to ASCII.
027200*
027300     MOVE FULL-RECORD TO RECEIVE-LENGTH.
027400*
027500     CALL INTRINSIC "MCRCVANDWAIT" USING RESOURCE-ID,
027600                                         RECEIVE-LENGTH,
027700                                         REQ-TO-SEND-REC,
027800                                         MASTER-DATA,
027900                                         WHAT-RECEIVED,
028000                                         INTRINSIC-STATUS.
028100*
028200     IF INTRINSIC-STATUS IS NOT EQUAL TO ZERO
028300         MOVE INTRINSIC-STATUS-INFO TO RCVANDWAIT-RTRNCD
028400         DISPLAY RCVANDWAIT-ERR-MSG,RCVANDWAIT-RTRNCD
028500         MOVE YES-SW TO QUIT-SW
028600         GO TO 202099-EXIT.
028700*
028800     IF WHAT-RECEIVED IS NOT EQUAL TO DATA-COMPLETE
028900         MOVE WHAT-RECEIVED TO DISPLAY-WHAT-RECEIVED
029000         DISPLAY WHAT-RECEIVED-MSG,DISPLAY-WHAT-RECEIVED
029100         MOVE YES-SW TO QUIT-SW
029200         GO TO 202099-EXIT.
029300*
029400     CALL INTRINSIC "MCRCVANDWAIT" USING RESOURCE-ID,
029500                                         RECEIVE-LENGTH,
029600                                         REQ-TO-SEND-REC,
029700                                         MASTER-DATA,
029800                                         WHAT-RECEIVED,
029900                                         INTRINSIC-STATUS.
030000*
030100     IF INTRINSIC-STATUS IS NOT EQUAL TO ZERO
030200         MOVE INTRINSIC-STATUS-INFO TO RCVANDWAIT-RTRNCD
030300         DISPLAY RCVANDWAIT-ERR-MSG,RCVANDWAIT-RTRNCD
030400         MOVE YES-SW TO QUIT-SW
030500         GO TO 202099-EXIT.
030600*
030700     IF WHAT-RECEIVED IS NOT EQUAL TO SEND-RECEIVED
030800         MOVE WHAT-RECEIVED TO DISPLAY-WHAT-RECEIVED
```

```
030900         DISPLAY WHAT-RECEIVED-MSG,DISPLAY-WHAT-RECEIVED
031000         MOVE YES-SW TO QUIT-SW
031100         GO TO 202099-EXIT.
031200*
031300     CALL INTRINSIC "CTRANSLATE" USING TRANSLATE-TO-ASCII,
031400                                       MASTER-DATA,
031500                                       MASTER-DATA,
031600                                       RECEIVE-LENGTH.
031700     IF CCODE << ZERO
031800         DISPLAY CTRANSLATE-ERR-MSG,
031900                "CCL - MASTER-DATA NOT TRANSLATED"
032000         MOVE YES-SW TO QUIT-SW
032100         GO TO 202099-EXIT.
032200*
032300     IF RECEIVE-LENGTH IS EQUAL TO FULL-RECORD
032400         PERFORM 301000-DISPLAY-ACCEPTANCE
032500     ELSE
032600         PERFORM 302000-DISPLAY-ERROR-MESSAGE.
032700*
032800  202099-EXIT.
032900     EXIT.
033000*
033100*-------------------------------------------------*
033200  301000-DISPLAY-ACCEPTANCE           SECTION.
033300*-------------------------------------------------*
033400*  This section evaluates the Risk Code received from
033500*  the remote TP to determine whether to approve or deny
033600*  credit, and then it writes a message to the user's terminal.
033700*
033800     IF RISK-CODE-MASTER IS LESS THAN ACCEPT-CODE
033900         DISPLAY "CREDIT DENIED"
034000     ELSE
034100         DISPLAY "CREDIT APPROVED".
034200*
034300     PERFORM 401000-QUIT-SCREEN.
034400*
034500  301099-EXIT.
034600     EXIT.
034700*
```

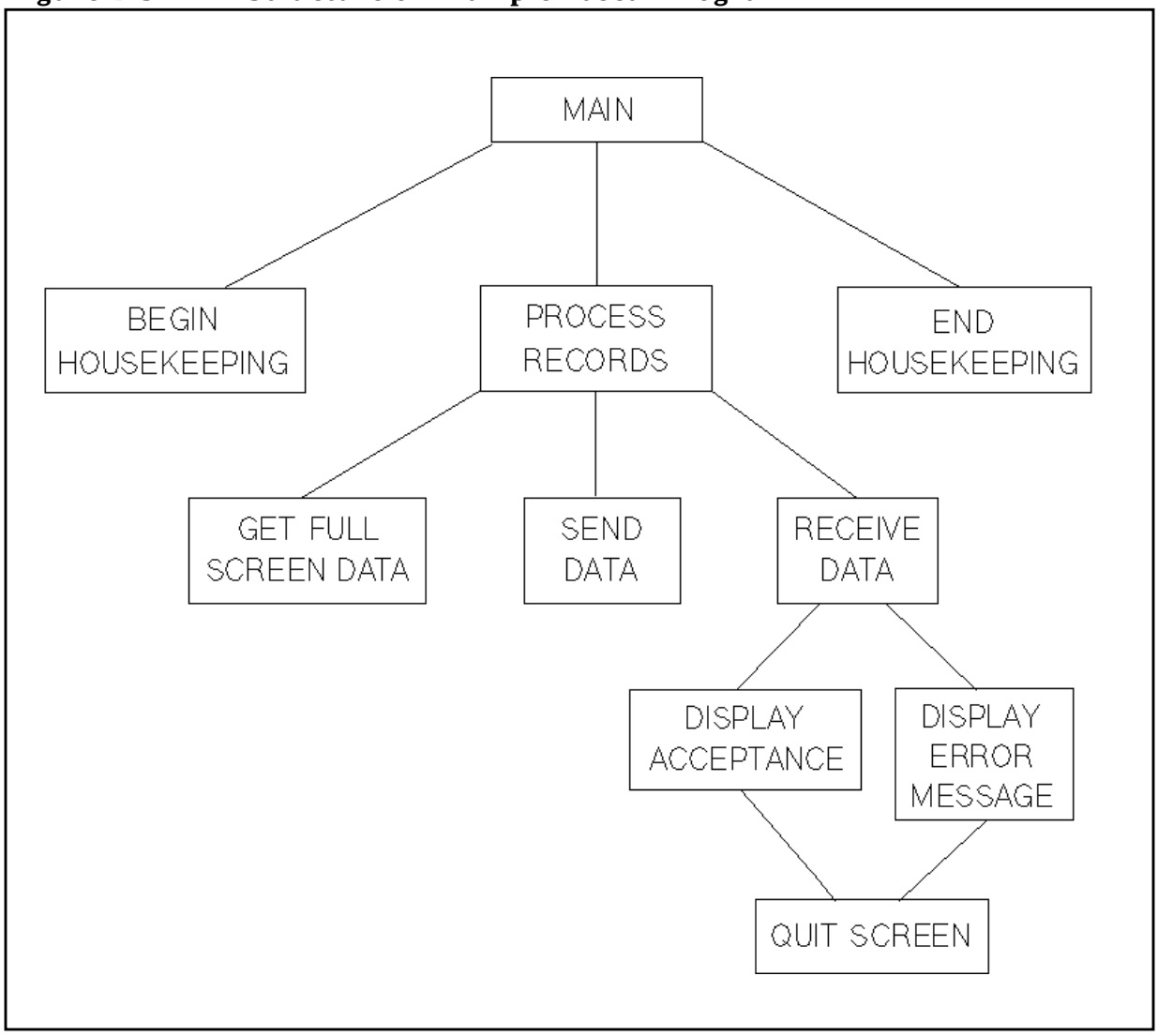
```
034800*-------------------------------------------------*
034900  302000-DISPLAY-ERROR-MESSAGE        SECTION.
035000*-------------------------------------------------*
035100*  This section evaluates the errorcode returned by the
035200*  remote TP and writes an error message to the user's
035300*  terminal.  The remote TP can return any of 3 error codes:
035400*     001 - The SS# is not in the database.
035500*     002 - The SS# is in the database, but the name does
035600*            not match the name sent by the HP 3000.
035700*     003 - Miscellaneous system errors.
035800*  Error codes 001 and 002 cause this section to call
035900*  QUIT-SCREEN.  Error code 003 causes this section to
036000*  set QUIT_SW to YES_SW.
036100*
036200     IF ERROR-CODE IS EQUAL TO SYSTEM-ERROR-CD
036300         DISPLAY SYSTEM-ERROR-CD
036400         MOVE YES-SW TO QUIT-SW
036500         GO TO 302099-EXIT.
036600*
036700     IF ERROR-CODE IS EQUAL TO SOCSEC-ERROR-CD
036800         DISPLAY "SS# not on file - CREDIT DENIED"
036900     ELSE
037000         DISPLAY "Invalid Name".
037100*
037200     PERFORM 401000-QUIT-SCREEN.
037300*
037400  302099-EXIT.
037500     EXIT.
037600*

037700*-------------------------------------------------*
037800  401000-QUIT-SCREEN               SECTION.
037900*-------------------------------------------------*
038000*  This section asks the user if he or she is ready
038100*  to quit.  If the user responds 'Y', this section
038200*  changes QUIT_SW to YES_SW.
038300*
038400     DISPLAY "READY TO QUIT (Y/N)?".
038500     ACCEPT QUIT-SW FREE.
038600*
038700     IF QUIT-SW IS NOT EQUAL TO YES-SW
038800         PERFORM 501000-FULL-SCREEN.
038900*
039000  401099-EXIT.
039100     EXIT.
039200*
```

