z/OS
Version 2 Release 4

MVS Programming: Writing Servers for APPC/MVS

SA23-1396-40

Note

Before using this information and the product it supports, read the information in “Notices” on page 109.

This edition applies to Version 2 Release 4 of z/OS (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2019-07-10

Contents

Figures

Tables

About This Book

APPC/MVS is an implementation of IBM's Advanced Program-to-Program Communication (APPC) in the MVS operating system. APPC/MVS allows MVS application programs to communicate on a peer-to-peer basis with other application programs on the same z/OS system, different z/OS systems, or different operating systems (including Microsoft Windows®, Sun Solaris, AIX, OS/400, OS/2, and VM in an SNA network. These communicating programs, known as *transaction programs* (TPs) and *servers*, together form cooperative processing applications that can exploit the strengths of different computer architectures. This book tells how to design and write APPC servers to run on MVS. A companion book, *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*, tells how to design and write APPC/MVS transaction programs.

In this book, the term *APPC/MVS transaction program* refers to a program, running in an MVS address space, that uses APPC/MVS services. The term *transaction* is not restricted to programs scheduled by the APPC/MVS transaction scheduler, or to programs using APPC/MVS services.

In this book, the term *APPC/MVS server* refers to a specific type of transaction program; one that can manage multiple inbound LU 6.2 conversations from multiple client transaction programs, serially or concurrently. The client programs may be running on the same system or on other systems in the SNA network (such as an OS/2 system running on a workstation).

Note that APPC/MVS transaction programs and servers are parts of cooperative processing applications and are distinct from, but coexistent and compatible with, CICS and IMS transaction processing applications.

This book is a companion to *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*, which describes callable services that are available to both APPC/MVS transaction programs and servers. This book describes the callable services that are useful only to APPC/MVS servers. Servers use these services to establish and manage one or more queues of inbound allocate requests from the installation's transaction programs. These queues are referred to as **allocate queues**. The services used to create and manage allocate queues are known as **allocate queue services**.

Who Should Use This Book

This book is written for application programmers who use APPC/MVS application programming interfaces (APIs) to design and code applications. The book assumes the user understands the basic concepts of APPC/MVS, and can code in one or more high-level languages (HLLs) that support APPC/MVS servers. For a list of the HLLs that can be used to code APPC/MVS servers, see Table 6 on page 33.

How to Use This Book

This book is one of the set of programming books for MVS. This set describes how to write programs in assembler language or high-level languages, such as C, FORTRAN, and COBOL. For more information about the content of this set of books, see *z/OS Information Roadmap*.

Where to Find More Information

Where necessary, this book references information in other books, using the shortened version of the book title. For complete titles and order numbers of the books for all products that are part of z/OS, see *z/OS Information Roadmap*. The following table lists the titles and order numbers of books for other IBM products.

Short Title Used in This Book	Title	Order Number
AS/400 APPC Programmer's Guide	*AS/400 Communications: Advanced Program-to-Program Communication Programmer's Guide*	SC41-8189
CPI-C Reference	*Common Programming Interface Communications Reference*	SC26-4399
OS/2 APPC Programming Guide and Reference	*OS/2 APPC Programming Guide and Reference*	SC31-6160
SNA Formats	*SNA Formats*	GA27-3136
SNA Transaction Programmer's Reference Manual for LU 6.2	*SNA Transaction Programmer's Reference Manual for LU 6.2*	SC31-6808
SNA Technical Overview	*SNA Technical Overview*	GC30-3073
VM/ESA CP Programming Services	*VM/ESA CP Programming Services*	SC24-5520

How to send your comments to IBM

We invite you to submit comments about the z/OS® product documentation. Your valuable feedback helps to ensure accurate and high-quality information.

Important: If your comment regards a technical question or problem, see instead "If you have a technical problem" on page xiii.

Submit your feedback by using the appropriate method for your type of comment or question:

Feedback on z/OS function
: If your comment or question is about z/OS itself, submit a request through the IBM RFE Community (www.ibm.com/developerworks/rfe/).

Feedback on IBM® Knowledge Center function
: If your comment or question is about the IBM Knowledge Center functionality, for example search capabilities or how to arrange the browser view, send a detailed email to IBM Knowledge Center Support at ibmkc@us.ibm.com.

Feedback on the z/OS product documentation and content
: If your comment is about the information that is provided in the z/OS product documentation library, send a detailed email to mhvrcfs@us.ibm.com. We welcome any feedback that you have, including comments on the clarity, accuracy, or completeness of the information.

To help us better process your submission, include the following information:

- Your name, company/university/institution name, and email address
- The following deliverable title and order number: z/OS MVS Writing Servers for APPC/MVS, SA23-1396-40
- The section title of the specific information to which your comment relates
- The text of your comment.

When you send comments to IBM, you grant IBM a nonexclusive authority to use or distribute the comments in any way appropriate without incurring any obligation to you.

IBM or any other organizations use the personal information that you supply to contact you only about the issues that you submit.

If you have a technical problem

If you have a technical problem or question, do not use the feedback methods that are provided for sending documentation comments. Instead, take one or more of the following actions:

- Go to the IBM Support Portal (support.ibm.com).
- Contact your IBM service representative.
- Call IBM technical support.

Summary of changes

This information includes terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations for the current edition are indicated by a vertical line to the left of the change.

Summary of changes for z/OS MVS Programming: Writing Servers for APPC/MVS for Version 2 Release 4 (V2R4)

This information contains no technical changes for this release.

Summary of changes for z/OS MVS Programming: Writing Servers for APPC/MVS for Version 2 Release 3 (V2R3)

This information contains no technical changes for this release.

Chapter 1. Introduction

Objective

This chapter is intended to help users decide whether writing an APPC/MVS server is the best solution to meeting a particular goal.

Distributed Applications

In a distributed processing environment, a programmer is free to use the strengths of different operating systems to meet the particular needs of the installation. In the past, programmers were forced to accept whatever unique limitations existed for the particular operating system for which they developed applications. Creativity, as a result, was hampered.

APPC supports distributed processing across PC, midrange, and mainframe operating systems. Using APPC, programmers can write applications that take advantage of the different strengths of each operating system. Such applications are called *distributed applications*.

For example, an application that formerly ran to completion entirely on a mainframe computer, can now be divided so that one part remains on the mainframe and the other part runs on an workstation. The application writer can then take advantage of the workstation's graphical user interface and the mainframe computer's number-crunching and database processing capabilities.

For distributed processing to work, the programs running on different operating systems must be able to communicate with each other effectively. APPC makes it possible for programs running on different operating systems to "speak the same language." Operating systems such as z/OS, OS/400, OS/2, and VM have a common set of communication services. Programs that reside on these operating systems can use the services (or verbs), to "converse" with each other across a SNA network.

Applications "connect" to the SNA network through one or more logical units (LUs) installed on the operating system. To fully participate in conversations across a SNA network, applications require a specific type of LU known as LU 6.2. With LU 6.2, the operating system is said to support APPC (Advanced Program-to-Program Communications).

The Client/Server Model

The term "client/server" describes a type of distributed processing in which an application is divided into two parts, each possibly residing on separate operating systems, but working together to provide a service to the end user. As shown in Figure 1 on page 2, one part of the application, the client, typically resides on a workstation and requests a service for the end-user. The other part of the application, the server, usually resides on a larger machine, such as a mainframe computer. The server program uses the resources of the mainframe computer to perform services requested by each client.

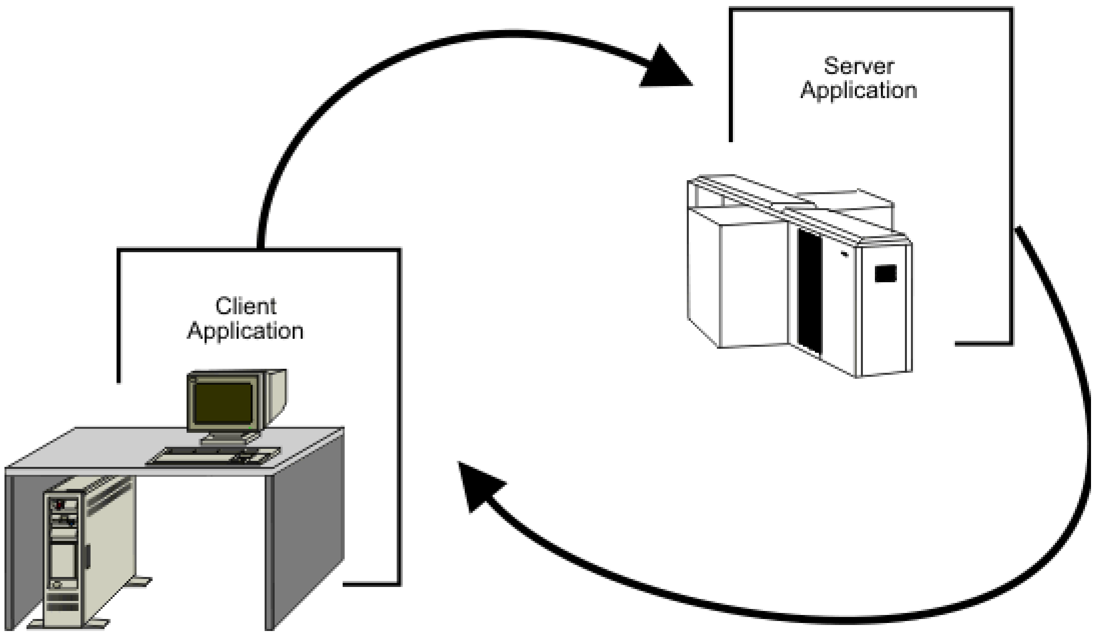

Figure 1. The Client/Server Computing Environment

Depending on how it is designed, a server can process requests from multiple clients concurrently. Generally, there is one server for many clients.

The client is usually the part of the application that is "seen" by the end-user. Therefore, the client half of a client/server application most often resides on a workstation, where the end-user can interact with the application through the workstation operating system's graphical user interface.

Servers, on the other hand, are usually transparent to the end-user. That is, the person who sits at the workstation only perceives the client half of the application, the part that displays the information (though it was actually retrieved by a remote server). Because there is only one server for a given set of clients, servers provide an ideal way of managing shared access to system resources, such as data sets. For this reason, servers are likely to reside on larger machines such as z/OS mainframe computers.

Usually, the same person writes both the client and server parts of a client/server application.

This book explains how to write servers for an APPC/MVS environment. These servers can be written to serve client programs (also called **client transaction programs**, or **client TPs**).

For information about writing client programs to interact with APPC/MVS servers, see the appropriate programming manual for the particular operating system on which the client program will reside.

If you are writing an APPC application that combines z/OS servers with client transaction programs on other IBM operating systems, see the appropriate books for corresponding APPC information:

- *OS/2 APPC Programming Guide and Reference*
- *VM/ESA CP Programming Services*
- *AS/400 APPC Programmer's Guide*
- *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*

Overview

Reference

Before you continue, you should be familiar with the basic concepts of APPC/MVS and the programming interfaces it provides. Part 1, "Introduction" of *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS* provides such information.

APPC/MVS Server Facilities

With the addition of APPC/MVS server facilities in MVS/ESA SP 4.3, MVS/ESA strengthens its support for the client/server model of programming. APPC/MVS server facilities provide installations with the high-level-language callable services needed to write APPC/MVS servers (MVS-resident programs that can manage multiple LU 6.2 inbound conversations from multiple client TPs concurrently).

As of MVS/ESA SP 4.2, MVS/ESA provides scheduling support for LU 6.2 conversations between partner transaction programs. Inbound transaction program requests are scheduled for execution by the APPC/MVS transaction scheduler. (The installation can also supply its own transaction scheduler, if desired.) The APPC/MVS transaction scheduler provides two types of scheduling for transaction programs — standard and multi-trans. In both cases, the transaction scheduler processes inbound requests by running each requested transaction program in a subordinate address space called an initiator. While multi-trans scheduling provides better performance for frequently accessed transaction programs, both scheduling types are limited; no more than one inbound conversation can run in an initiator address space at a time.

APPC/MVS server facilities provide another way to process LU 6.2 inbound transaction program requests. Address spaces (started tasks, batch initiators, TSO/E users, or APPC/MVS initiators) can directly receive specific inbound requests, instead of requiring the transaction scheduler to process them. And, unlike scheduled TPs, these address spaces can receive multiple inbound conversations concurrently.

Resource Management

Installations can use APPC/MVS servers to perform global functions on behalf of users. An installation can centralize, within a single server address space, many of the requests for resources commonly made during transaction processing. For example, servers can be used to manage access to MVS resources, such as data sets and data bases.

APPC/MVS servers thus present an ideal solution for consolidating some of the redundant processing that normally occurs when transaction programs must access the same resources on MVS.

Owning Inbound Conversations

APPC/MVS server facilities provide a set of high-level language callable services that allows a server application to "own" and thereafter manage a subset of the transaction program requests that enter an APPC/MVS system. A server owns inbound allocate requests by *registering* for them. Thereafter, APPC/MVS monitors the allocate requests that enter the system for those requests for which a server has registered. Rather than routing such requests to a transaction scheduler, APPC/MVS places the requests on a queue to await further processing by the APPC/MVS server. These queues are called **allocate queues**. APPC/MVS servers process allocate requests by receiving them from allocate queues and performing the function requested by the client TP. The services that allow servers to create and manage allocate queues are called **allocate queue services**.

APPC/MVS directs inbound allocate requests for which no server has registered to an APPC/MVS transaction scheduler.

In previous releases of MVS, applications needed to use the APPC/MVS transaction scheduler services to receive multiple inbound conversations concurrently. Now, many of the same applications can be written as APPC/MVS servers, which offer the following advantages over transaction schedulers:

- Servers are easier to code.
- Servers need not run authorized. That is, servers can run in problem state, with PSW key 8-15.
- Servers can run in any MVS address space (started tasks, batch initiators, TSO/E users, or APPC/MVS initiators) except a transaction scheduler address space.
- Servers can select a desired subset of the inbound conversations from a particular LU. (The transaction scheduler interface requires the caller to own an LU—and all of its inbound conversations—exclusively.)

The Allocate Queue

APPC/MVS servers might best be thought of as "managers" of one or more allocate queues. Therefore, the concept of the allocate queue is central to understanding how to use APPC/MVS servers.

This discussion covers the following topics:

- "Creating an Allocate Queue" on page 4
- "APPC/MVS Inbound Processing" on page 4
- "Server Processing: An Overview" on page 5
- "Request Flow" on page 6
- "Managing Allocate Queues" on page 6.

Creating an Allocate Queue

An MVS application can receive inbound LU 6.2 conversations directly (rather than through a transaction scheduler) by requesting the conversations through a process called *registering*. When it registers, the server application specifies, through the Register_For_Allocates service, which allocate requests it is to serve. Specifically, the server specifies the name of the local TP to which the client TP issues the allocate request, and the name of the LU that is targeted by the allocate request. An application that has successfully registered for allocate requests is considered to be an APPC/MVS server.

APPC/MVS places inbound allocate requests for which a server has registered on a structure called an allocate queue, on a first-in, first-out (FIFO) basis. APPC/MVS creates the allocate queue at the completion of the server's registration. APPC/MVS creates a separate allocate queue for every unique server registration. The server selects allocate requests from the allocate queue for subsequent processing.

A server can register for all the TP/LU pairs it has RACF authority to serve. Your installation's security administrator grants this authority through the APPCSERV general resource class of RACF (or a functionally equivalent security product).

APPC/MVS Inbound Processing

When a server application registers for allocate requests, it assumes a subset of the inbound workload that would otherwise be routed to a transaction scheduler. The server application is responsible for processing the allocate requests for which it has registered.

Figure 2 on page 5 illustrates inbound processing.

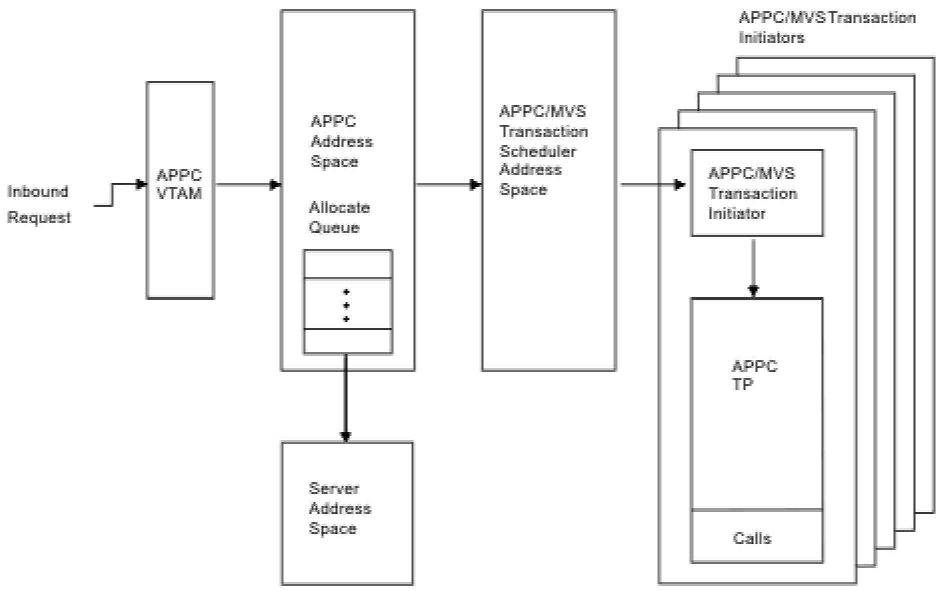
Inbound Request
APPC VTAM
APPC Address Space
Allocate Queue
Server Address Space
APPC/MVS Transaction Scheduler Address Space
APPC/MVS Transaction Initiators
APPC/MVS Transaction Initiator
APPC TP
Calls

Figure 2. APPC/MVS Inbound Processing

In Figure 2 on page 5, an inbound allocate request enters the system. APPC/MVS first checks to see whether any address spaces on the system had previously registered to serve the request. If so, APPC/MVS places the request on an allocate queue from which the server can later select it for processing. When the server selects the request from the allocate queue (through the Receive_Allocate service), the server receives the conversation ID of the conversation. A conversation with the issuer of the request (the client TP), can then take place.

In general, if no servers have registered for the request, APPC/MVS attempts to schedule the request to a transaction scheduler for subsequent processing.

For information about APPC/MVS scheduling, see *z/OS MVS Planning: APPC/MVS Management*.

Server Processing: An Overview

The following list presents an overview of the main steps that servers follow when processing inbound conversations. Later chapters of this book provide the details for performing these steps.

1. Becoming a server —

   An application becomes an APPC/MVS server when it registers to serve inbound allocate requests (through the Register_For_Allocates service). When the server's Register_For_Allocates call completes successfully, APPC/MVS begins placing inbound allocate requests that arrive for the server in an allocate queue.

   The primary parameter returned from Register_For_Allocates is the allocate queue token, which uniquely identifies the queue from which the server will select requests.
2. Serving requests —

   To obtain an allocate request from an allocate queue, a server calls the Receive_Allocate service, and supplies the allocate queue token it received at the completion of its call to the Register_For_Allocates service.

   The Receive_Allocate service returns to the server the conversation ID of the conversation just received. The server uses the conversation ID to hold APPC communications with the client TP. Both client and server can use the existing CPI-C or MVS specific conversation services described in *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*, such as Send_Data or Receive_and_Wait.
3. End processing —

   To stop serving allocate requests, a server calls the Unregister_For_Allocates service.

Request Flow

Figure 3 on page 6 shows the general flow of inbound allocate requests to a server address space. In the figure, a server has registered to receive the allocate requests that arrive at LUA for TPA. APPC/MVS places these requests on an allocate queue to await retrieval by the server. To retrieve an allocate request, the server calls the Receive_Allocate service. The Receive_Allocate service provides the server with the conversation ID, which the server uses to communicate with the client TP that issued the allocate request.

Presumably, during the conversation, the server provides a service to the client TP, such as searching data that the server controls, and returning data to the client TP. At the completion of its processing, the server might determine that it will stop serving inbound conversations. At this point, the server calls the Unregister_For_Allocates service.

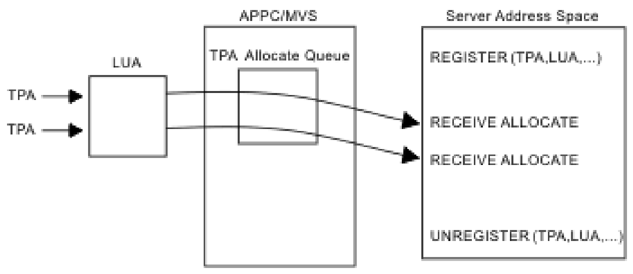

Figure 3. Flow of Inbound Conversations to a Server Address Space

While this book also describes callable services that provide more advanced functions, "basic" servers require only the services that are shown in Figure 3 on page 6. (For information about using these services, see “The Basic Server Functions” on page 10.)

Managing Allocate Queues

Servers can use APPC/MVS callable services to manage their allocate queues. For example:

- Use the Set_Allocate_Queue_Notification and Get_Event services to be notified when an allocate queue reaches a specified number of allocate requests.
- Use the Query_Allocate_Queue service to obtain commonly needed information about the status of an allocate queue, such as the number of allocate requests waiting to be processed.
- Use the Set_Allocate_Queue_Attributes service to have APPC/MVS preserve the allocate queue during interruptions in server processing.

For information about using the APPC/MVS callable services needed to perform these functions, see “Advanced Server Functions” on page 18.

Server Initialization

APPC/MVS servers can be initialized in any of the following ways:

- Operator START command (if the server runs as a started task). Here, you must define the server as a started procedure in PROCLIB. Your installation's security administrator can assign user IDs to started servers by modifying the RACF started procedures table (see “Installing APPC/MVS Servers” on page 30).
- TSO/E LOGON command (if the server is a TSO/E address space). Here, you can define the server as a command list (CLIST) or REXX exec.
- Batch initiator (if the server is submitted as a batch job). Here, you must provide the JCL needed to run the server.
- APPC/MVS initiator (if the server is scheduled by the APPC/MVS transaction scheduler in response to an inbound allocate request). For information, see the section that follows.

Starting Servers With APPC/MVS: A server can run in an address space under the control of the APPC/MVS scheduler, but, to avoid potential confusion, schedule the server by a TP name different from any TP names it serves. Having a server scheduled with a TP name, and allowing it to register for the same TP name, is acceptable if the server can guarantee that only one request for the TP name has been routed to the scheduler before its call to the Register_For_Allocates service completes successfully.

Otherwise, it is possible for the situation to arise where two or more transaction program requests of the same name are served and scheduled. (If many TP requests enter the system simultaneously, APPC/MVS attempts to schedule them. If one of the scheduled TPs registers for the same TP name, later requests for that same TP name are served).

Special Considerations for Authorized Servers

You might decide that your server should run as an authorized program. The primary advantage of authorized servers is that they can use authorized callable services and macros.

Additional reasons for having your server run authorized include:

- Using authorized RACF macros to tailor security environments for individual users.

  Multiple user requests of various security levels might run in the same server address space. These users all run under the server's security environment. Authorized servers, however, can create customized security environments to match the security access defined to individual users (through the RACROUTE REQUEST=VERIFY macro, for example).

- Joining the APPC XCF group to receive information that is otherwise available only to transaction schedulers.

  Servers must be authorized to join the APPC XCF group. As a member of this group, the server could receive APPC information, or information about an LU with which the server is associated. For example:

  - LU messages
  - "APPC ACTIVE/INACTIVE" status messages
  - Installation-determined data specified through the USERVAR and ALTLU keywords in the APPCPMxx member of the parmlib concatenation.

  Note that servers running as members of the APPC XCF group are more difficult to write. You must, for example, also provide an XCF message user routine that runs in SRB mode.

  For information about the APPC XCF group, and writing message user routines, see *z/OS MVS System Messages, Vol 3 (ASB-BPX)*.

Related System Functions

In addition to allocate queue callable services, APPC/MVS also provides the following related support for APPC/MVS servers:

- The NOSCHED option in the APPCPMxx member of the parmlib concatenation allows the installation to define LUs that are not owned by transaction schedulers (NOSCHED LUs). For APPC/MVS servers, NOSCHED LUs allow the installation to separate the workload processed by APPC/MVS servers from that done by transaction schedulers. Installations can also use NOSCHED LUs to flow outbound allocate requests without having a transaction scheduler active.
- RACF general resource class, APPCSERV, allows the installation to control which user IDs can become APPC/MVS servers.
- SMF type 33, subtype 2 records provide accounting information for each APPC conversation in an address space.
- APPC/MVS servers can specify symbolic destination names (in the SIDEINFO data set) when registering to receive inbound conversations. In previous releases of MVS/ESA, symbolic destination names could only be used for allocating outbound conversations.