```
039300*-------------------------------------------------*
039400  501000-FULL-SCREEN                SECTION.
039500*-------------------------------------------------*
039600*  This section prompts the user for data and
039700*  receives the data from the terminal.
039800*
039900     MOVE SPACE TO TRANS-DATA.
040000     MOVE SPACES TO MASTER-DATA.
040100*
040200     DISPLAY CONSOLE-HEADING.
040300*
040400     DISPLAY "SOCSEC #  :".
040500     PERFORM 601000-ACCEPT-SOCSEC
040600         UNTIL SOCSEC-TRANS IS NUMERIC.
040700*
040800     DISPLAY "LASTNAME  :".
040900     ACCEPT LAST-NAME-TRANS FREE.
041000*
041100     DISPLAY "FIRSTNAME :".
041200     ACCEPT FIRST-NAME-TRANS FREE.
041300*
041400     DISPLAY "MI        :".
041500     ACCEPT MI-NAME-TRANS FREE.
041600*
041700  501099-EXIT.
041800     EXIT.
041900*
042000*-------------------------------------------------*
042100  601000-ACCEPT-SOCSEC              SECTION.
042200*-------------------------------------------------*
042300*  This section prompts the user for a social security
042400*  number and accepts it from the terminal.
042500*
042600     ACCEPT SOCSEC-TRANS FREE.
042700*
042800     IF SOCSEC-TRANS IS EQUAL TO SPACES
042900         DISPLAY "SOCSEC # MUST BE NUMERIC"
043000         DISPLAY "SOCSEC #  :"
043100         GO TO 601099-EXIT.
043200*
043300     IF SOCSEC-TRANS IS NOT NUMERIC
043400         DISPLAY "SOCSEC # MUST BE NUMERIC"
043500         DISPLAY "SOCSEC #  :".
043600*
043700  601099-EXIT.
043800     EXIT.
```

Pascal Program

Figure B-3 is a chart of the program structure for the Pascal TP that runs on the HP 3000.

Figure B-3 Structure of Example Pascal Program



```
$uslinit$
$standard_level 'HP3000'; tables on; code_offsets on; xref on$
$global 'SPL'$
$PAGE$

program credit(input,output);

{ Date written:  August, 1987.}
{ Date compiled: August, 1987.}

const
  ACCEPT_CODE          = '3';
  DATA_COMPLETE        = 1;
  FULL_RECORD          = 80;
  LENGTH_REMOTE_TPNAME = 4;
  NO_SW                = false;
  YES_SW               = true;
  ON                   = 2;
  CONVSYNCLEVEL        = 0;
  SEND                 = 4;
  SOC_SEC_ERROR_CD     = 1;
  SYSTEM_ERROR_CD      = 3;
  TRANSLATE_TO_ASCII   = 1;
  TRANSLATE_TO_EBCDIC  = 2;
  TRANSLENGTH          = 30;
  YES                  = ['y', 'Y'];

  AllocateErrMsg     = text ['Allocate Error      '];
  CTranslateErrMsg   = text ['CTranslate Error    '];
  DeallocateErrMsg   = text ['Deallocate Error    '];
  EndedErrMsg        = text ['TP Ended Error      '];
  RcvAndWaitErrMsg   = text ['RcvAndWait Error    '];
  SendDataErrMsg     = text ['Send Data Error     '];
  StartedErrMsg      = text ['TP Started Error    '];
  WhatReceivedErrMsg = text ['What Received Error '];

type
  shortint    = -32768..32767;
  pac4type    = packed array [1..4] of char;
  nametype    = packed array [1..10] of char;
  errmsgtype  = packed array [1..20] of char;
  ssnumtype   = packed array [1..9] of char;
  balancetype = packed array [1..6] of char;

  MasterDataType = record
    case shortint of
      0:  (SocSecMaster    : ssnumtype;
           LastNameMaster  : nametype;
           FirstNameMaster : nametype;
           MINameMaster    : char;
           CoCodeMaster1   : char;
           BalanceMaster1  : balancetype;
           CoCodeMaster2   : char;
```

```
            BalanceMaster2  : balancetype;
            CoCodeMaster3   : char;
            BalanceMaster3  : balancetype;
            CoCodeMaster4   : char;
            BalanceMaster4  : balancetype;
            CoCodeMaster5   : char;
            BalanceMaster5  : balancetype;
            Filler          : packed array [1..14] of char;
            RiskCodeMaster  : char);
      1:  (ErrorCode        : pac4type;
            ErrorFiller     : packed array [1..76] of char);
    end;

  short_text    = packed array [1..8] of char;
  text          = packed array [1..20] of char;
  TPNameType    = packed array [1..LENGTH_REMOTE_TPNAME] of char;

  TransDataType = record
                    SocSecTrans   : ssnumtype;
                    LastNameTrans : nametype;
                    FirstNameTrans: nametype;
                    MINameTrans   : char;
                  end;
hpe_status       = record
    case integer of
      0 : (all    : integer);
      1 : (info   : shortint;
           subsys : shortint);
    end;

var
  LocalTPName,
  SessionType       : short_text;
  RemoteTPNameASCII : TPNameType;
  ResourceID,
  TPID,
  TraceOn,
  ReceiveLength,
  WhatReceived,
  DeallocateType    : shortint;
  TransData         : TransDataType;
  Ready             : char;
  Quit_SW           : boolean;

procedure TPStarted;    intrinsic;
procedure TPEnded;      intrinsic;
procedure MCAllocate;   intrinsic;
procedure MCDeallocate; intrinsic;
procedure MCSendData;   intrinsic;
procedure MCRcvAndWait; intrinsic;
procedure CTranslate;   intrinsic;
function bin $alias 'binary'$ : shortint;  intrinsic;
$PAGE$
```

```
{************************************************************
  ErrorHandler
      This procedure returns the error message associated
      with a status info value.
************************************************************}

procedure ErrorHandler (IntrinsicMsg : text;
                        Status : shortint;
                        var Quit_SW : boolean);

begin
   Quit_SW := YES_SW;
   writeln (IntrinsicMsg, Status:3);
end;

$PAGE$
{************************************************************
  GetFullScreenData
      This procedure prompts the user for data and receives
      the data from the terminal.
************************************************************}

procedure GetFullScreenData (var TransData : TransDataType);

begin
   with TransData do
   begin
      SocSecTrans    := '         ';
      LastNameTrans  := '          ';
      FirstNameTrans := '          ';
      MINameTrans    := ' ';

      writeln ('Credit Risk Check.');
      writeln;

      writeln ('Social Security Number:');
      readln (SocSecTrans);

      writeln ('Last Name:');
      readln (LastNameTrans);

      writeln ('First Name:');
      readln (FirstNameTrans);

      writeln ('Middle Initial:');
      readln (MINameTrans);

   end;
end;

$PAGE$
```

```
{*********************************************************
  BeginHouseKeeping
      This procedure calls TPStarted to initialize resources
      for the local TP, and then it calls MCAllocate to
      allocate a conversation with the remote TP.
*********************************************************}

procedure BeginHouseKeeping (LocalTPName : short_text;
                             RemoteTPNameASCII : TPNameType;
                             SessionType : short_text;
                             var TPID, ResourceID : shortint;
                             TraceOn : shortint;
                             var Quit_SW : boolean);

var
  IntrinsicStatus : hpe_status;
  RemoteTPNameEBCDIC : TPNameType;

begin
   Quit_SW := NO_SW;

   TPStarted (LocalTPName, TPID, IntrinsicStatus, TraceOn);

   if IntrinsicStatus.all <<>> 0 then
      ErrorHandler (StartedErrMsg, IntrinsicStatus.info, Quit_SW)

   else
   begin
      CTranslate (TRANSLATE_TO_EBCDIC, RemoteTPNameASCII,
                  RemoteTPNameEBCDIC, LENGTH_REMOTE_TP_NAME);

      if CCode = 1 then
      begin
         Quit_SW := YES_SW;
         writeln (CTranslateErrMsg, 'CCL - Remote TP Name not translated.');
      end

      else
      begin
         MCAllocate (TPID, SessionType, RemoteTPNameEBCDIC,
                     LENGTH_REMOTE_TP_NAME, ResourceID, IntrinsicStatus);

         if IntrinsicStatus.all <<>> 0 then
            ErrorHandler (AllocateErrMsg, IntrinsicStatus.info, Quit_SW);

      end;
   end;
end;

$PAGE$
```

```
{************************************************************
  SendData
      This procedure translates the data received from the
      user's screen into EBCDIC and sends it to the remote TP.
************************************************************}

procedure SendData (ResourceID : shortint;
                    TransData : TransDataType;
                    var Quit_SW : boolean);

var
  IntrinsicStatus : hpe_status;
  ReqToSendRec : shortint;

begin
   CTranslate (TRANSLATE_TO_EBCDIC, TransData, TransData, TRANSLENGTH);

   if CCode = 1 then
   begin
      Quit_SW := YES_SW;
      writeln (CTranslateErrMsg, 'CCL - TransData not translated.');
   end

   else
   begin
      MCSendData (ResourceID, TransData, TRANSLENGTH,
                  ReqToSendRec, IntrinsicStatus);

      if IntrinsicStatus.all <<>> 0 then
         ErrorHandler (SendDataErrMsg, IntrinsicStatus.info, Quit_SW);
   end;
end;