- The DISPLAY APPC command provides status information (through system messages) about APPC/MVS servers and their associated allocate queues.
- CTRACE and IPCS capabilities, symptom records, and the application programming interface (API) trace facility, for diagnosing problems with APPC/MVS servers.

What the Application Programmer Provides

You are responsible for providing many of the functions your server might need, such as:

- Performing the function the client TP requested.
- Load balancing, if desired (through multi-tasking).
- Serializing access to resources, such as data sets or control blocks.
- Security checking of requestor user IDs, if desired. (The server must be authorized to do this.)
- Providing recovery from its abends.
- Starting and stopping servers.

What Your Installation Can Provide

Your installation can define one or more NOSCHED logical units (LUs) to support APPC/MVS servers. NOSCHED LUs are not "owned" by transaction schedulers and can be used to separate servers from transaction schedulers. An installation might want to isolate servers to better monitor their performance (through the DISPLAY APPC,LU command). For more information about NOSCHED LUs, see *z/OS MVS Planning: APPC/MVS Management*.

Chapter 2. Using Allocate Queue Services

Objective
This chapter provides the conceptual information you need to use APPC/MVS allocate queue callable services. For specific information about coding the parameters on these services, see Chapter 4, "APPC/MVS Allocate Queue Services," on page 37.

In the discussion that follows, the APPC/MVS allocate queue callable services are organized, based on complexity, into two groups: basic and advanced.

The basic functions allow MVS applications to serve inbound allocate requests. These functions are used to create, serve, and delete allocate queues. The allocate queue services that provide the basic functions are:

- Register_For_Allocates (ATBRFA2) — Register for inbound allocate requests.
- Receive_Allocate (ATBRAL2) — Request an inbound conversation for processing.
- Unregister_For_Allocates (ATBURA2) — Unregister for inbound allocate requests.

These services are described in "The Basic Server Functions" on page 10.

APPC/MVS servers use the advanced functions to manage the allocate queues for which the servers are registered. Servers could, for example, use advanced functions to aid in performing load balancing. The allocate queue services that provide the advanced functions are:

- Set_Allocate_Queue_Notification (ATBSAN2) — Request to be notified when an allocate queue reaches a specified maximum or minimum number of allocate requests (or cancel a previous request for such notification).
- Get_Event (ATBGTE2) — Obtain notification of events that was previously requested through the Set_Allocate_Queue_Notification service.
- Query_Allocate_Queue (ATBQAQ2) — Obtain commonly needed information about the status of an allocate queue, such as the number of allocate requests that currently reside on the queue.
- Set_Allocate_Queue_Attributes (ATBSAQ2) — Have APPC/MVS preserve a particular allocate queue during periods of time when no servers are registered for the queue.

These services are described in "Advanced Server Functions" on page 18.

Coded Example
This book concludes with a coded example of an APPC client/server application (written in the C programming language), in three parts: • Appendix C, "Sample APPC/MVS Server," on page 87 • Appendix D, "Sample Client Program," on page 97 • Appendix E, "Sample Error Routine and Header File," on page 103.

Using the Services Asynchronously

You can use allocate queue services *asynchronously* by specifying the address of an event control block (ECB) on the notify_type parameter. If APPC/MVS accepts the request for asynchronous processing, it sets a return code of zero for the service. The server can then continue other processing before waiting for the ECB to be posted. When all parameters are returned and the service completes, APPC/MVS notifies the server by posting the ECB.

By contrast, when you call a service *synchronously* (by setting the notify_type parameter to a value of *none*), your server loses control until the service completes. Your server regains control when APPC/MVS completes the request, and passes return and reason codes, and any other returned parameters from the service.

When you specify an ECB on the notify_type parameter, you must clear the ECB to zero and meet all requirements for the POST macro. When the system posts the ECB to show that asynchronous processing is complete, the completion code in the ECB is the return code for the service. The system processes all input parameters before returning to the server, so the server is free to use these areas on return without affecting the asynchronous processing. The system accesses and manipulates all output parameters directly by the asynchronous processing, however, and therefore the server should not reference or modify these parameters until the system notifies the server that call processing is complete.

Asynchronous processing is recommended for performance-intensive server applications, such as:

- Servers that perform multi-processing
- Servers that cannot be suspended
- Servers that process multiple conversations in a single dispatchable unit.

Asynchronous processing is available on many TP conversation services. For more information, see *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*.

Using JES Services

APPC/MVS servers can use the same JES SYSOUT and data set integrity checking services available to other MVS applications. Note that any SYSOUT data generated by a server is printed under the server's jobname and user ID (not the user IDs of the server's individual client TPs).

Generally, all JES services used by the server run under the server's user ID.

The Basic Server Functions

This section explains how applications can use allocate queue services to create, serve, and delete allocate queues. For additional reference information for each of the allocate queue services, including syntax and parameter descriptions, see Chapter 4, “APPC/MVS Allocate Queue Services,” on page 37.

Creating An APPC/MVS Server

An application becomes a server of inbound allocate requests through a process called *registering*. When it registers, the application specifies, through the Register_For_Allocates service, the name of the local TP to which the client TP issues its allocate requests, and the name of the LU that is targeted by the allocate requests. An application that has successfully registered for allocate requests is considered to be an APPC/MVS server.

Normally, APPC/MVS directs inbound allocate requests to a transaction scheduler. When an application registers for allocate requests, it essentially "steals" a subset of the inbound workload that would normally be routed to a transaction scheduler. The server application then becomes responsible for processing the allocate requests for which it has registered. (Note that servers do not share LUs with schedulers, unless the servers are owned by the schedulers. For more information, see “For Which Local LUs Can a Server Register?” on page 12.)

Specifying Symbolic Destination Names

On its call to the Register_For_Allocates service, the server application specifies the TP name and local LU name of the inbound allocate requests to be served. The server can specify explicit values for the TP_name and local_LU_name parameters, or it can specify a *symbolic destination name* which resolves to the TP name/local LU name. (Installations define symbolic destination names in the side information data set.) If you specify a symbolic destination name, the local LU name and TP name are obtained from the side information data set (the mode name, which is also contained in the side information, is ignored for calls to the Register_For_Allocates service). The local LU name is taken from the partner LU specified in the symbolic destination entry.

For information about defining symbolic destination names, see *z/OS MVS Planning: APPC/MVS Management*.

Filtering Requests

When registering for inbound allocate requests, applications can limit their selection of requests by specifying certain "filter" parameters. Applications can limit their registration to the allocate requests for particular TP/local LU pairs *that bear a certain user ID, profile, or partner LU name* — or some combination of these filters.

APPC/MVS creates a separate allocate queue for each unique combination of filters you specify to the Register_For_Allocates service. When multiple servers register for the same TP name/LU name, and specify the same filter values, the servers share the allocate queue that results. If servers register for the same TP name/LU name, but specify different filter values, APPC/MVS creates a different allocate queue for each unique registration.

Specify a blank value for any filter group you do not want to limit. If you specify blanks for all filters, your server can receive allocate requests for the specified TP name and local LU name, regardless of user ID, profile, or partner LU name.

It is possible for one server to register for a subset of the inbound requests that are being served by another server. In this situation, APPC/MVS uses a hierarchy to determine which server is to receive the inbound request.

For example, Server A registers for a particular TP name, and specifies blanks for the filters, to receive all requests for the TP regardless of the client TP's user ID, profile, or partner LU name. Server B registers for the same TP name, but specifies PAYROLL for the profile filter (and blanks for User ID and partner LU name), to receive only the client TP's requests from the PAYROLL security profile.

Table 1 on page 11 shows the order in which APPC/MVS checks for a server for allocate requests:

Table 1. Search Order for Filters

Order of Priority	**Request is queued for the server that specified the following filters:**
1	User ID, profile, and partner LU name
2	User ID and profile
3	User ID and partner LU name
4	User ID
5	Profile and partner LU name
6	Profile
7	Partner LU name
8	No filters were specified (accept any allocate request from the client TP, regardless of filter values).

Applying this hierarchy to our example shows that APPC/MVS places the client TP's allocate requests from security profile PAYROLL on Server B's allocate queue, and places all other requests from the client TP on Server A's allocate queue.

Receiving the Allocate Queue Token

The primary output from a successful Register_For_Allocates call is the *allocate queue token*, which uniquely identifies the allocate queue. Use the allocate queue token to indicate the particular allocate queue from which your server is to receive allocate requests (through the Receive_Allocate service). A server also requires the allocate queue token to perform other types of processing provided by allocate queue services.

Servers that share an allocate queue also share the same allocate queue token. As mentioned previously, multiple servers share an allocate queue by registering for the same TP name/local LU name and filters, if any.

Securing Access to Client TPs

RACF allows you to define APPCSERV RACF security profiles to control servers' access to client TPs. APPCSERV profile names allow the installation to grant or deny servers access to specific TP or LU names.

APPCSERV profile names are of the form dbtoken.tpname, where:

- *dbtoken* is the database token (1 to 8 characters) of the TP profile data set. The TP profile dataset is associated with the LU at which the server resides. (This is the LU that your server specifies on the local_LU_name parameter of the Register_For_Allocates service.)
- *tpname* is the name of the transaction program to be served.

To register for a particular TP name, the user ID under which the server runs must have been granted READ access to the TP's security profile in the APPCSERV RACF general resource class.

RACF uses the user ID from the server's task-level security environment, or, if the task-level security environment does not exist, RACF uses the user ID associated with the server's address space security environment.

If the TP name is not protected by the APPCSERV class, and the APPCSERV class is active, APPC/MVS fails the request with return code 16 (atbcts_request_unsuccessful) and reason code 14 (atbcts_not_auth_to_serve_tp).

For the procedure your installation's security administrator could use to set-up security for TPs and servers, see “Installing APPC/MVS Servers” on page 30.

For Which Local LUs Can a Server Register?

In some situations, a server cannot specify certain LUs as the local LU on its call to the Register_For_Allocates service. Use Table 2 on page 13 to determine whether your server can specify a particular local LU.

In Table 2 on page 13, a "yes" or "no" indicates whether a server in the address space in the left column (Column 1) is permitted to specify a particular local LU. If an address space attempts to register for an LU it cannot use, APPC/MVS fails the request with return code 16 (atbcts_request_unsuccessful) and reason code 15 (atbcts_not_auth_to_local_lu).

Column 1 lists different types of address spaces that contain servers attempting to register. The top two entries in column 1 show address spaces that are not "owned by" (connected to) a transaction scheduler. Typical examples of such address spaces are TSO/E address spaces and JES batch initiators. The last four rows of Table 2 on page 13 show address spaces that are owned by transaction schedulers.

Address spaces can also differ depending on whether their conversations can flow through the system base LU (the default LU). Normally, an address space's conversations can use the system base LU. If an address space is to be later connected to a transaction scheduler, however, the scheduler might need to limit the address space's conversations to LUs owned by the scheduler. Here, the scheduler would prohibit the address space from using the system base LU through the Set_AS_Attributes service.

For information about the system base LU, see *z/OS MVS Planning: APPC/MVS Management*. For information about writing transaction schedulers and using the Set_AS_Attributes service, see *z/OS MVS System Messages, Vol 3 (ASB-BPX)*.

The columns to the right of Column 1 in Table 2 on page 13 show the results when servers attempt to register for different types of local LUs. Column 2 shows the results when servers specify NOSCHED LUs when registering. (A NOSCHED LU is an LU that is not associated with a transaction scheduler. Like other LUs, NOSCHED LUs are defined through the LUADD statement in the APPCPMxx member of the parmlib concatenation. For more information about defining NOSCHED LUs, see *z/OS MVS Initialization and Tuning Reference*.)

Columns 3 and 4 of Table 2 on page 13 show the results when servers attempt to register for LUs that are owned by different transaction schedulers ("scheduler A" and "scheduler B") that coexist on the same system. The IBM-supplied APPC/MVS transaction scheduler, ASCH, is an example of such a scheduler.