$PAGE$

{************************************************************
  QuitScreen
      This procedure asks the user if he or she is ready to
      quit.  If the user responds 'Y', this procedure changes
      Quit_SW to YES_SW.
************************************************************}

procedure QuitScreen (var Quit_SW : boolean);

begin
   writeln ('Ready to quit (Y/N)?');
   readln (Ready);

   if Ready in YES then
      Quit_SW := YES_SW;
end;

$PAGE$
```

```
{************************************************************
  DisplayAcceptance
      This procedure evaluates the Risk Code received from the
      remote TP to determine whether to approve or deny credit,
      and then it writes a message to the user's terminal.
************************************************************}

procedure DisplayAcceptance (RiskCode : shortint;
                             var Quit_SW : boolean);

begin
   if ord(RiskCode) << ord(ACCEPT_CODE) then
      writeln ('Credit Denied.')

   else
      writeln ('Credit Approved.');

   QuitScreen (Quit_SW);
end;

$PAGE$
{************************************************************
  DisplayErrorMessage
      This procedure evaluates the errorcode returned by the
      remote TP and writes an error message to the user's
      terminal.  The remote TP can return any of 3 error codes:
          001 - The SS# is not in the database.
          002 - The SS# is in the database, but the name does
                not match the name sent by the HP 3000.
          003 - Miscellaneous system errors.
      Error codes 001 and 002 cause this procedure to call
      QuitScreen.  Error code 003 causes this procedure to
      set Quit_SW to YES_SW.
************************************************************}

procedure DisplayErrorMessage (ErrorCode : shortint;
                               var Quit_SW : boolean);

begin
   if ErrorCode = SYSTEM_ERROR_CD then
   begin
      writeln (errorcode:4);
      Quit_SW := YES_SW;
   end

   else
   begin
      if ErrorCode = SOCSEC_ERROR_CD then
         writeln ('SS# not on file - Credit Denied.')
      else
         writeln ('Invalid Name');
      QuitScreen (Quit_SW);
```

```
   end;
end;

$PAGE$

 {*********************************************************
  ReceiveData
      This procedure calls MCRcvAndWait twice:  once to
      receive a data record from the remote TP and once to
      receive the instruction to change to Send state.  If
      this procedure receives a complete data record, it
      calls CTranslate to translate it to ASCII.
*********************************************************}

procedure ReceiveData (ResourceID : shortint;
                       var Quit_SW : boolean);

var
  IntrinsicStatus : hpe_status;
  MasterData      : MasterDataType;
  ReqToSendRec    : shortint;

begin
   ReceiveLength := FULL_RECORD;

   MCRcvAndWait (ResourceID, ReceiveLength, ReqToSendRec, MasterData,
                 WhatReceived, IntrinsicStatus);

   if IntrinsicStatus.all <<>> 0 then
      ErrorHandler (RcvAndWaitErrMsg, IntrinsicStatus.info, Quit_SW)

   else
   begin
      if WhatReceived <<>> DATA_COMPLETE then
         ErrorHandler (WhatReceivedErrMsg, WhatReceived, Quit_SW)

      else
      begin
         MCRcvAndWait (ResourceID, ReceiveLength, ReqToSendRec,
                       MasterData, WhatReceived, IntrinsicStatus);

         if IntrinsicStatus.all <<>> 0 then
            ErrorHandler (RcvAndWaitErrMsg, IntrinsicStatus.info, Quit_SW)

         else
         begin
            if WhatReceived <<>> SEND then
               ErrorHandler (WhatReceivedErrMsg, WhatReceived,
                             Quit_SW)

            else
            begin
               CTranslate (TRANSLATE_TO_ASCII, MasterData, MasterData,
```

```
                        ReceiveLength);

               if CCode = 1 then
               begin
                  Quit_SW := YES_SW;
                  writeln (CTranslateErrMsg,
                            'CCL - MasterData not translated.');
               end;

               if not Quit_SW then
               begin
                  if ReceiveLength = FULL_RECORD then
                     DisplayAcceptance (MasterData.RiskCodeMaster,
                                        Quit_SW)

                  else
                     DisplayErrorMessage (bin(MasterData.ErrorCode, 4),
                                        Quit_SW);
                end
             end
          end
       end
    end
end;

$PAGE$

{************************************************************
  ProcessRecords
      This procedure calls GetFullScreenData, SendData, and
      ReceiveData.
************************************************************}

procedure ProcessRecords (ResourceID : shortint;
                          var Quit_SW : boolean);

begin
   GetFullScreenData (TransData);
   SendData (ResourceID, TransData, Quit_SW);

   if not Quit_SW then
      ReceiveData (ResourceID, Quit_SW);
end;

$PAGE$
```

```
{************************************************************
  EndHousekeeping
      This procedure deallocates the conversation and calls
      TPEnded to free the resources used by the local TP.
************************************************************}

procedure EndHousekeeping (ResourceID, TPID : shortint);

var
  IntrinsicStatus : hpe_status;

begin
   MCDeallocate (ResourceID, DeallocateType, IntrinsicStatus);

   if IntrinsicStatus.all <<>> 0 then
      ErrorHandler (DeallocateErrMsg, IntrinsicStatus.info, Quit_SW)

   else
   begin
      TPEnded (TPID, IntrinsicStatus);

      if IntrinsicStatus.all <<>> 0 then
         ErrorHandler (EndedErrMsg, IntrinsicStatus.info, Quit_SW)
   end;
end;

$PAGE$
{************************************************************
  Main Program
************************************************************}

begin
   LocalTPName       := 'USERTP  ';
   RemoteTPNameASCII := 'Z027';
   Traceon           := ON;
   SessionType       := 'APISESS ';
   DeallocateType    := CONVSYNCLEVEL;

   BeginHousekeeping (LocalTPName, RemoteTPNameASCII, SessionType,
                      TPID, ResourceID, Traceon, Quit_SW);

   While not Quit_SW do
      ProcessRecords (ResourceID, Quit_SW);

   EndHousekeeping (ResourceID, TPID, DeallocateType);

end.
```

CICS Program (PL/I)

CICS provides a high-level command interface to the LU 6.2 verbs. Table B-1 gives the mappings between the LU 6.2 verbs and the equivalent CICS commands issued by the CICS TP.

Table B-1 Mapping of CICS Commands to LU 6.2 Verbs

LU 6.2 Verb	CICS Command
MCGetAllocate	EXEC CICS EXTRACT PROCESS
MCRcvAndWait	EXEC CICS RECEIVE
MCSendData	EXEC CICS SEND
MCConfirmed	EXEC CICS ISSUE CONFIRMATION

The EXEC Interface Block (EIB) returns parameter values from the LU 6.2 verbs to the PL/I TP. Table B-2 lists the EIB values used in the PL/I TP and their meanings in the conversation.

Table B-2 Meanings of EXEC Interface Block Values

EIB Value	Meaning
EIBFREE	The remote TP called MCDeallocate to end the conversation normally.
EIBCONF	The remote TP called MCConfirm< to request confirmation.
EIBEOC	End-of-chain indicator.
EIBCOMPL	The CICS TP has received a complete data record.
EIBRECV	The CICS TP is in Receive state.

```
S027:

   PROCEDURE OPTIONS (MAIN);

DCL
  1 RECEIVE_AREA,
    2 KEY   CHAR(9) INIT((9)' '),
    2 NAME  CHAR(21) INIT((21)' ');

DCL
  1 SEND_AREA,
    2 KEY   CHAR(9) INIT((9)' '),
    2 NAME  CHAR(21) INIT((21)' '),
    2 DATA  CHAR(50) INIT((50)' ');

/***  error codes ***/
DCL NOT_FOUND    CHAR(4) INIT('0001');
DCL INVALID_NAME CHAR(4) INIT('0002');
DCL MISC_ERROR   CHAR(4) INIT('0003');

DCL ADDR        BUILTIN;
DCL CSTG        BUILTIN;
DCL HIGH        BUILTIN;
DCL LOW         BUILTIN;
DCL SUBSTR      BUILTIN;
DCL STG         BUILTIN;
DCL VERIFY      BUILTIN;

DCL CONV_GONE     BIT(1)  INIT('0'B);
DCL CONFIRM_REQ   BIT(1)  INIT('0'B);
DCL DATA_COMPLETE BIT(1)  INIT('0'B);
DCL INLEN   FIXED BIN(31) INIT(30);
DCL RSC     CHAR(6)       INIT('TPFILE');
DCL SYNC    FIXED BIN(15) INIT(0);

/***   Begin MAIN   ***/

   EXEC CICS HANDLE CONDITION NOTFND(L_NFD)
                              ERROR(L_ERR);

/* Receive attach from HP 3000.  Equivalent to MCGetAllocate. */

   EXEC CICS EXTRACT PROCESS SYNCLEVEL(SYNC);

RCV_LOOP:
   DO WHILE ((CONV_GONE = '0'B) &
             (CONFIRM_REQ = '0'B));

/* Until the partner TP deallocates the conversation or */
/* the partner TP issues MCConfirm to request confirmation, */
/* receive another 30-byte record with 9-digit key and 21-character name. */

   EXEC CICS RECEIVE INTO(RECEIVE_AREA) LENGTH(INLEN);
```

```
/* If the partner TP deallocated, exit the receive loop. */

   IF DFHEIBLK.EIBFREE = HIGH(1) THEN
     DO;
     CONV_GONE = '1'B;
     LEAVE RCV_LOOP;
     END;

/* If the partner TP called MCConfirm, exit the receive loop. */

   IF DFHEIBLK.EIBCONF = HIGH(1) THEN
     DO;
     CONFIRM_REQ = '1'B;
     LEAVE RCV_LOOP;
     END;

/* If End-Of-Chain & DATA_COMPLETE & partner TP called MCPrepToRcv */

   IF ( (DFHEIBLK.EIBEOC   = HIGH(1)) &
        (DFHEIBLK.EIBCOMPL = HIGH(1)) &
        (DFHEIBLK.EIBRECV  = LOW(1)) )  THEN
     DO;

       IF VERIFY(RECEIVE_AREA.KEY,'0123456789') = 0 THEN
       DO;

/* Query the database for the key received from the remote TP. */

         EXEC CICS ENQ RESOURCE(RSC) LENGTH(6);
         EXEC CICS READ DATASET('TPFILE') INTO(SEND_AREA)
                        RIDFLD(RECEIVE_AREA.KEY);
         EXEC CICS DEQ RESOURCE(RSC) LENGTH(6);

   IF RECEIVE_AREA.NAME ¬ = SEND_AREA.NAME THEN

/* The above line contains a logicalnot or (NOT EQUAL) sign. */
/* If the name in the database doesn't match the name from the remote TP, */
/* issue an error code and call McPrepToRcv (INVITE WAIT). */

         DO;
           EXEC CICS SEND FROM(INVALID_NAME) INVITE WAIT;
         END;

/* If names match, send the data record and call MCPrepToRcv (INVITE WAIT).
*/

         ELSE DO;
           EXEC CICS SEND FROM(SEND_AREA) INVITE WAIT;
         END;
       END;

/* Otherwise, report a misc. error and call MCPrepToRcv (INVITE WAIT). */
```

```
       ELSE DO;
         EXEC CICS DEQ RESOURCE(RSC) LENGTH(6);
L_ERR:   EXEC CICS SEND FROM(MISC_ERROR) INVITE WAIT;
       END;
     END;
END RCV_LOOP;

/* If the partner TP called MCConfirm, */
/* respond with MCConfirmed (EXEC CICS ISSUE CONFIRMATION). */

L_BYE:
   IF ((SYNC = 1) & (CONFIRM_REQ = '1'B)) THEN
     DO;
       EXEC CICS ISSUE CONFIRMATION;
       CONFIRM_REQ = '0'B;
     END;

/* If the remote TP deallocated, deallocate the local conversation. */

   IF DFHEIBLK.EIBFREE = HIGH(1) THEN
     DO;
       EXEC CICS RETURN;
     END;

/* If the key sent by the remote TP is not in the database, */
/* report an error, call MCPrepToRcv (INVITE WAIT), and */
/* return to the receive loop to receive another record. */

L_NFD:
     DO;
       EXEC CICS DEQ RESOURCE(RSC) LENGTH(6);
       EXEC CICS SEND FROM(NOT_FOUND)  INVITE WAIT;
       GOTO RCV_LOOP;
     END;