Table 2. Local LUs for Which a Server Can Register

The server address space is...	**The LU is...**		
	...a NOSCHED LU	***...owned by Scheduler A***	***...owned by Scheduler B***
...not owned by a scheduler	Yes	No	No
...not owned by a scheduler and "prohibit default LU" is specified	No	No	No
...owned by Scheduler A	Yes	Yes	No
...owned by Scheduler A, with "prohibit default LU" specified	No	Yes	No
...owned by Scheduler B	Yes	No	Yes
...owned by Scheduler B, with "prohibit default LU" specified	No	No	Yes

Additional Considerations for Registering

When using the Register_For_Allocates service, you will need to observe the following conventions and restrictions:

- An address space that calls the Register_For_Allocates service for a TP/local LU takes precedence over a program that specifies the same TP/local LU on a call to the Register_Test service. (The Register_Test service is described in *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*.)
- A transaction scheduler (one that has already called the APPC Identify service) cannot register for allocate requests. If a transaction scheduler calls the Register_For_Allocates service, APPC/MVS fails the request with return code 16 (atbcts_request_unsuccessful) and reason code 10 (atbcts_sched_cant_register).
- An APPC/MVS server cannot use a VTAM generic resource name for a local LU, either explicitly on the Register_For_Allocates service or implicitly through a side information entry. In these cases, the Register_For_Allocates service returns control with return code 16 (atbcts_request_unsuccessful), and reason code 12 (atbcts_inval_local_lu).
- An APPC/MVS server may filter allocate requests by specifying a VTAM generic resource name for the partner LU. If it does so, but the local LU first establishes a conversation with the partner using the partner's specific name, the server will not receive any allocate requests even if APPC/MVS receives requests that exactly match the filters. To avoid this problem, register the server with an APPC/MVS LU that handles outbound conversations using only the generic resource name for the partner LU.

Receiving Inbound Conversations

An allocate queue stores allocate requests in the order which they arrive; that is, in first-in, first-out (FIFO) order. To request an inbound conversation, an APPC/MVS server calls the Receive_Allocate service, which returns the oldest allocate request (inbound conversation) on the specified allocate queue. Specifically, Receive_Allocate returns the conversation ID of the inbound conversation, which allows your server to communicate with the client TP. (Note that APPC/MVS servers use the Receive_Allocate service to receive inbound conversations, instead of the Accept_Conversation or Get_Conversation calls, which are commonly used by scheduled transaction programs.)

Specifying the Allocate Queue Token

To indicate the particular allocate queue from which your server is to receive an allocate request, specify the allocate queue token the server received when it registered. Specify the allocate queue token in the allocate_queue_token parameter of the Receive_Allocate service.

Allowing the Request to Wait

At times, an allocate queue might not contain an allocate request when your server requests one. To handle this possibility, you can specify that the Receive_Allocate service is allowed to wait for an allocate request to become available (through the receive_allocate_type parameter).

To specify whether the Receive_Allocate request is allowed to wait, and, if so, for how long, set the receive_allocate_type parameter to one of the following values:

atbcts_wait
Wait indefinitely for an allocate request to become available.

atbcts_timed
Wait for a specified interval before returning control (this interval is specified, in seconds, in the time_out_value parameter).

atbcts_immediate
Do not wait (return control immediately) if an allocate request is not currently available.

Specifying the Timeout Value

You can specify a *timeout value* for your calls to the Receive_Allocate service. The timeout value controls the amount of time the Receive_Allocate request can wait for an allocate request to become available. If an allocate request becomes available before the timeout value is exceeded, the server receives the allocate request. Otherwise, APPC/MVS returns control to the server with return code 16 (atbcts_request_unsuccessful) and reason code 21 (atbcts_no_alloc_to_receive).

Have the server specify a timeout value large enough to allow for intervals when there are no allocate requests on the allocate queue, and the server must wait for a new request to arrive.

Used With Notify_Type

To further define how APPC/MVS is to process your request, set the notify_type parameter to indicate whether the request is to be processed synchronously or asynchronously. Table 3 on page 14 shows the effects of specifying the receive_allocate_type parameter with the notify_type parameter:

Table 3. Results of Notify_Type/Receive_Allocate_Type Combinations

Notify_Type Value:	**Receive_Allocate_Type Value:**		
	atbcts_immediate	***atbcts_wait***	***atbcts_timed***
NONE	Control returns to the caller immediately with either an allocate request (if one is available), or a return and reason code that indicates the request failed because no allocate requests are available to be received.	Control does not return to the caller until an allocate request is received.	Control does not return to the caller until an allocate request is received, or the interval specified in the time_out_value parameter is exceeded.

Table 3. Results of Notify_Type/Receive_Allocate_Type Combinations (continued)

Notify_Type Value:	**Receive_Allocate_Type Value:**		
	atbcts_immediate	***atbcts_wait***	***atbcts_timed***
ECB	Control returns to the caller immediately, while APPC/MVS processes the call asynchronously. If an allocate request is available, Receive_Allocate returns the allocate request. Otherwise, the caller receives a return and reason code which indicates that no allocate requests are available to be received.	Control returns to the caller immediately, while APPC/MVS processes the call asynchronously. APPC/MVS posts the caller's ECB when an allocate request is returned.	Control returns to the caller immediately, while APPC/MVS processes the call asynchronously. If an allocate request is available, Receive_Allocate returns the allocate request. Otherwise, Receive_Allocate waits for the interval specified in the time_out_value parameter for an allocate request before posting the ECB.

For both synchronous and asynchronous calls to the Receive_Allocate service, the server is notified when the allocate request arrives, or when the specified timeout value is exceeded.

Receiving the Conversation ID

The primary output from a successful Receive_Allocate call is the conversation ID, which uniquely identifies the conversation that was received. A server uses the conversation ID to communicate with the transaction program that initiated the conversation (the client TP). The conversation ID is required as input to other APPC/MVS callable services, such as the LU 6.2 Send and the CPI-C Receive services. Once a server obtains the conversation ID, communications can proceed between the partner programs (client TP and server) until one partner ends the conversation.

For information about using APPC/MVS TP conversation services, see *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*.

Other Outputs From Receive_Allocate

Besides the conversation ID, the Receive_Allocate service returns the following information about the conversation that was received:

- Conversation mode name and partner LU name
- Conversation type and synchronization level
- User ID and security profile.

Conversation Mode Name/Partner LU Name

The Receive_Allocate service returns the conversation mode name and partner LU name, which define the session in which the conversation is running.

Conversation Type/Synchronization Level

The server can use the returned parameters, conversation type and synchronization level, to determine whether particular conversations conform to expected formats.

User ID/Security Profile

The Receive_Allocate service returns the user ID and security profile associated with the inbound conversation that was received. The server can use this information to perform security-related functions.

If your server is to run authorized (that is, in supervisor state, or with PSW key 0-7), it can use RACF macros to check the security of client user IDs, or create a security environment that is appropriate for the client, if desired. For example:

- Use the RACROUTE macro to examine the security environment of requestors' user IDs, or call the Get_Attributes service to obtain the requestor's user token (UTOKEN).
- Set up separate security environments for users with various security clearances (using the RACROUTE REQUEST=VERIFY macro).

For more information about using the RACROUTE macro, see *z/OS Security Server RACROUTE Macro Reference*.

If your server is to run unauthorized (that is, in problem state, or with PSW key 8-15), your installation can use the APPCTP security class to limit the user IDs that can run certain TPs.

Using the Get_Attributes Service

Your server can extract information about the conversations it is processing, by calling the Get_Attributes service. This service returns information about a particular conversation, such the following:

- Partner LU
- Mode name
- Synchronization level
- Data format (mapped or basic)
- RACF UTOKEN associated with the inbound request.

The server could use this information, for example, to run under different combinations of the above, using conditional logic to function appropriately.

For information about using the Get_Attributes service, see *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*.

Rejecting Conversations

If necessary, your server can reject particular conversations by calling the Reject_Conversation service. A server might reject conversations for reasons such as:

- After issuing Get_Attributes, the server determines that the inbound conversation has attributes the server does not support, such as a different synchronization level or data format.
- The server determines that the output values from the Receive_Allocate service do not match values that the server is designed to expect.
- When the server cannot satisfy inbound requests quickly enough to meet the installation's performance goals. For example, assume a database the server needs to access has become unavailable. In this situation, you might decide that it is better to require users to retry rather than wait for an unacceptably long period.

When rejecting a conversation, your server communicates the reason for the rejection to the client TP by specifying a sense code (as an input parameter to the Reject_Conversation service). The partner LU resolves the sense code to a return code that it passes to the client TP.

The Reject_Conversation service must be used early in a transaction; that is, before any communication activity has occurred (such as data being sent or received). Otherwise, APPC/MVS fails the call to the Reject_Conversation service. A limited set of services, however, can be used on a conversation without causing the conversation to be rendered ineligible for rejection. These services allow you to obtain the conversation ID, as well as conversation-related information that might help you in making rejection decisions.

Specifically, you can call the following services on a conversation and still retain the ability to reject it:

- Receive_Allocate
- Get_Conversation
- Accept_Conversation
- Get_Type
- Get_Attributes

- Extract_Conversation_Type
- Extract_Mode_Name
- Extract_Sync_Level
- Extract_Partner_LU_Name.

The Reject_Conversation service is available to all TPs (including APPC/MVS servers). For more information about using the Reject_Conversation service, see *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*.

Unregistering For Allocate Requests

To have your server stop serving one of its allocate queues, or stop serving *all* the allocate queues for which it is registered (to allow for a temporary pause in service, for example), call the Unregister_For_Allocates service.

Through the Unregister_For_Allocates service, specify either of the following:

- Stop serving a particular allocate queue. Specify as an input the allocate queue token for that allocate queue.
- Stop serving all allocate queues for which the server is registered (in other words, stop being an APPC/MVS server). Here, specify eight bytes of zeroes as input, in place of an allocate queue token.

On successful completion of the Unregister_For_Allocates service, the server is no longer registered for one — or all — of its allocate queues. The server cannot serve an allocate queue for which it is no longer registered.

Unregistering from an allocate queue also causes the server's outstanding calls to Receive_Allocate for that allocate queue to fail with return code 16 (atbcts_request_unsuccessful) and reason code 20 (atbcts_request_cancelled).

To resume serving the same allocate queue, the server must re-register for the queue, and specify the same parameter values it specified on its original call to the Register_For_Allocates service. (The server receives an allocate queue token, which it uses to resume serving the allocate queue.)

Normally, APPC/MVS purges an allocate queue after the last server of the queue unregisters from it. APPC/MVS rejects the allocate requests that reside on the allocate queue with sense code X'084C0000' (TP_Not_Available_No_Retry).

You can, however, have APPC/MVS preserve an allocate queue when no servers are available. Here, APPC/MVS maintains the queue for an interval you specify through the Set_Allocate_Queue_Attributes service. During the interval, APPC/MVS continues to place new inbound requests for the server on the allocate queue. If no server re-registers for the allocate queue before the interval expires, APPC/MVS purges the queue and attempts to schedule later inbound requests for the TP/LU name. APPC/MVS rejects any inbound requests that cannot be scheduled. For more information, see "Allowing the Allocate Queue to Persist" on page 26.

If you do not specify that an allocate queue is to be maintained when no servers are registered for it, the following occurs on successful completion of the Unregister_For_Allocates service:

- If there are other servers registered for this allocate queue, the allocate queue remains active.
- If no other servers are registered for the allocate queue, APPC/MVS purges the allocate requests on the allocate queue.

Note that when you call the Unregister_For_Allocates service, you cause APPC/MVS to clean-up any event notification options you might have requested for the allocate queue. For more information about event notification options, and the effects of calling Unregister_For_Allocates, see "Receiving Notification of Events" on page 19.

APPC/MVS automatically unregisters a server from its allocate queues for the following situations:

- Server's address space ends
- Job that started the server ends

- Cleanup_AS or Cleanup_TP service is issued for the server's address space
- Server's local LU is deleted.

If the server is running in a transaction scheduler's subordinate address space, APPC/MVS unregisters the server from its allocate queues when the owning scheduler calls the Unidentify service, or implicitly causes the Unidentify service to be called. In this situation, all conversations that were previously received through calls to the Receive_Allocate service are unaffected by this clean-up processing and remain active.

Using Multiple Servers

To handle high volumes of inbound requests, you can write servers that perform multi-tasking (see "Multi-Tasking Servers" on page 27), or, if transaction program activity exceeds the capacity of a single server address space, you can start additional server address spaces to serve the same allocate queue. The initial server can use callable services to determine the transaction program load, and then start a new "copy" of itself if appropriate.