END S027;
```

C State Transition Tables

This appendix contains the state transition tables for all the conversation states. Each table contains the following information:

- The intrinsics that can be called from the state.
- The state of the local side of the conversation after the intrinsic has executed and a status info value has been returned.
- The state of the remote side of the conversation after the intrinsic has executed.

*** means that the state cannot be determined from the local side of the conversation.

Table C-1 shows the Confirm State transition table.

Table C-1 Confirm State

Intrinsics You Can Call	Status Info	Local State After Intrinsic Execution	Remote State After Intrinsic Execution
MCConfirmed	0 Successful Completion	Receive	Send
MCDeallocate (ABEND)	0 Successful Completion	Reset	Deallocate
MCErrMsg	Any value	Confirm	Send
MCGetAttr	Any value	Confirm	Send
MCReqToSend	Any value	Confirm	Send
MCSendError	0 Successful Completion	Send	Receive
	-51 Resource Failure No Retry	Deallocate	***
	-52 Resource Failure Retry	Deallocate	***

Table C-2 shows the Confirm Deallocate State transition table.

Table C-2 Confirm Deallocate State

Intrinsics You Can Call	Status Info	Local State After Intrinsic Execution	Remote State After Intrinsic Execution
MCConfirmed	0 Successful Completion	Deallocate	Reset
MCDeallocate (ABEND)	0 Successful Completion	Reset	Deallocate
MCErrMsg	Any value	Confirm Deallocate	Send
MCGetAttr	Any value	Confirm Deallocate	Send
MCSendError	0 Successful Completion	Send	Receive
	-51 Resource Failure No Retry	Deallocate	***
	-52 Resource Failure Retry	Deallocate	***

Table C-3 shows the Confirm Send State transition table.

Table C-3 Confirm Send State

Intrinsics You Can Call	Status Info	Local State After Intrinsic Execution	Remote State After Intrinsic Execution
MCConfirmed	0 Successful Completion	Send	Receive
MCDeallocate (ABEND)	0 Successful Completion	Reset	Deallocate
MCErrMsg	Any value	Confirm Send	Receive
MCGetAttr	Any value	Confirm Send	Receive
MCSendError	0 Successful Completion	Send	Receive
	-51 Resource Failure No Retry	Deallocate	***
	-52 Resource Failure Retry	Deallocate	***

Table C-4 shows the Deallocate State transition table.

Table C-4 Deallocate State

Intrinsics You Can Call	Status Info	Local State After Intrinsic Execution	Remote State After Intrinsic Execution
MCDeallocate (LOCAL)	0 Successful Completion	Reset	Reset
MCErrMsg	Any value	Deallocate	Reset
MCGetAttr	Any value	Deallocate	Reset

Table C-5 show the Receive State transition table.

Table C-5 Receive State

Intrinsics You Can Call	Status Info	Local State After Intrinsic Execution	Remote State After Intrinsic Execution
MCDeallocate (ABEND)	0 Successful Completion	Reset	Deallocate
MCErrMsg	Any value	Receive	Send
MCGetAttr	Any value	Receive	Send
MCPostOnRcpt	Any value	Receive	Send

Table C-5 Receive State

Intrinsics You Can Call	Status Info	Local State After Intrinsic Execution	Remote State After Intrinsic Execution
MCRcvAndWait or MCRcvNoWait	0 Successful Completion		
	WhatReceived= DATA_COMPLETE	Receive	Send
	WhatReceived= DATA_INCOMPLETE	Receive	Send
	WhatReceived= SEND	Send	Receive
	WhatReceived= CONFIRM	Confirm	Send
	WhatReceived= CONFIRM_SEND	Confirm Send	Receive
	WhatReceived= CONFIRM_DEALLOCATE	Confirm Deallocate	Deallocate
	-50 Allocation Error	Deallocate	***
	-51 Resource Failure No Entry	Deallocate	***
	-52 Resource Failure Retry	Deallocate	***
	-56 Prog Error No Truncation	Receive	Send
	-60 Prog Error Data Purged	Receive	Send
	+80 Timer has expired	Receive	***
	+100 Deallocate Normal	Deallocate	Reset
	-1020 Deallocate Abend	Deallocate	***
MCReqToSend	0 Successful Completion	Receive	Send
MCSendError	0 Successful Completion	Send	Receive
	-51 Resource Failure No Entry	Deallocate	***
	-52 Resource Failure Retry	Deallocate	***
	+100 Deallocate Normal	Deallocate	Reset

Table C-5 Receive State

Intrinsics You Can Call	Status Info	Local State After Intrinsic Execution	Remote State After Intrinsic Execution
MCTest (POSTED)	0 Successful Completion	Receive	Send
	-37 Posting Not Active	Receive	Send
	-38 Not Posted	Receive	Send
	-51 Resource Failure No Retry	Deallocate	***
	-52 Resource Failure Retry	Deallocate	***
	-56 Prog Error No Truncation	Receive	Send
	-60 Prog Error Data Purged	Receive	Send
	+100 Deallocate Normal	Deallocate	Reset
	-1020 Deallocate Abend	Deallocate	***
MCTest (RequestToSend Received)	Any value	Receive	Send
MCWait	0 Successful Completion	Receive	Send
	-37 Posting Not Active	Receive	Send
	-50 Allocation Error	Deallocate	***
	-51 Resource Failure No Retry	Deallocate	***
	-52 Resource Failure Retry	Deallocate	***
	-56 Prog Error No Truncation	Receive	Send
	-60 Prog Error Data Purged	Receive	Send
	+100 Deallocate Normal	Deallocate	Reset
	-1020 Deallocate Abend	Deallocate	***

Table C-6 shows the Reset State transition table.

Table C-6 Reset State

Intrinsics You Can Call	Status Info	Local State After Intrinsic Execution	Remote State After Intrinsic Execution
MCDeallocate	0 Successful Completion	Send	Receive
	Any other value	Reset	***
MCGetAllocate	0 Successful Completion	Receive	Send
	Any other value	Reset	***

Table C-7 shows the Send State transition table.

Table C-7 Send State

Intrinsics You Can Call	Status Info	Local State After Intrinsic Execution	Remote State After Intrinsic Execution
MCConfirm	0 Successful Completion	Send	Confirm
	-50 Allocation Error	Deallocate	***
	-51 Resource Failure No Retry	Deallocate	***
	-52 Resource Failure Retry	Deallocate	***
	-60 Prog Error Data Purged	Receive	Send
	+80 Timer has expired	Send	Receive
	-1020 Deallocate Abend	Deallocate	***
MCDeallocate (FLUSH)	0 Successful Completion	Reset	Deallocate

Table C-7 Send State

Intrinsics You Can Call	Status Info	Local State After Intrinsic Execution	Remote State After Intrinsic Execution
MCDeallocate (CONFIRM)	0 Successful Completion	Reset	Confirm Deallocate
	-50 Allocation Error	Deallocate	***
	-51 Resource Failure No Retry	Deallocate	***
	-52 Resource Failure Retry	Deallocate	***
	-60 Prog Error Data Purged	Receive	Send
	+80 Timer has expired	Send	Receive
	-1020 Deallocate Abend	Deallocate	***
MCDeallocate (ABEND)	0 Successful Completion	Reset	Deallocate
MCErrMsg	Any value	Send	Receive
MCFlush	Any value	Send	Receive
MCGetAttr	Any value	Send	Receive
MCPrepToRcv (FLUSH)	0 Successful Completion	Receive	Send
MCPrepToRcv (CONFIRM)	0 Successful Completion	Receive	Send
	-50 Allocation Error	Deallocate	***
	-51 Resource Failure No Retry	Deallocate	***
	-52 Resource Failure Retry	Deallocate	***
	-60 Prog Error Data Purged	Receive	Send
	+80 Timer has expired	Receive	***
	-1020 Deallocate Abend	Deallocate	***

Table C-7 Send State

Intrinsics You Can Call	Status Info	Local State After Intrinsic Execution	Remote State After Intrinsic Execution
MCRcvAndWait	0 Successful Completion		
	WhatReceived= DATA_COMPLETE	Receive	Send
	WhatReceived= DATA_INCOMPLETE	Receive	Send
	WhatReceived= SEND	Send	Receive
	WhatReceived= CONFIRM	Confirm	Send
	WhatReceived= CONFIRM_SEND	Confirm Send	Receive
	WhatReceived= CONFIRM_DEALLOCATE	Confirm Deallocate	Deallocate
	-50 Allocation Error	Deallocate	***
	-51 Resource Failure No Retry	Deallocate	***
	-52 Resource Failure Retry	Deallocate	***
	-56 Prog Error No Truncation	Receive	Send
	-60 Prog Error Data Purged	Receive	Send
	+80 Timer has expired	Receive	***
	+100 Deallocate Normal	Deallocate	Reset
	1020 Deallocate Abend	Deallocate	***
MCSendData	0 Successful Completion	Send	Receive
	-50 Allocation Error	Deallocate	***
	-51 Resource Failure No Retry	Deallocate	***
	-52 Resource Failure Retry	Deallocate	***
	-60 Prog Error Data Purged	Receive	Send
	-1020 Deallocate Abend	Deallocate	***

Table C-7 Send State

Intrinsics You Can Call	Status Info	Local State After Intrinsic Execution	Remote State After Intrinsic Execution
MCSendError	0 Successful Completion	Send	Receive
	-50 Allocation Error	Deallocate	***
	-51 Resource Failure No Retry	Deallocate	***
	-52 Resource Failure Retry	Deallocate	***
	-60 Prog Error Data Purged	Receive	Send
	-1020 Deallocate Abend	Deallocate	***
MCTest	Any value	Send	Receive

D LU 6.2 Verb Table

The following Table D-1, is Hewlett-Packard's LU 6.2 API intrinsics to IBM's architected LU 6.2 verbs.

Table D-1 LU 6.2 Verb Table

HP 3000 Intrinsic	LU 6.2 Mapped Conversation Verb
TPStared	none
TPEnded	none
MCAllocate	MC_ALLOCATE
MCConfirm	MC_CONFIRM
MCConfirmed	MC_CONFIRMED
MCDeallocate	MC_DEALLOCATE
(FLUSH)	(FLUSH)
(CONFIRM)	(CONFIRM)
(ABEND)	(ABEND)
(LOCAL)	(LOCAL)
MCFlush	MC_Flush
MCGetAllocate	none
MCGetAttr	MC_GET_ATTRIBUTES
MCPostOnRcpt	MC_POST_ON_RECEIPT
MCPrepToRcv	MC_PREPARE_TO_RECEIVE
MCRcvAndWait	MC_RECEIVE_AND_WAIT
MCRcvNoWait	MC_RECEIVE_IMMEDIATE
MCReqToSend	MC_REQUEST_TO_SEND
MCSendData	MC_SEND_DATA
MCSendError	MC_SEND_ERROR
MCTest	MC_TEST
MCWait	WAIT

E Transact Parameter Masks

LU 6.2 API intrinsics contain optional parameters that may or may not be passed on any given call. Whenever intrinsics that take optional parameters are used, a communication mechanism must exist between the calling program and the intrinsic to indicate which parameters are being passed and which have been omitted. In many languages, this communication mechanism is handled by the compiler. TPs written in the Transact language running on MPE V must provide this communication mechanism by including a **parameter mask** in each intrinsic call that specifies which parameters are being passed and which are being omitted.

NOTE The parameter mask is required in Transact programs only on MPE V. It must not be included in Transact TPs running on MPE XL. When migrating Transact TPs from MPE V to MPE XL, be sure to remove any code pertaining to the parameter masks.

The Parameter Mask

The parameter mask, or bit mask, is a string of bits, each corresponding to an intrinsic parameter. The leftmost bit corresponds to the first parameter in the intrinsic call. If a bit is set to `1`, the corresponding parameter is passed, and if the bit is set to `0` the parameter is omitted. For intrinsics with up to 16 parameters, a 16-bit mask is used. For intrinsics with 17 through 32 parameters, a 32-bit mask is used.

Parameters for Future Expansion

To allow for future expansion on MPE V, all LU 6.2 API intrinsics contain additional parameters that are not documented in this manual. When coding the bit mask, you must account for these “hidden” parameters as well as the documented parameters. In the future, if the hidden parameters are implemented, you will not have to change and re-compile your TP.

Parameter Mask Templates

Table E-1 and Table E-2 describe the parameter mask for each intrinsic. Table E-1 lists the intrinsics that require 16-bit masks

The parameter mask templates for each intrinsic indicate which bits correspond to required, optional, and hidden parameters. Required parameters are represented by 1's, and hidden parameters are represented by 0's. Optional parameters are represented by x's in the templates, and you must replace them with 0's or 1's when you code the parameter mask into your TP. If you are passing an optional parameter, put a 1 in the corresponding bit of the parameter mask, and if you are omitting it, put a 0 in the bit mask.

After you replace every x with a 0 or 1, translate the bit string to a decimal value to be coded into your TP. For intrinsics with no optional parameters, the mask value will always be the same and is given in the “Mask Constant” column.

Table E-1 Intrinsics Requiring a 16-Bit Mask

Intrinsic	Number of Hidden Parameters	Bit Map Template 1 = required x = optional 0 = hidden	Mask Constant
TPStarted	7	111xxxx0000000	n/a
TPEnded	5	1100000	96
MCConfirm	5	11100000	224
MCConfirmed	5	1100000	96
MCDeallocate	5	1x100000	n/a
MCErrMsg	4	11110000	240
MCFlush	5	1100000	96
MCPostOnRcpt	5	111100000	480
MCPrepToRcv	5	11xx00000	n/a
MCRcvAndWait	6	111111000000	4032
MCRcvNoWait	6	111111000000	4032
MCReqToSend	5	1100000	96
MCSendData	7	111110000000	3968
MCSendError	5	11100000	224
MCTest	5	1x1x00000	n/a
MCWait	5	1111x00000	n/a

Table E-2 lists the intrinsics that require 32-bit masks.

Table E-2 Intrinsics Requiring a 32-Bit Mask

Intrinsic	Number of Hidden Parameters	Bit Map Template 1 = required x = optional 0 = hidden	Mask Constant
MCAllocate	3	111111xxxxxxxxxxxxxxxxxxxxxx000	n/a
MCGetAllocate	3	11111xxxxxxxxxxxxxxxxxxxxx000	n/a
MCGetAttr	24	11xxxxx00000000000000000000000	n/a

Using the Parameter Mask in TPs

To use the parameter mask in the TP you must do the following:

1. Declare both the 16-bit and 32-bit parameter masks at the beginning of the program.
2. Assign the appropriate number to the parameter mask in a LET statement before each LU 6.2 API intrinsic call. The number in the parameter mask must be the decimal representation of the bit mask described in "Parameter Mask Templates."
3. Include place holders for the hidden parameters in the intrinsic call. Commas serve as place holders in Transact.
4. List the parameter mask in the intrinsic call as the last parameter passed. The parameter mask is passed by value.

Examples

Following are examples of LU 6.2 API intrinsic calls in Transact. Optional parameters are marked, and they must have their mask bits set to 0 or 1, depending on whether or not they are passed. All commas shown are required. Parameters common to several intrinsics appear in the example declarations for every intrinsic that uses them. These parameters need to be declared only once in a program.

Example E-1 Example Parameter Mask Declaration

```
< Declarations of bit masks and status parameter >>

DEFINE(ITEM) MASK16 I(4):       << 16-bit parameter mask >>
             MASK32 I(9):       << 32-bit parameter mask >>
             STATUS I(9):
               INFO I(4) = STATUS(I):
               INFO I(4) = STATUS (3);

LIST MASK16: MASK32: STATUS;


------------------------------------------------------------------------
```

Example E-2 TPStarted Declarations and Intrinsic Call