Figure 4 on page 18 shows two server address spaces serving the same allocate queue:

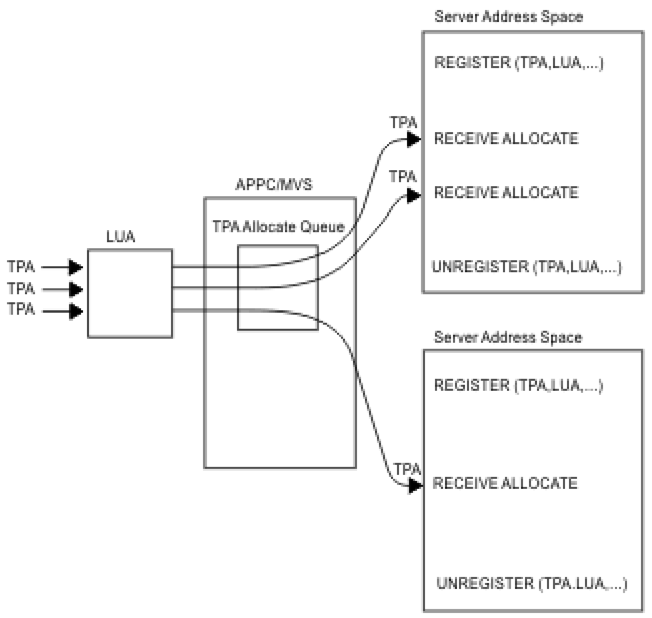

Figure 4. Flow of Inbound Conversations to Multiple Server Address Spaces

Note that APPC/MVS does not attempt to balance the transaction program requests among address spaces.

Server address spaces must be started before they can serve any of the transaction programs they are to serve (that is, there is no demand scheduling of server address spaces).

Advanced Server Functions

Besides the basic server functions described in the previous section, APPC/MVS servers can perform additional functions, such as:

- Obtain commonly needed information about the status of an allocate queue, such as the number of allocate requests that currently reside on the queue.
- Receive notification when a particular allocate queue reaches a certain number of allocate requests.
- Have the system preserve the allocate queue during interruptions in server processing.
- Write client-specific information to SMF accounting records.

This section describes how to use APPC/MVS allocate queue services to perform these advanced server functions. For additional reference information for each of the allocate queue services, including syntax and parameter descriptions, see Chapter 4, “APPC/MVS Allocate Queue Services,” on page 37.

Querying the Allocate Queue

An APPC/MVS server can call the Query_Allocate_Queue service to obtain commonly needed information about an allocate queue it is serving. The Query_Allocate_Queue service returns the following information about the specified allocate queue:

- TP name/local LU name for which the server registered
- Number of allocate requests in the allocate queue
- Age of the oldest allocate request on the allocate queue
- Time when the last call to the Receive_Allocate service was made by the server
- Time when the Receive_Allocate service last returned control to the server.

A server can query any allocate queue for which it is currently registered. The server supplies, as one input to this service, the allocate queue token it received when it registered for the allocate queue.

TP Name/Local LU name

Among the outputs from the Query_Allocate_Queue service are the name of the TP and LU associated with the allocate queue.

The server specified these names when it registered for allocate requests. If the server is serving more than one allocate queue, it can use the data returned in the TP_name and local_LU_name parameters to distinguish one allocate queue from another.

Other Information About the Allocate Queue

The Query_Allocate_Queue service also returns information that a server can use to determine whether it is processing the requests on an allocate queue quickly enough to meet the installation's throughput goals. Among the information returned is the number of allocate requests that currently reside on the allocate queue (in the allocate_queue_size parameter).

The Query_Allocate_Queue service also returns the following information:

- Amount of time (in seconds) that the oldest allocate request has been in the allocate queue
- Time when the last call to the Receive_Allocate service was made by the server
- Time when the Receive_Allocate service last returned control to the server.

A server can examine these parameters to determine whether it is meeting the installation's throughput goals, or if the allocate queue is growing at a rate that requires more resources (such as additional servers) to process. If desired, your server can take actions to increase its throughput. For example, if the server does multi-tasking, it might add subtasks to process allocate requests at a faster rate.

A server can use the Query_Allocate_Queue service with the Get_Event service to determine the status of the allocate queue for load balancing or diagnostic purposes. The Get_Event service is described in “Receiving Notification of Events” on page 19.

Receiving Notification of Events

An APPC/MVS server can base its processing on the current sizes of its allocate queues. For example, your server might allocate additional resources to manage a growing workload more efficiently. This type of processing is sometimes called workload balancing.

To check the size of an allocate queue at any one point in time, your server can call the Query_Allocate_Queue service. To monitor the growth of an allocate queue *dynamically* (that is, on an ongoing basis), however, use the Set_Allocate_Queue_Notification service with the Get_Event service, as follows:

- Call the Set_Allocate_Queue_Notification service to request notification whenever an allocate queue reaches a particular number of allocate requests.
- Call the Get_Event service to actually receive the notification you requested through the Set_Allocate_Queue_Notification service.

Requesting Notification of Events

An APPC/MVS server can request to be notified when an allocate queue it is serving reaches a particular number of inbound allocate requests. The server requests such notification by calling the Set_Allocate_Queue_Notification service.

Use the Set_Allocate_Queue_Notification service to be notified when a particular allocate queue:

- Increases to a certain number of allocate requests or
- Decreases to a certain number of allocate requests.

Notification of such events might be useful to servers that perform load balancing. For example, if your server uses a variable number of subtasks to process an allocate queue, the server can request notification when its allocate queue increases to a predetermined maximum number of allocate requests. In this event, the server might then take some action in response to the growing workload, such as increasing the number of subtasks it uses to process the allocate queue.

By calling the Set_Allocate_Queue_Notification service, a server uses APPC/MVS to monitor the size of a particular allocate queue. This service allows the server to specify a number (or *threshold*) of allocate requests on which notification is to occur. A server can specify a minimum or maximum threshold, or both.

When the allocate queue reaches the specified threshold, APPC/MVS "informs" the server by placing an element that represents the event on a structure known as the server's **event queue**. There is one event queue per server, and it can contain any number of event elements for any of the allocate queues for which the server is registered. Each event element contains the type of threshold that was reached ('MIN' or 'MAX'), as well as the allocate queue token that identifies the particular allocate queue, and a timestamp that shows when the event occurred.

For example, assume a server has requested to be notified when its allocate queue increases to 40 allocate requests, or decreases to 15. Figure 5 on page 20 shows how APPC/MVS creates event elements in response to the allocate queue reaching either of these thresholds.

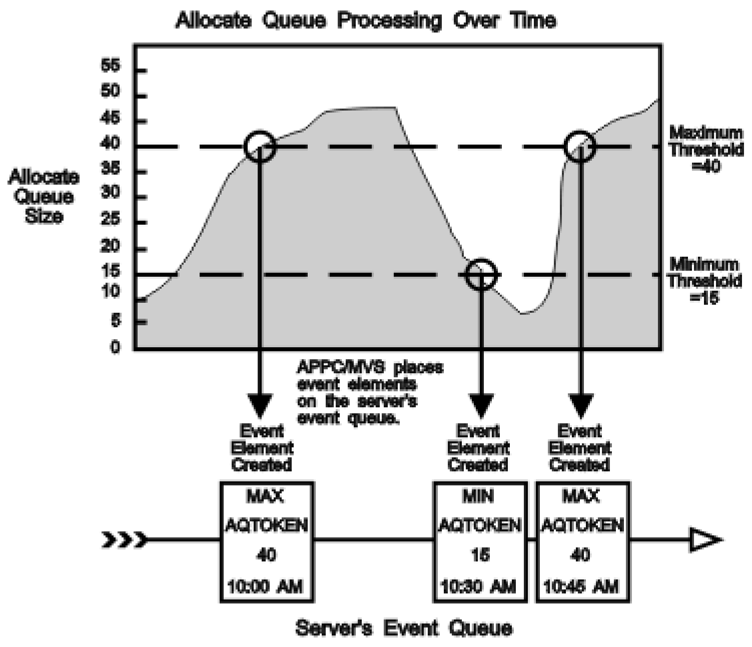

Figure 5. Relationship Between Allocate Queue and Event Queue

APPC/MVS places elements on the server's event queue in the order which they occur — that is, on a first-in, first-out, (FIFO) basis. To retrieve an event element, a server calls the Get_Event service, which returns the oldest event on the server's event queue.

Your server can request notification for any allocate queue for which it is currently registered. The server supplies, as one input to this service, the allocate queue token it received when registering for the allocate queue (through the Register_For_Allocates service). The allocate queue token uniquely identifies a specific allocate queue.

Your server can also cancel its previous requests for notification (see "Cancelling Event Notification" on page 24).

Using Get_Event to Retrieve Event Elements

Use the Get_Event service in conjunction with the Set_Allocate_Queue_Notification service. To determine whether a requested event has occurred, and, if so, the type of event, have your server attempt to retrieve an event element from its event queue by calling the Get_Event service.

Specifying Event Notification Types

Besides specifying minimum or maximum thresholds (or both), you can further define your notification requests by specifying whether notification is to be *one-time* or *continuous* (through the event_notification_type parameter). One-time notification causes APPC/MVS to monitor for a single occurrence of the specified event. After the event occurs, APPC/MVS notifies the server (through the server's event queue) and stops monitoring for the event. Continuous notification causes APPC/MVS to notify the server every time the specified event occurs, and remains in effect until one of the following happens:

- The server cancels the notification request
- The server unregisters from the allocate queue (causing APPC/MVS to cancel the notification request)
- The APPC/MVS address space ends.

Specifying Thresholds

Use the event_code and event_qualifier parameters of the Set_Allocate_Queue_Notification service to specify a minimum or maximum threshold for a particular allocate queue.

- Use the event_code parameter to indicate whether the threshold is a minimum or maximum threshold.
- Use the event_qualifier parameter to define a specific numeric value for the threshold.

When you specify a maximum threshold, APPC/MVS notifies your server if the number of allocate requests on the allocate queue increases to the threshold. For example, if the server sets a maximum threshold of 50 allocate requests, APPC/MVS notifies the server when the number of allocate requests on the allocate queue increases to 50.

For maximum thresholds, the allocate queue must *increase* to trigger notification. For example, assume an allocate queue exceeds a maximum threshold of 50 allocate requests, and then decreases below it (because the server has processed some of the allocate requests). APPC/MVS notifies the server when the allocate queue grows above 50 allocate requests, but does not notify the server a second time when the allocate queue later decreases below the maximum threshold. If the server requested continuous notification for this event, APPC/MVS does not notify the server again until the number of allocate requests on the allocate queue decreases to less than 50 and then increases to 50 again (as shown in Figure 6 on page 22).

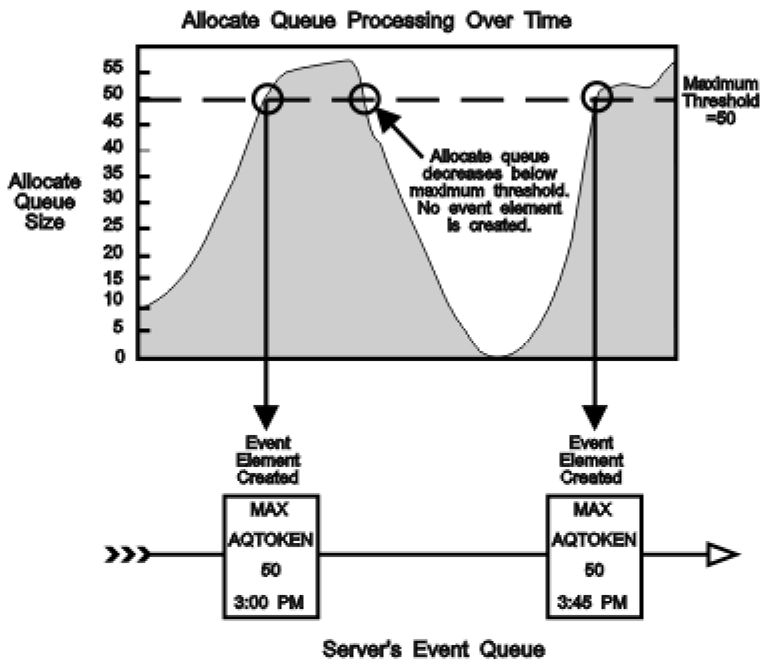

Figure 6. Maximum Threshold Reached

When you specify a minimum threshold, APPC/MVS notifies your server if the number of allocate requests on the allocate queue decreases to the threshold. For example, if the server sets a minimum threshold of 1 allocate request, APPC/MVS notifies the server when the number of allocate requests on the allocate queue decreases to 1.

For minimum thresholds, the allocate queue must *decrease* to trigger notification. For example, assume a server sets a minimum threshold of 1 allocate request for an allocate queue. APPC/MVS notifies the server when the number of allocate requests on the allocate queue decreases to 1. APPC/MVS does not notify the server if the allocate queue increases from 0 to 1, but rather when it decreases from 2 to 1. If the same server requested continuous notification for this event, APPC/MVS would not notify the server again until the allocate queue increases to more than 1 allocate request and then decreases to 1 (as shown in Figure 7 on page 23).

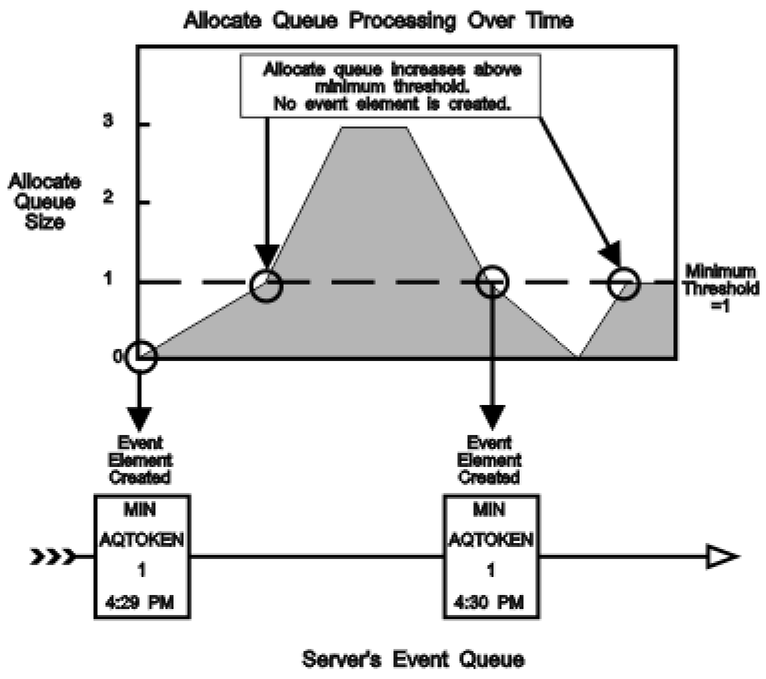

Figure 7. Minimum Threshold Reached

Threshold Already Reached

If a specified threshold is already reached when your server calls the Set_Allocate_Queue_Notification service, APPC/MVS immediately places an element on the server's event queue.

For maximum thresholds, APPC/MVS places an element on the server's event queue if the allocate queue is currently equal to, or greater than, the specified threshold. For example, if the server requests to be notified when an allocate queue reaches 50 allocate requests, and the allocate queue already contains 60 allocate requests, APPC/MVS immediately places an event element on the server's event queue.

For minimum thresholds, APPC/MVS considers the threshold to have been already reached if the allocate queue contains less than or equal to the specified threshold. Figure 7 on page 23 shows that if the server specifies a minimum threshold of 1, and the allocate queue contains no allocate requests, APPC/MVS immediately places an event element on the server's event queue.

Note that APPC/MVS's handling of the "minimum threshold already reached" situation allows for the possibility that APPC/MVS will queue an event element without the allocate queue having actually *decreased* to the specified minimum threshold. To determine whether a threshold has already been reached, the server can call the Get_Event service immediately after calling the Set_Allocate_Queue_Notification service.

Specifying Multiple Thresholds

A server can specify both a minimum and maximum threshold for a particular allocate queue, but only one per call to the Set_Allocate_Queue_Notification service. Therefore, the server must make a separate call for each threshold it sets.

Also, a server can have one or more of the following combinations of threshold types (minimum or maximum) and notification types (continuous or one-time) active at the same time:

- Minimum/one-time
- Minimum/continuous
- Maximum/one-time
- Maximum/continuous.

A server cannot have more than one of each of the preceding combinations active at the same time. For example, the server cannot have two maximum/continuous requests active. The latest request overrides the earlier request (as explained in “Modifying Thresholds” on page 24, which follows).

Modifying Thresholds

You can override a threshold that you set through a previous notification request by re-issuing the Set_Allocate_Queue_Notification service and specifying a new value for the event_qualifier parameter (the event notification type must be the same; otherwise, APPC/MVS monitors for both events). The new threshold value overrides the value set on the previous call to the Set_Allocate_Queue_Notification service.

Example of Modifying a Threshold

To override a previous threshold, you must specify the same event notification type that you specified on the request to be overridden. For example, assume that a server requested to be notified when an allocate queue increases to 25 allocate requests and specified one-time notification. The server then calls the Set_Allocate_Queue_Notification service with the same parameters, with the exception that the server changes the maximum threshold to 30. The second call overrides the first call.

Assume the same server again calls the Set_Allocate_Queue_Notification service. This time, the server specifies a maximum threshold of 40 allocate requests, but specifies continuous notification. APPC/MVS considers this call to be different from the previous notification requests. As a result, APPC/MVS monitors the server's allocate queue for the following events:

- A maximum threshold of 30, for which the server is to receive one-time notification
- A maximum threshold of 40, for which the server is to receive continuous notification.

Cancelling Event Notification

You can cancel active notification requests through the Set_Allocate_Queue_Notification service. You can cancel either a particular request, or, through a single call to the Set_Allocate_Queue_Notification service, cancel all of your server's active notification requests for an allocate queue.

To cancel a particular notification request, call the Set_Allocate_Queue_Notification service with parameters that match the request to be cancelled, with the exception that the event_notification_type parameter is set to a value of atbcts_cancel_notify. APPC/MVS stops monitoring for the particular event, and deletes from the server's event queue any elements that were queued because of the original notification request.

To cancel all of its active notification requests for an allocate queue, call the Set_Allocate_Queue_Notification service with:

- The event_notification_type parameter set to a value of atbcts_cancel_all_notify and
- The allocate_queue_token parameter containing the allocate queue token received from the Register_For_Allocates service.

APPC/MVS checks, but does not use, the other parameters in this case. APPC/MVS stops monitoring for the allocate queue and deletes the event elements related to this allocate queue from the server's event queue.

Note that cancelling notification requests might cause APPC/MVS to cancel outstanding calls to the Get_Event service. In the "cancel all" situation, APPC/MVS cancels the server's outstanding call to the Get_Event service when APPC/MVS is not monitoring any other allocate queues for the server. In the "cancel a particular request" situation, APPC/MVS cancels an outstanding call to the Get_Event service when the server has no other outstanding requests for notification.

Event Notification Clean-Up

When a server unregisters from an allocate queue, APPC/MVS cancels all the server's active notification requests for the allocate queue.

Unregistering from the allocate queue also causes APPC/MVS to purge the server's event queue of any elements related to this allocate queue. Also, if the server has an outstanding call to the Get_Event service, APPC/MVS cancels the call.

If the event notification requests being cancelled are the only ones the server has requested, and the server has an outstanding Get_Event request, the Get_Event service will return to the server with return code 16 (atbcts_request_unsuccessful) and reason code 20 (atbcts_request_cancelled).

Retrieving Event Elements

To retrieve the next event element on an event queue, call the Get_Event service. You can have only one outstanding call to the Get_Event service at a time. On the call to Get_Event, specify the event_get_type and notify_type parameters to indicate how APPC/MVS is to process the call, as follows:

- Specify event_get_type to indicate whether the Get_Event service can wait for a new event element to be added to the event queue, if the event queue currently contains no elements (Get_Event waits under the caller's task.)
- Specify notify_type to indicate whether the Get_Event call is to be processed synchronously or asynchronously.

If you set event_get_type to 1 (atbcts_immediate), APPC/MVS returns an event element if one is immediately available. If there are no event elements, APPC/MVS provides return and reason codes that show the call was unsuccessful because no event elements were available. If you specify an event_get_type of 2 (atbcts_wait), APPC/MVS waits until an event element is available before returning control.

If you call Get_Event synchronously, APPC/MVS does not return control until Get_Event completes, or you *implicitly cancel* the request (for example, unregistering from the allocate queue causes APPC/MVS to cancel an outstanding Get_Event request for the queue). If you call Get_Event asynchronously, APPC/MVS returns control immediately, but monitors the event queue for an event element. APPC/MVS notifies the server by posting the specified ECB when the event for which notification was requested occurs.

Table 4 on page 25 shows the results for different combinations of notify_type and event_get_type:

Table 4. Results of Notify_Type/Event_Get_Type Combinations

Notify_Type Value:	**Event_Get_Type Value:**	
	atbcts_immediate	***atbcts_wait***
NONE	Get_Event returns an event element if one is available, or return code 16 (atbcts_request_unsuccessful) and reason code 30 (atbcts_no_event_available), if the event queue is currently empty.	The caller waits until an event element is retrieved, or this call is cancelled implicitly.
ECB	Control returns to the caller immediately, while APPC/MVS processes the Get_Event call asynchronously. If the event element is available, Get_Event returns the element. Otherwise, the caller receives return code 16 (atbcts_request_unsuccessful) and reason code 30 (atbcts_no_event_available).	Control returns to the caller immediately, while APPC/MVS processes the Get_Event call asynchronously. When an event element is available, Get_Event posts the caller's ECB.

Specifying the Event Buffer

To receive notification of an event, your server must supply a buffer on its call to Get_Event. APPC/MVS returns the event element in the event buffer.

The event buffer you specify must be large enough to contain the event element. If you specify a buffer that is not large enough, APPC/MVS sets the event_element_size parameter to the length it requires to satisfy the request. The Get_Event service returns to the server with return code 16 (atbcts_request_unsuccessful) and reason code 41 (atbcts_buffer_too_small). The server can create a larger event buffer and call the Get_Event service again to retrieve the event element successfully.

If you specify a buffer large enough to contain the event element, APPC/MVS sets the event_element_size parameter to the size of the event element that was returned, and the server receives a return code of zero.

Like most parameters returned from APPC/MVS callable services, the event buffer must reside in the caller's primary address space and be accessible to the caller's PSW key.

Effect of Unregister_For_Allocates

A call to the Unregister_For_Allocates service implicitly cancels a Get_Event if the call leaves the caller without allocate queues or outstanding calls to the Set_Allocate_Queue_Notification service.

Allowing the Allocate Queue to Persist

Use the Set_Allocate_Queue_Attributes service to indicate whether APPC/MVS is to maintain an allocate queue for which no servers are registered. Use this service when your server must stop serving an allocate queue for some interval of time, and will later resume serving the queue.

When calling the Set_Allocate_Queue_Attributes service, a server specifies a "keep time" for the allocate queue. The keep time is the number of seconds an allocate queue is maintained after the last server of the allocate queue unregisters. The server should set the allocate queue keep time only if it intends to resume serving the allocate queue within this time limit.

For example, you can use the Set_Allocate_Queue_Attributes service in situations where your server is to be restarted periodically. The server could register for a particular allocate queue and call the Set_Allocate_Queue_Attributes service, specifying a keep time for the allocate queue that would allow enough time for the server to be taken down and brought back up. The server could then re-register for the allocate queue, and resume receiving allocate requests.

To have an allocate queue cleaned up immediately after its last server unregisters, specify zero for the allocate queue keep time (or take the default of zero by not calling the Set_Allocate_Queue_Attributes service). In response, APPC/MVS rejects the allocate requests that reside on the queue with sense code X'084C0000' (TP_Not_Available_No_Retry). APPC/MVS attempts to place any new allocate requests that enter the system on another appropriate allocate queue if one exists, or attempts to schedule them. If there are no other appropriate allocate queues for inbound allocate requests, and they cannot be scheduled, APPC/MVS rejects the allocate requests with sense code X'084C0000' (TP_Not_Available_No_Retry).

If you specify a keep time for an allocate queue, however, APPC/MVS maintains all inbound allocate requests received for the server for the specified amount of time. During this period, APPC/MVS continues to add new inbound allocate requests for the server to the allocate queue.

A server can re-register for the allocate queue by calling Register_For_Allocates with the same parameter values it specified on its initial Register_For_Allocates call. The server must re-register for the allocate queue, however, before the specified keep time is exceeded, to serve any previously queued requests before they are purged with sense code X'084C0000' (TP_Not_Available_No_Retry).