```
DEFINE(ITEM) LOCAL-TP-NAME X(8):
             TP-ID I(4):
             TRACE-ON I(4):
             TRACE-SIZE I(4):
             TRACE-FILE X(35):
             DEFAULT-FILE X(28);

@COMPUTERTEXTW = LIST LOCAL-TP-NAME: TP-ID: TRACE-ON: TRACE-SIZE: TRACE-FILE:
DEFAULT-FILE;

LET (MASK16) = 15872; << "11111000000000" >>

PROC TPSTARTED ( %(LOCAL-TP-NAME),
                  (TP-ID),
                  (STATUS),
                  (TRACE-ON),         << optional >>
                 #(TRACE-SIZE),       << optional >>
               << %(TRACE-FILE),         optional, not used >>
               << %(DEFAULT-FILE),       optional, not used >>
                  , , , , , , ,       << 7 hidden parameters >>
                 #(MASK16) );
```

Example E-3 TPEnded Declarations and Intrinsic Call

```
DEFINE(ITEM) TP-ID I(4);

LIST TP-ID;

LET (MASK16) = 96; << "1100000" >>

PROC TPENDED ( #(TP-ID),
                (STATUS),
                , , , , ,          << 5 hidden parameters >>
               #(MASK16) );
```

Example E-4 MCAllocate Declarations and Intrinsic Call

```
DEFINE(ITEM) TP-ID I(4):
             SESSION-TYPE X(8):
             REMOTE-TP-NAME X(64):
             REMOTE-TP-LEN I(4):
             RESOURCE-ID I(4):
             RETURN-CONTROL I(4):
             SYNC-LEVEL I(4):
             TIMER I(4):
             SECURITY I(4):
             NUM-PIPS I(4):
             PIP-LENGTHS[16] I(4):
             PIP1 X(1980):
             PIP2 X(1980):     << The lengths given in the PIP >>
             PIP3 X(1980):     << declarations may vary from   >>
             PIP4 X(1980):     << 0 to 1980 bytes.             >>
             PIP5 X(1980):
             PIP6 X(1980):     << The maximum combined length  >>
             PIP7 X(1980):     << of all PIP parameters is     >>
             PIP8 X(1980):     << 1980 bytes.                  >>
             PIP9 X(1980):
             PIP10 X(1980):
             PIP11 X(1980):
             PIP12 X(1980):
             PIP13 X(1980):
             PIP14 X(1980):
             PIP15 X(1980):
             PIP16 X(1980);

LIST TP-ID: SESSION-TYPE: REMOTE-TP-NAME: REMOTE-TP-LEN:
     RESOURCE-ID: RETURN-CONTROL: SYNC-LEVEL: TIMER:
     SECURITY: NUM-PIPS: PIP-LENGTHS: PIP1: PIP2: PIP3:
     PIP4: PIP5: PIP6: PIP7: PIP8: PIP9: PIP10: PIP11:
     PIP12: PIP13: PIP14: PIP15: PIP16;

LET (MASK32) = 2115960832; << "11111100001111100000000000000000" >>

PROC MCALLOCATE ( #(TP-ID),
                  %(SESSION-TYPE),
                  %(REMOTE-TP-NAME),
                  #(REMOTE-TP-LEN),
                   (RESOURCE-ID),
                   (STATUS),
                << #(RETURN-CONTROL),   << optional, not used >>
                << #(SYNC-LEVEL),       << optional, not used >>
                << #(TIMER),            << optional, not used >>
                << #(SECURITY),         << optional, not used >>
                   #(NUM-PIPS),         << optional >>
                    (PIP-LENGTHS),      << optional >>
                   %(PIP1),             << optional >>
                   %(PIP2),             << optional >>
```

```
                  %(PIP3),              << optional >>
                << %(PIP4)   >>,        << optional, not used >>
                << %(PIP5),             << optional, not used >>
                << %(PIP6),             << optional, not used >>
                << %(PIP7),             << optional, not used >>
                << %(PIP8),             << optional, not used >>
                << %(PIP9),             << optional, not used >>
                << %(PIP10),            << optional, not used >>
                << %(PIP11),            << optional, not used >>
                << %(PIP12),            << optional, not used >>
                << %(PIP13),            << optional, not used >>
                << %(PIP14),            << optional, not used >>
                << %(PIP15),            << optional, not used >>
                << %(PIP16),            << optional, not used >>
                   , , ,                    << 3 hidden parameters >>
                  #(MASK32) );
```

Example E-5 MCConfirm Declarations and Intrinsic Call

```
DEFINE(ITEM) RESOURCE-ID I(4):
             REQUEST-TO-SEND-RECEIVED I(4);

LIST RESOURCE-ID: REQUEST-TO-SEND-RECEIVED;

LET (MASK16) = 224; << "11100000" >>

PROC MCCONFIRM ( #(RESOURCE-ID),
                  (REQUEST-TO-SEND-RECEIVED),
                  (STATUS),
                  , , , , ,          << 5 hidden parameters >>
                 #(MASK16) );
```

Example E-6 MCConfirmed Declarations and Intrinsic Call

```
DEFINE(ITEM) RESOURCE-ID I(4);

LIST RESOURCE-ID;

LET (MASK16) = 96; << "1100000" >>

PROC MCCONFIRMED ( #(RESOURCE-ID),
                    (STATUS),
                    , , , , ,          << 5 hidden parameters >>
                   #(MASK16) );
```

Example E-7 MCDeallocate Declarations and Intrinsic Call

```
DEFINE(ITEM) RESOURCE-ID I(4):
             DEALLOCATE-TYPE I(4);

LIST RESOURCE-ID: DEALLOCATE-TYPE;

LET (MASK16) = 160; << "10100000" >>


@COMPUTERTEXTW = PROC MCDEALLOCATE ( #(RESOURCE-ID),
                 << #(DEALLOCATE-TYPE),    optional, not used >>
                     (STATUS),
                     , , , , ,          << 5 hidden parameters >>
                    #(MASK16) );
```

Example E-8 MCErrMsg Declarations and Intrinsic Call

```
DEFINE(ITEM) OLD-STATUS I(9):
             MESSAGE-BUFFER X(256):
             MESSAGE-LENGTH I(4);

LIST OLD-STATUS: MESSAGE-BUFFER: MESSAGE-LENGTH;

LET (MASK16) = 240; << "11110000" >>

PROC MCERRMSG ( #(OLD-STATUS),
                %(MESSAGE-BUFFER),
                 (MESSAGE-LENGTH),
                 (STATUS),
                 , , , ,            << 4 hidden parameters >>
                #(MASK16) );
```

Example E-9 MCFlush Declarations and Intrinsic Call

```
DEFINE(ITEM) RESOURCE-ID I(4);

LIST RESOURCE-ID;

LET (MASK16) = 96; << "1100000" >>

PROC MCFLUSH ( #(RESOURCE-ID),
                (STATUS),
                , , , , ,          << 5 hidden parameters >>
               #(MASK16) );
```

Example E-10 MCGetAllocate Declarations and Intrinsic Call

```
DEFINE(ITEM) TP-ID I(4):
             SESSION-TYPE X(8):
             TP-NAME X(64):
             RESOURCE-ID I(4):
             SYNC-LEVEL I(4):
             TIMER I(4):
             SECURITY I(4):
             NUM-PIPS I(4):
             PIP-LENGTHS[16] I(4):
             PIP1 X(1980):
             PIP2 X(1980):        << The lengths given in the PIP >>
             PIP3 X(1980):        << declarations may vary from   >>
             PIP4 X(1980):        << 0 to 1980 bytes.             >>
             PIP5 X(1980):
             PIP6 X(1980):        << The maximum combined length  >>
             PIP7 X(1980):        << of all PIP parameters is     >>
             PIP8 X(1980):        << 1980 bytes.                  >>
             PIP9 X(1980):
             PIP10 X(1980):
             PIP11 X(1980):
             PIP12 X(1980):
             PIP13 X(1980):
             PIP14 X(1980):
             PIP15 X(1980):
             PIP16 X(1980);

LIST TP-ID: SESSION-TYPE: TP-NAME: RESOURCE-ID:
     SYNC-LEVEL: TIMER: SECURITY: NUM-PIPS: PIP-LENGTHS:
     PIP1: PIP2: PIP3: PIP4: PIP5: PIP6: PIP7: PIP8: PIP9:
     PIP10: PIP11: PIP12: PIP13: PIP14: PIP15: PIP16;

LET (MASK32) = 522125312; << "11111000111110000000000000000" >>


PROC MCGETALLOCATE ( #(TP-ID),
                     %(SESSION-TYPE),
                     %(TP-NAME),
                      (RESOURCE-ID),
                      (STATUS),
                   << #(SYNC-LEVEL),      optional, not used >>
                   << #(TIMER),           optional, not used >>
                   << #(SECURITY),        optional, not used >>
                      #(NUM-PIPS),     << optional >>
                      #(PIP-LENGTHS),  << optional >>
                      %(PIP1),         << optional >>
                      %(PIP2),         << optional >>
                      %(PIP3),         << optional >>
                   << %(PIP4),            optional, not used >>
                   << %(PIP5),            optional, not used >>
                   << %(PIP6),            optional, not used >>
                   << %(PIP7),            optional, not used >>
                   << %(PIP8),            optional, not used >>
```