When you specify a keep time, specify a value large enough to allow the server time to re-register for the allocate queue. Do not set the value too high, however, because the allocate requests will remain idle on the allocate queue, causing users to wait for their work to complete.

An allocate queue's keep time is zero by default. Only one keep time can be in effect for an allocate queue at a time, regardless of how many servers have registered for the queue. An allocate queue keep time specified on the Set_Allocate_Queue_Attributes service overrides the previous allocate queue keep time, if any. Also, if the last server of an allocate queue unregisters and re-registers for the same allocate queue before the allocate queue keep time has been exceeded, APPC/MVS resets the keep time to the value that was specified on the last call to the Set_Allocate_Queue_Attributes service.

Use of Unregister_For_Allocates

Use the Set_Allocate_Queue_Attributes service with the Unregister_For_Allocates service. When the last server of an allocate queue calls Unregister_For_Allocates, APPC/MVS checks the allocate queue's keep time, if any, and processes the queue as follows:

- If the keep time is zero, APPC/MVS purges the allocate queue (that is, the allocate requests are rejected).
- If the keep time is greater than zero, APPC/MVS maintains the allocate queue for the specified amount of time. During this time, APPC/MVS continues to add new inbound allocate requests to the allocate queue.

Multi-Tasking Servers

Because of the potentially heavy volume of transactions they will be serving, servers are likely to multi-task to handle multiple requests in parallel. This section lists common multi-tasking models you might consider for use in your server applications. The discussion refers to three models:

- Model One: "Empowerment"
- Model Two: "Management-Directed"
- Model Three: "Unmanaged".

Model One — Empowerment

In this model, the main task registers for inbound allocate requests and attaches one or more subtasks to handle each conversation.

Figure 8 on page 28 shows this type of multi-tasking server. In this case, the server registers for two different TP names (Steps 1 and 2), and dedicates two of its three subtasks to handling allocate requests from TPA. The main task attaches a third subtask to handle requests from TPB. The main task passes the allocate queue tokens it received from Register_For_Allocates (AQTOKEN1 and AQTOKEN2) to the subtasks so that they can begin processing requests from each allocate queue.

When invoked, each subtask calls the Receive_Allocate service (Steps 3 through 5), specifying the allocate queue token it received from the main task. Each subtask can now process requests from the appropriate allocate queue.

At the end of server processing, the main task unregisters from its allocate queues (Step 6).

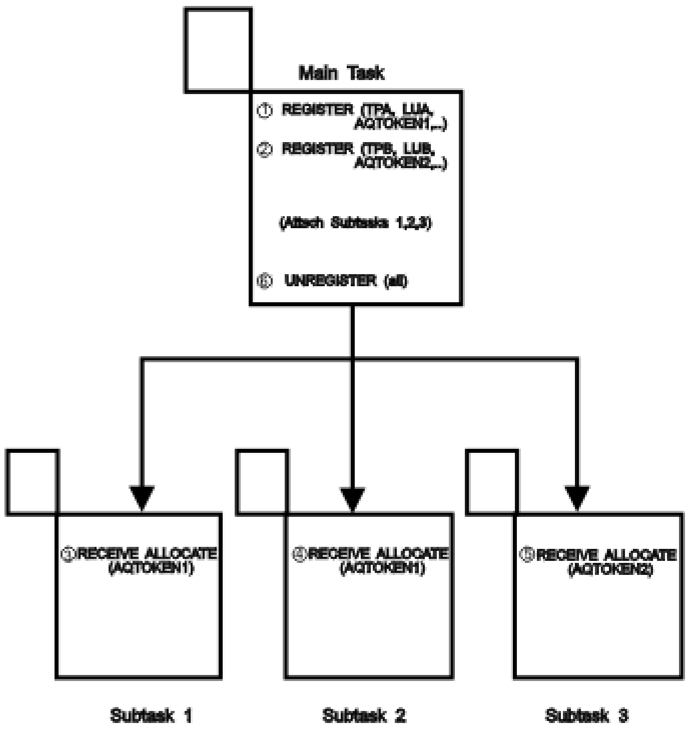
Main Task
① REGISTER (TPA, LUA, AQTOKEN1,...)
② REGISTER (TPB, LUB, AQTOKEN2,...)
(Attach Subtasks 1,2,3)
⑥ UNREGISTER (all)
③RECEIVE ALLOCATE (AQTOKEN1)
④RECEIVE ALLOCATE (AQTOKEN1)
⑤RECEIVE ALLOCATE (AQTOKEN2)
Subtask 1
Subtask 2
Subtask 3

Figure 8. Example of a Multi-tasking Server Address Space

Model Two — Management-Directed

This model is similar to the empowerment model, with the exception that the main task also calls the Receive_Allocate service. The main task passes the conversation IDs it receives to the subtasks (instead of allowing the subtasks to call Receive_Allocate).

This model provides a greater degree of centralized control within the main task. In this case, however, the main task must monitor its subtasks periodically to determine which subtasks are available to process a conversation.

Model Three — Unmanaged

This model is the opposite of the management-directed model. The main task attaches one or more subtasks which call Register_For_Allocates and Receive_Allocate. While this model poses more difficulties than the preceding models (for example, communication between the subtasks, if any, must be managed between the subtasks), you might find this model suitable to your particular application.

General Considerations for Multi-Tasking Servers

- Initialization.

  Regardless of the multi-tasking model you use, the server's main task should perform the initialization tasks, such as allocating common resources or issuing messages to operators.
- Security.

  Consider the potential risks of running multiple users in the same address space.
- Workload balancing.

  Consider using the allocate queue services described in "Receiving Notification of Events" on page 19 to manage the installation's workload efficiently. To simplify workload balancing, have the same task perform the event notification functions for the server.
- Processing protected conversations.

  Servers designed following the empowerment and unmanaged models may process protected conversations (conversations with a synchronization level of syncpt) as they would any other conversation, as long as each server subtask processes only one conversation at a time. Servers that are designed following the management-directed model, however, must register with registration services as a resource manager to process protected conversations using privately managed contexts.

If you want to code a management-directed model server to manage protected conversations, you need to understand the concepts and requirements for resource recovery in *z/OS MVS Programming: Resource Recovery*. Design this server and its subtasks to use allocate queue services, along with registration and context callable services, in the following sequence:

1. Register for TP/LU pairs through the Register_For_Allocates service. The LUs for which the server registers must be defined as capable of handling conversations with a synchronization level of syncpt. See the session management section of *z/OS MVS Planning: APPC/MVS Management* for further information about enabling LUs for protected conversation support.
2. Register with registration services through the Register_Resource_Manager service, supplying a resource manager name for this server. The service returns a resource manager token that the server uses on subsequent calls to registration and context services.
3. Call the Set_Exit_Information service to cause the server resource manager state to change to SET state. The server is now in the correct state with context services to issue context callable services.
4. Create a privately managed context through the Begin_Context service. The service returns a context token for the newly created context.
5. Switch to the newly created context by issuing a call to the Switch_Context service. After the service returns, the privately managed context is the current context.
6. Receive an inbound protected conversation by issuing the Receive_Allocate service. After the service returns, the identifier of this protected logical unit of work is associated with the privately managed context.
7. Attach a subtask to process the protected conversation. The server must pass to the subtask not only the conversation ID, but also the context token for the privately managed context. Once the subtask receives control, it must issue the Switch_Context service, specifying that context token.

The server may then repeat steps “4” on page 29 through “7” on page 29 to receive additional allocate queue requests, each with a different context, for additional subtasks to process. For servers, only one protected conversation may be associated with a context at one time; so a server may issue another Receive_Allocate call before deallocating a previous allocate request **only** through the use of privately managed contexts.

Managing Protected Conversations

To manage more than one protected conversation at a time, a server must register as a resource manager and use privately managed contexts to represent inbound allocate requests. Using privately managed contexts is only necessary for servers that are designed following the management-directed model. Other model types may process protected conversations as they would any other conversation, as long as each server subtask processes only one conversation at a time.

Note that designing and coding a server to act as a resource manager is relatively difficult. If you want to code a management-directed server to manage protected conversations, you need to:

- Understand the concepts and requirements for resource recovery in *z/OS MVS Programming: Resource Recovery*.
- Use the guidelines listed in “General Considerations for Multi-Tasking Servers” on page 28 to design and code a management-directed server to act as a resource manager.

Accounting for Server Usage

Your installation can track the use of server-specific resources through the SMF record type 33, subtype 2. Also, APPC/MVS servers can add client-specific information to these records to simplify accounting tasks.

Tracking Server-Specific Resources through SMF

When either partner program deallocates an APPC/MVS conversation, SMF writes a type 33, subtype 2 record. SMF provides detailed information about both partners, their associated LUs, and the amount of data transferred over the network. For a description of SMF accounting for APPC/MVS servers, see the section on APPC/MVS accounting in *z/OS MVS System Management Facilities (SMF)*.

Adding User Data to Accounting Records

Servers can write up to 255 bytes of user-defined data to these accounting records through the Set_Conversation_Accounting_Information service. The user data also appears in the user data field that the Extract_Information service returns when programs extract information about specific conversations.

You can use the user data field to charge resources to a specific conversation, or to correlate outbound conversations to inbound conversations, by specifying a conversation ID in the user data, for example.

For information about using the Set_Conversation_Accounting_Information service, see *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*.

Performance Considerations for Allocate Queue Services

The relative performance of APPC/MVS callable services varies depending on the functions that the callable service performs. For example, services that call VTAM or cause the movement of data buffers involve a greater number of internal instructions.

Table 5 on page 30 provides a summary of performance considerations for individual APPC/MVS allocate queue callable services. For an overview of performance considerations for APPC/MVS TP callable services, see *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*.

Table 5. Performance Considerations for Allocate Queue Services

APPC/MVS Service	Calls VTAM	Causes DASD I/O	Causes buffer moves	Calls RACF	Creates SMF record
Register_For_Allocates	No	See **Note**	No	RACROUTE REQUEST= AUTH	No
Receive_Allocate	No	No	No	No	No
Unregister_For_Allocates	No	No	No	No	No
Query_Allocate_Queue	No	No	No	No	No
Set_Allocate_Queue_Notification	No	No	No	No	No
Get_Event	No	No	No	No	No
Set_Allocate_Queue_Attributes	No	No	No	No	No

Note: Reads from the side information file if you specify a symbolic destination name.

Installing APPC/MVS Servers

This section is provided mainly as background information. In most cases, the server writer does not have to do more than step 1. You might, however, want to check with the system programmer and security administrator to make sure that other steps have been taken.

For detailed information on configuring APPC/MVS, see *z/OS MVS Planning: APPC/MVS Management*.

To install your server, do the following:

1. Install the server program on the z/OS system. Unauthorized server programs can reside in any library. Authorized server programs must reside in an authorized library.
2. The system programmer defines NOSCHED LUs, if needed.
3. The system programmer defines TP profile data sets, if needed. Include database tokens (dbtokens) for servers.
4. The security administrator does the following:
    a. For started servers, assigns user IDs by modifying the RACF started procedures table (module ICHRIN03). For information about the RACF started procedures table, see *z/OS Security Server RACF Security Administrator's Guide*.
    b. Defines RACF profiles in APPCSERV class, and protects the APPCSERV class with UACC(NONE). For example:

    ```
    RDEFINE APPCSERV dbtoken.tpname UACC(NONE)
    ```

    where *dbtoken* identifies the client's local LU, and tpname is the name of the client TP.
    c. Activates the APPCSERV class, and activates RACLIST processing for the class. For example:

    ```
    SETROPTS CLASSACT(APPCSERV) RACLIST(APPCSERV)
    ```

    d. Grants READ access to individual server user IDs for the client's profile in the APPCSERV class. The security administrator could use the following RACF command:

    ```
    PERMIT dbtoken.tpname CLASS(APPCSERV) ACCESS(READ) ID(userid)
    ```

    For information about defining RACF security classes and using RACF commands, see *z/OS Security Server RACF Security Administrator's Guide*.
5. Tuning: The system programmer can specify SRM tuning parameters for server address spaces, if desired.

Diagnosing Problems with APPC/MVS Servers

Using the API trace facility, programmers can collect data about each call that an APPC/MVS server issues for any APPC/MVS allocate queue service, except for Get_Event and Set_Allocate_Queue_Notification. For the API trace facility to collect trace data for these services, the server must specify a valid allocate queue token on those services that accept an allocate queue token as input.