```
                  << %(PIP9),            optional, not used >>
                  << %(PIP10),           optional, not used >>
                  << %(PIP11),           optional, not used >>
                  << %(PIP12),           optional, not used >>
                  << %(PIP13),           optional, not used >>
                  << %(PIP14),           optional, not used >>
                  << %(PIP15),           optional, not used >>
                  << %(PIP16),           optional, not used >>
                     , , ,            << 3 hidden parameters >>
                    #(MASK32) );
```

Example E-11 MCGetAttr Declarations and Intrinsic Call

```
DEFINE(ITEM) RESOURCE-ID I(4):
             OWN-FULLY-QUALIFIED-LU-NAME X(17):
             PARTNER-LU-NAME X(8):
             PARTNER-FULLY-QUALIFIED-LU-NAME X(17):
             MODE-NAME X(8):
             SYNC-LEVEL I(4);


@COMPUTERTEXTW = LIST RESOURCE-ID: OWN-FULLY-QUALIFIED-LU-NAME:
     PARTNER-LU-NAME: PARTNER-FULLY-QUALIFIED-LU-NAME:
     MODE-NAME: SYNC-LEVEL;

LET (MASK32) = 1795162112; << "11010110000000000000000000000000" >>

PROC MCGETATTR ( #(RESOURCE-ID),
                   (STATUS),
                << %(OWN-FULLY-QUALIFIED-LU-NAME),      optional, not used >>
                   %(PARTNER-LU-NAME),               << optional >>
                << %(PARTNER-FULLY-QUALIFIED-LU-NAME),  optional, not used >>
                   %(MODE-NAME),                      << optional >>
                    (SYNC-LEVEL),                     << optional >>
                    , , , , , ,
                    , , , , , ,
                    , , , , , ,
                    , , , , , ,                  << 24 hidden parameters >>
                   #(MASK32) );
```

Example E-12 MCPostOnRcpt Declarations and Intrinsic Call

```
DEFINE(ITEM) RESOURCE-ID I(4):
             LENGTH I(4):
             DATA X(4092);

LIST RESOURCE-ID: LENGTH: DATA;

LET (MASK16) = 480; << "111100000" >>

PROC MCPOSTONRCPT ( #(RESOURCE-ID),
                    #(LENGTH),
                    %(DATA),
                     (STATUS),
                     , , , , ,          << 5 hidden parameters >>
                    #(MASK16) );
```

Example E-13 MCPrepToRcv Declarations and Intrinsic Call

```
DEFINE(ITEM) RESOURCE-ID I(4):
             PREP-TO-RCV-TYPE I(4):
             LOCKS I(4);

LIST RESOURCE-ID: PREP-TO-RCV-TYPE: LOCKS;

LET (MASK16) = 384; << "110000000" >>

PROC MCPREPTORCV ( #(RESOURCE-ID),
                      (STATUS),
                  << #(PREP-TO-RCV-TYPE),    optional, not used >>
                  << #(LOCKS),               optional, not used >>
                      , , , , ,          << 5 hidden parameters >>
                     #(MASK16) );
```

Example E-14 MCRcvAndWait Declarations and Intrinsic Call

```
DEFINE(ITEM) RESOURCE-ID I(4):
             LENGTH I(4):
             REQUEST-TO-SEND-RECEIVED I(4):
             DATA X(4092):
             WHAT-RECEIVED I(4);

LIST RESOURCE-ID: LENGTH: REQUEST-TO-SEND-RECEIVED:
     DATA: WHAT-RECEIVED;

LET (MASK16) = 4032; << "111111000000" >>

PROC MCRCVANDWAIT ( #(RESOURCE-ID),
                     (LENGTH),
                     (REQUEST-TO-SEND-RECEIVED),
                    %(DATA),
                     (WHAT-RECEIVED),
                     (STATUS),
                     , , , , , ,        << 6 hidden parameters >>
                    #(MASK16) );
```

Example E-15 MCRcvNoWait Declarations and Intrinsic Call

```
DEFINE(ITEM) RESOURCE-ID I(4):
             LENGTH I(4):
             REQUEST-TO-SEND-RECEIVED I(4):
             DATA X(4092):
             WHAT-RECEIVED I(4);

LIST RESOURCE-ID: LENGTH: REQUEST-TO-SEND-RECEIVED:
     DATA: WHAT-RECEIVED;

LET (MASK16) = 4032; << "111111000000" >>

PROC MCRCVNOWAIT ( #(RESOURCE-ID),
                    (LENGTH),
                    (REQUEST-TO-SEND-RECEIVED),
                   %(DATA),
                    (WHAT-RECEIVED),
                    (STATUS),
                    , , , , , ,        << 6 hidden parameters >>
                   #(MASK16) );
```

Example E-16 MCReqToSend Declarations and Intrinsic Call

```
DEFINE(ITEM) RESOURCE-ID I(4);

LIST RESOURCE-ID;

LET (MASK16) = 96; << "1100000" >>

PROC MCREQTOSEND ( #(RESOURCE-ID),
                   (STATUS),
                    , , , , ,          << 5 hidden parameters >>
                   #(MASK16) );
```

Example E-17 MCSendData Declarations and Intrinsic Call

```
DEFINE(ITEM) RESOURCE-ID I(4):
             DATA X(4092):
             LENGTH I(4):
             REQUEST-TO-SEND-RECEIVED I(4);

LIST RESOURCE-ID: DATA: LENGTH: REQUEST-TO-SEND-RECEIVED;

LET (MASK16) = 3968; << "111110000000" >>

PROC MCSENDDATA ( #(RESOURCE-ID),
                  %(DATA),
                  #(LENGTH),
                   (REQUEST-TO-SEND-RECEIVED),
                   (STATUS),
                    , , , , , , ,      << 7 hidden parameters >>
                  #(MASK16) );
```

Example E-18 MCSendError Declarations and Intrinsic Call

```
DEFINE(ITEM) RESOURCE-ID I(4):
             REQUEST-TO-SEND-RECEIVED I(4);

LIST RESOURCE-ID: REQUEST-TO-SEND-RECEIVED;

LET (MASK16) = 224; << "11100000" >>

PROC MCSENDERROR ( #(RESOURCE-ID),
                    (REQUEST-TO-SEND-RECEIVED),
                    (STATUS),
                    , , , , ,          << 5 hidden parameters >>
                   #(MASK16) );
```

Example E-19 MCTest Declarations and Intrinsic Call

```
DEFINE(ITEM) RESOURCE-ID I(4):
             TEST I(4):
             POSTED-TYPE I(4);

LIST RESOURCE-ID: TEST: POSTED-TYPE;

LET (MASK16) = 480; << "111100000" >>

PROC MCTEST ( #(RESOURCE-ID),
              #(TEST),            << optional >>
               (STATUS),
               (POSTED-TYPE),     << optional >>
               , , , , ,          << 5 hidden parameters >>
              #(MASK16) );
```

Example E-20 MCWait Declarations and Intrinsic Call

```
DEFINE(ITEM) RESOURCE-LIST[8] I(4):
             NUM-RESOURCES I(4):
             RESOURCE-POSTED I(4):
             POSTED-TYPE I(4);

LIST RESOURCE-LIST: NUM-RESOURCES: RESOURCE-POSTED:
     POSTED-TYPE;

LET (MASK16) = 992; << "1111100000" >>

PROC MCWAIT (  (RESOURCE-LIST),
              #(NUM-RESOURCES),
               (RESOURCE-POSTED),
               (STATUS),
               (POSTED-TYPE),     << optional >>
               , , , , ,          << 5 hidden parameters >>
              #(MASK16) );
```

F Migrating Transaction Programs

This appendix describes any source code changes you might have to make in order to migrate a TP from LU 6.2 API/V to LU 6.2 API/XL (or from an early version of LU 6.2 API/XL to the version that supports Node Type 2.1).

You must change your TPs to migrate them to LU 6.2 API/XL if any of the following is true:

1. Your TPs issue `APPCCONTROL` commands programmatically.
2. Your TPs call the `MCGetAllocate` intrinsic.
3. Your TPs are written in Transact on MPE V.

General information on migrating COBOL II and Pascal applications from MPE V to MPE XL can be found in the *MPE XL Languages Migration Guides*.

TPs that Issue APPCCONTROL Commands

If your TPs start or stop the APPC subsystem or change the number of active sessions programmatically, you must change them to run on the Node Type 2.1 version of LU 6.2 API/XL. On the Node Type 2.1 version, TPs cannot call the MPE COMMAND intrinsic to issue APPCCONTROL commands programmatically, because APPCCONTROL commands are not interpreted by the MPE command interpreter.

A command interpreter for APPCCONTROL commands is installed with the Node Type 2.1 version of LU 6.2 API/XL. A system UDC file, which is installed with the product, translates APPCCONTROL commands into MPE RUN commands that invoke the APPCCONTROL command interpreter.

Control Operator Intrinsics

If you want to start or stop the APPC subsystem or change the number of active sessions programmatically, Hewlett-Packard recommends that you use the control operator intrinsics instead of APPCCONTROL commands in your transaction programs. Unlike APPCCONTROL commands, control operator intrinsics will return status information to programs that call them. The control operator intrinsics are documented in the *APPC Subsystem on MPE XL Node Manager's Guide*.

MPE HPCICOMMAND Intrinsic

If necessary, you can still issue APPCCONTROL commands programmatically. However, because APPCCONTROL commands are implemented with UDCs, you must use the MPE HPCICOMMAND intrinsic instead of the COMMAND intrinsic. See the *MPE XL Intrinsics Reference Manual*.

TRACEON Parameter of APPCCONTROL START

If your TPs issue the APPCCONTROL START command, you cannot specify the *TRACEOFF* parameter, because it is not supported on the Node Type 2.1 version of LU 6.2 API/XL. On the Node Type 2.1 version, APPC subsystem internal tracing is turned off by default, and the *TRACEON* parameter of the APPCCONTROL START command turns it on at subsystem startup. You must remove the *TRACEOFF* parameter of the APPCCONTROL START command from all transaction programs and job streams.

Remotely Initiated TPs

On all versions of LU 6.2 API prior to the Node Type 2.1 version, whenever a remote TP sends an allocate request to initiate a conversation with a local TP, LU 6.2 API streams a job that runs the local TP. This method of starting up remotely initiated TPs can be very slow, because a job must be streamed every time an allocate request is received from a remote TP.

The Node Type 2.1 version of LU 6.2 API/XL allows a single TP process on the HP 3000 to receive multiple allocate requests from remote TPs. Each remotely initiated local TP must be configured through NMMGR. The configuration file specifies whether a TP is to receive multiple or single allocate requests.

A local TP configured to receive multiple allocate requests from remote TPs is started up only once. Allocate requests for that TP are queued, and the TP must make multiple calls to the `MCGetAllocate` intrinsic in order to receive all the allocate requests.

A local TP configured to receive a single allocate request is started up every time an allocate request is received from the remote TP. Multiple instances of it may be running at once, and each instance must call the `MCGetAllocate` intrinsic only once.

For information on TP configuration, see the APPC Subsystem on *MPE XL Node Manager's Guide*.

Source Code Changes to TPs

You must change your remotely initiated TPs in the following ways in order to run them on the Node Type 2.1 version of LU 6.2 API/XL:

1. Change the *`LocalTPName`* parameter of the `MCGetAllocate` intrinsic from an output parameter to an input parameter. Instead of receiving the *`LocalTPName`* from the intrinsic, your program must pass the *`LocalTPName`* to the intrinsic. Chapter 5 , "Intrinsic Descriptions," of this manual contains a complete description of the `MCGetAllocate` intrinsic.
2. Make sure the *`LocalTPName`* parameter of the `TPStarted` intrinsic matches the *`LocalTPName`* parameter of the `MCGetAllocate` intrinsic. For older versions of LU 6.2 API, these parameters did not need to match, but for the Node Type 2.1 version of LU 6.2 API/XL, they must match.
3. Have the node manager configure the remotely initiated TP through NMMGR/XL. The *`LocalTPName`* parameter of the `MCGetAllocate` and `TPStarted` intrinsics must match a configured TP name in the APPC subsystem configuration. See the *APPC Subsystem on*

MPE XL Node Manager's Guide for information on TP configuration.

The remote TP must send this configured TP name in the allocate request. In older versions of LU 6.2 API, the remote TP sends the name of the job file that runs the local TP. In order to avoid changing the remote TP, make the configured TP name (and the *LocalTPName* parameter) match the job file name.

4. If you want your TPs to receive multiple (queued) allocate requests from remote TPs, modify them to call the MCGetAllocate intrinsic multiple times — once for each allocate request.

   You can write a TP to receive a predetermined number of allocate requests, or you can write it to loop through the conversation intrinsics, from MCGetAllocate to MCDeallocate, until no more allocate requests arrive from the remote system. (See the MCGetAllocate intrinsic description in Chapter 5 , "Intrinsic Descriptions.")

   A time-out value for the MCGetAllocate intrinsic may be configured through NMMGR. If no allocate request arrives from the remote TP before the time-out value expires, the MCGetAllocate intrinsic returns with a status info value of +29. If you want your TP to receive an unknown number of allocate requests, you can design it to loop through the conversation intrinsics until +29 is returned in the *Status* parameter of the MCGetAllocate intrinsic. See the *APPC Subsystem on MPE XL Node Manager's Guide* for information on configuring the time-out value.

TPs In Transact

On MPE V, TPs written in Transact must pass a parameter mask in each intrinsic call, telling the intrinsic which parameters are being passed and which are being omitted. When you migrate a Transact TP from MPE V to MPE XL, you must remove the parameter mask from all intrinsic calls. The parameter mask is described in Appendix E , "Transact Parameter Masks."

Glossary

A

Advanced Program-to-Program Communication (APPC):

Programmatic communication based on IBM's LU 6.2 architecture. APPC provides partner programs with a common set of rules for communication.

Application Program Interface (API): A set of subprograms, callable from inside applications, that carry out data communications tasks.

B

basic conversation: A programmatic conversation in which the applications must be able to create and interpret GDS headers (*see* **GDS header**) for transmitting and receiving data.

basic conversation verbs: the programmatic implementation of functions and protocols in a basic conversation between transaction programs. (*See* **mapped conversation verbs**.)

C

cluster controller: A machine that allows multiple devices to send and receive data over the same communications link.

communications controller: A device that controls network data traffic for the hosts in the network.

Confirm state: The conversation state from which a TP can reply to a confirmation request and enter Receive state.

Confirm Deallocate state: The conversation state from which a TP can reply to a confirmation request and enter Deallocate state.

Confirm Send state: The conversation state from which a TP can reply to a confirmation request and enter Send state.

confirmation request: A request sent by a TP, asking its partner TP to confirm the receipt of data.

control information:

Information exchanged by TPs to control conversations. Examples are requests for conversation allocation and deallocation, confirmation requests, and error notifications

conversation: The logical communication between two transaction programs.

conversation states: The conditions of programmatic conversations under which certain activities can occur. For example, if one side of a conversation is in the condition of Send state, it can send data, but it cannot receive data until it changes to the condition of Receive state.

conversation with confirm: A conversation established in such a way that confirmation requests and responses can be sent and received.

conversation without confirm: A conversation established in such a way that confirmation requests and responses cannot be sent or received.

Customer Information Control System (CICS): An IBM application subsystem that provides file handling and data communications services for application programs.

D

Deallocate state: The conversation state from which a TP can deallocate the mapped conversation locally.

dependent LU: An LU capable of conducting only one LU-LU session at a time. A dependent LU always functions as a secondary LU and cannot send a BIND to initiate an LU-LU session; to request a session, it must send an INIT_SELF to the host and wait for the host to send the BIND. *See* **independent LU**.

E

end user: The ultimate destination of data in a communications network. An end user can be a human user, a peripheral device (like a printer or a terminal), or an application program.

G

Generalized Data Stream (GDS): The name of the LU 6.2 data stream. LU 6.2 data packets must include GDS headers (*see* **GDS heade**r).

GDS header A portion of an LU 6.2 data packet that contains information about the kind of data being sent or received.

I

independent LU: An LU capable of conducting multiple, simultaneous (parallel) APPC sessions with another independent LU on a remote system. An independent LU can function as either a primary or secondary LU. *See* **dependent LU**.

intrinsic: A subprogram provided by Hewlett-Packard to perform common functions such as opening files, opening communications lines, performing subsystem-defined functions, or transmitting data over a communications line.

L

local TP: The TP running on the local processor.

Logical Unit (LU): The SNA entity through which application data is transmitted within an SNA network. Logical Units are the ports through which end users have access to the network (see end user).

LU type: A Logical Unit type, defined by SNA to perform a particular type of communication.

LU-LU session: *See* **session**.

LU 6.2: An SNA LU type which defines the communication that can take place between two application programs on separate processors. LU 6.2 includes specifications for programmatic interfaces, document interchange, and data distribution.

M

mapped conversation: A programmatic conversation in which the application is freed from handling the GDS headers required by LU 6.2 architecture.

mapped conversation verbs:

The programmatic implementation of functions and protocols in a mapped conversation between transaction programs. (*See* **basic conversation verbs**.)

N

node type: A node type defined by SNA to perform a particular type of communication. Some common SNA node types are defined as follows:

Node Type 2.0: the node type for a peripheral node or cluster controller. Node Type 2.0 is supported by LU 6.2 API on MPE V and MPE XL.

Node Type 2.1: the node type for a peripheral node or cluster controller capable of peer-to-peer communication. Node Type 2.1 is supported by LU 6.2 API on MPE XL.

Node Type 4: the node type for a subarea node with a communications controller.

Node Type 5: the node type for a host node with an SSCP (System Services Control Point).

O

one-way conversation: A conversation in which data is sent from only one TP.

P

parameter mask: A bit mask that must be passed in intrinsic calls in Transact on MPE V. The parameter mask tells the intrinsic which parameters are being passed and which are being omitted.

peripheral: A device on the network.

Physical Unit: An SNA term that refers to the software and hardware that controls the resources of a node.

PU: *See* **Physical Unit**.

PU 2.0: *See* **Node Type 2.0.**

PU 2.1: *See* **Node Type 2.1**.

R

Receive state: The conversation state from which a TP can receive information from the remote TP.

remote TP: The TP running on the remote processor.

Reset state: The conversation state from which a TP can allocate a mapped conversation.

S

Send state: The conversation state from which a TP can send data or request confirmation.

session: The logical connection between two logical devices in an SNA network.

Systems Network Architecture (SNA): IBM's comprehensive specification for data communications networks.

synchronization level: A term that refers to the amount of synchronization information (confirmation requests and responses) that can be sent in a conversation.

T

transaction program (TP): An application program that processes distributed transactions.

two-way conversation: A conversation in which data is sent and received by both TPs.

U

UDC: User-Defined Command. An MPE feature that allows a user to create file of commands to be executed as a single program when the user types a command. UDCs are set with the MPE SETCATALOG command

V

verbs: The programmatic implementation of functions and protocols in a conversation between transaction programs.

Index