See *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS* for more information about the API trace facility.

Chapter 3. Invocation Details for Allocate Queue Services

Objective
This chapter describes the available programming languages, syntax and linkage conventions, and parameter types for the APPC/MVS allocate queue callable services.

The figure below shows a partial list of high-level language compilers that support APPC/MVS calls. Calls can be made with other compiler levels and other compiler products that meet the requirements and linkage conventions described in this chapter.

To use allocate queue services, programs must run in 31-bit addressing mode. Note that this requirement might limit some language functions you can use.

Table 6. Some High-Level Language Compilers for APPC/MVS Calls

Language	**Compiler**
C	C/370 Compiler Version 1, Release 2.0
COBOL	VS COBOL II, Release 2.0
FORTRAN	VS FORTRAN Compiler Version 2, Release 3.0
PL/I	OS PL/I Compiler Version 2, Release 2.1
REXX	TSO/E Version 2, Release 5

Interface Definition Files (IDFs)

IBM provides IDFs (also called pseudonym files or headers) for the preceding high level languages. IDFs include prototypes for each call and constant declarations for parameter, return code, and reason code values.

"Interface Definition Files (IDFs) for APPC/MVS Services" on page 36 lists the IDFs available for use with APPC/MVS allocate queue services.

Syntax and Linkage Conventions for Allocate Queue Services

All allocate queue callable services have a general calling syntax as follows:

```
CALL routine_name  (parameters,reason_code,return_code)
```

Some specific calling formats for languages that can call APPC/MVS services are:

COBOL
CALL "routine_name" USING parm1,parm2,...reason_code,return_code

FORTRAN
CALL routine_name (parm1,parm2,...reason_code,return_code)

C
routine_name (parm1,parm2,...reason_code,return_code)

PL/I
CALL routine_name (parm1,parm2,...reason_code,return_code)

REXX

ADDRESS APPCMVS “routine_name parm1 parm2...reason_code return_code”

For REXX, enclose the routine name and all parameters within one pair of single or double quotes. Parameters must be initialized to appropriate values. The host command environment resolves the parameter values. For more information, see *z/OS TSO/E REXX Reference*.

Assembler Call macro

CALL routine_name,(parm1,parm2,...reason_code,return_code),VL

Note that all allocate queue services return (as parameters) a return and reason code that shows the result of the call.

Linkage Conventions

Callers must use the following linkage conventions for all allocate queue callable services:

- Register 1 must contain the address of a parameter list, which is a list of consecutive words, each containing the address of a parameter to be passed. The last word in this list must have a 1 in the high-order (sign) bit.
- Register 13 must contain the address of an 18-word save area.
- Register 14 must contain the return address.
- Register 15 must contain the entry point address of the service being called.
- If the caller is running in AR ASC mode, access registers 1, 13, 14, and 15 must all be set to zero.

On return from the service, general and access registers 2 through 14 are restored (registers 0, 1 and 15 are not restored).

Any high-level language that generates this type of interface can be used to call allocate queue services.

Parameter Description for Allocate Queue Services

All the parameters of the allocate queue callable services are required positional parameters. When you call a service, you must specify all the parameters in the order listed. APPC/MVS checks all parameters for valid values, regardless of whether the parameters are used in call processing. Even though a language may allow parameters to be omitted, APPC/MVS services do not.

Some parameters do not require values and allow you to substitute zeroes or a string of blanks for the parameter. For example, if you do not specify a symbolic destination on the Register_For_Allocates call, you must set the Sym_dest_name parameter to eight blanks. The descriptions of the parameters identify those which can be replaced by blanks or zeroes, and when to do so.

Each service has returned to it (through parameters) a return and reason code that shows the result of the call. If the return code is set to zero or decimal 64, APPC/MVS does not set the reason code (and thus it need not be checked). If the return code is a non-zero value other than decimal 64, APPC/MVS sets the reason code to show the reason for the bad return code.

In the descriptions of services in this book, each parameter is described as supplied or returned:

Supplied

You supply a value for the parameter in the call.

Returned

The service returns a value in the named parameter when the call is finished (for example, *return_code*).

Each parameter is also described in terms of its data type, character set, and length:

Data type

Either integer, character string, or structure.

Character set

Applies only to parameters whose values are character strings and governs the values allowed for that parameter. Possible character sets are:

- No restriction

  There is no restriction on the byte values contained in the character string.
- Type A EBCDIC

  The string may contain only uppercase alphabetics, numerics, and national characters (@, $, #) and must begin with an alphabetic or national character. Use of @, $, and # is discouraged because those characters display differently on different national code pages.
- 01134

  The string may contain uppercase alphabetics or numerics, with no restriction on the first character.
- 00640

  The string may contain upper- or lowercase alphabetics, numerics, or any of 19 special characters with no restriction on the first character.

  Note: APPC/MVS does not allow blanks in 00640 character strings.

For more detailed information about the characters in each character set, see Appendix A, “Character Sets,” on page 75.

Length
Depends on the data type of the parameter:

- For an integer item, the length shows the size of the field in bits.
- For a character string parameter, the length value shows the number of characters that may be contained in a character type parameter. The length may specify a single number or a minimum and maximum number.
- For a structure parameter, the length value shows the size of the structure in bytes, or a minimum and maximum size if the size of the structure is variable.

Required Modules

Programs that use APPC/MVS callable services must have addressability to modules in SYS1.CSSLIB that define the services. You can use either of the following methods described here to access APPC/MVS callable services.

- Method One — Have your program issue the MVS LOAD macro for the APPC/MVS service to obtain its entry point address. Use that address to call the APPC/MVS service.
- Method Two — Link-edit one or more of the following modules from SYS1.CSSLIB with any program that issues APPC/MVS services:
  - ATBPBI — with programs that issue CPI Communications calls or TP conversation services
  - ATBATP — with programs that issue APPC/MVS advanced TP services
  - ATBCTS — with programs that issue APPC/MVS allocate queue services, Reject_Conversation, or Set_Conversation_Accounting_Information.
  - ATBCSS — with programs that issue APPC/MVS system services.

  If you use this second method, you must again link-edit these modules and any load modules containing copies of them with your APPC/MVS applications after new releases of MVS are installed, or maintenance is applied that affects APPC/MVS callable services. In this case, provide a post-install job to re-link-edit the modules with your APPC/MVS applications.

Additional language-specific statements may be necessary so that language compilers can provide the proper assembler interface. Other programming notation, such as variable declarations, are also language-dependent.

Versions of Callable Services

APPC/MVS allocate queue callable services have a version number as the last character of the call name (for example, ATBRFA2). That number corresponds to the version of APPC/MVS in which the call was introduced.

To determine which calls are valid on a system, you can obtain the current APPC/MVS version number from the APPC/MVS Version service. For more information about the APPC/MVS Version service, see *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*.

Interface Definition Files (IDFs) for APPC/MVS Services

IBM provides IDFs (also called pseudonym files or headers) that define variables and values for parameters of APPC/MVS services. IDFs are available for different languages, and can be included or copied from a central library into programs that call APPC/MVS services.

For a list of the IDFs available for use with CPI-C, LU 6.2, and APPC/MVS services, see *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*.

IBM provides the following IDFs for programs that call APPC/MVS allocate queue services, the Reject_Conversation service or the Set_Conversation_Accounting_Information service:

Language	**SYS1.MACLIB Member**
Assembler	ATBCTASM in SYS1.MACLIB

Table 7. IDFs for Allocate Queue Services

Language	**In member:**
C	ATBCTC in SYS1.SIEAHDR.H
Note: ATBCTC is also shipped in the z/OS UNIX System Services HFS directory /usr/include.	
COBOL	ATBCTCOB in SYS1.SIEAHDR.H
FORTRAN	ATBCTFOR in SYS1.SIEAHDR.H
PL/I	ATBCTPLI in SYS1.SIEAHDR.H
REXX	ATBCTREX in SYS1.SIEAHDR.H

Chapter 4. APPC/MVS Allocate Queue Services

Objective
This chapter provides the syntax, parameter descriptions, and return and reason codes for the APPC/MVS allocate queue callable services.

The services are:

Get_Event

APPC/MVS servers use the Get_Event service to retrieve the next event element from the server's event queue.

Environment for Get_Event

The requirements for the caller are:

Minimum authorization:	Problem state, with any PSW key
Dispatchable unit mode:	Task or SRB
Cross memory mode:	Any PASN, any HASN, any SASN
AMODE:	31-bit
ASC mode:	Primary or access register (AR)
Interrupt status:	Enabled for I/O and external interrupts
Locks:	No locks held
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Restrictions

When using this service, observe the following restrictions:

- Programs can have only one outstanding call to the Get_Event service at a time. If a program attempts to call the Get_Event service while the program has another Get_Event call outstanding, APPC/MVS rejects the subsequent call.
- Programs that call the Get_Event service while in task mode should not have any enabled unlocked task (EUT) functional recovery routines (FRRs) established. For more information about EUT FRRs, see the section on providing recovery in *z/OS MVS Programming: Authorized Assembler Services Guide*.

Input Register Information

Before calling the Get_Event service, the caller must ensure that the following GPRs contain the specified information:

Register	Contents
1	Address of the parameter list
13	Address of a standard 18-word save area
14	Return address
15	Entry point address of the service being called.

Before calling the Get_Event service, the caller must ensure that the following access registers (ARs) contain the specified information:

Register	Contents
1	0
13 - 15	0

Output Register Information

When control returns to the caller of the Get_Event service, the general purpose registers (GPRs) contain:

Register	Contents
0 - 1	Used as work registers by the system
2 - 14	Unchanged
15	Used as a work register by the system

When control returns to the caller of the Get_Event service, the access registers (ARs) contain:

Register	Contents
0 - 1	Used as work registers by the system
2 - 14	Unchanged
15	Used as a work register by the system

Some callers depend on register contents remaining the same before and after issuing a service. If the system changes the contents of registers on which the caller depends, the caller must save them before issuing the service, and restore them after the system returns control.

Syntax Format for Get_Event

The figure below shows the syntax of the CALL statement for the Get_Event service. You must code all parameters on the CALL statement in the order shown.

```
CALL ATBGTE2(
        Notify_type,
        Event_get_type
        Event_code,
        Event_timestamp,
        Event_buffer_length,
        Event_buffer,
        Event_element_size,
        Reason_code,
        Return_code
       );
```

Figure 9. Syntax for Get_Event

Parameters for Get_Event

The following section describes the parameters you specify when calling the Get_Event service.

Notify_type

Supplied parameter

- Type: Structure
- Length: 4-8 bytes

Specifies the type of processing and notification (synchronous or asynchronous) requested for this service. The possible types are:

- None

  No notification is requested. APPC/MVS processes this call synchronously, and returns control to the caller when processing is complete. APPC/MVS sets all returned parameters on return to the caller. To specify no notification, set this parameter to a four-byte structure that contains binary zeroes.
- ECB

  Programs can request asynchronous processing by specifying an ECB to be posted when processing completes. To specify an ECB, set this parameter to an eight-byte structure that contains a fullword binary one (X'00000001'), followed by the address of a fullword area to be used as the ECB. The ECB must reside in the caller's home address space.

  When you specify an ECB, APPC/MVS returns control to the caller before processing is complete, with only the return code set. If APPC/MVS accepts the asynchronous request, it sets the return code to 0 to show that it is processing the service asynchronously. APPC/MVS fills in the other returned parameters during asynchronous processing, and posts the specified ECB when it has set all the returned parameters. The completion code field in the ECB contains the return code for the service. APPC/MVS places the reason code, if any, in the server's reason_code parameter.

Event_get_type

Supplied parameter

- Type: Integer
- Length: 32 bits

Specifies whether the call is allowed to wait for a new event element to be added to the event queue, if the event queue currently contains no elements.

Valid values for this parameter are:

Value

Meaning

1
atbcts_immediate

Control is to be returned to the caller immediately, regardless of whether an event element can be retrieved. If an event element is not immediately available, APPC/MVS sets return_code to 16 (atbcts_request_unsuccessful) and reason_code to 30 (atbcts_no_event_available).

2
atbcts_wait

This call remains active until an event element can be retrieved, or the call is cancelled.

Event_code
Returned parameter

- Type: Integer
- Length: 32 bits

Contains a value that indicates which event has occurred.

Possible values returned for this parameter are:

Value
Meaning

1
atbcts_allocate_queue_min

The allocate queue indicated in the event element has decreased to its minimum threshold.

2
atbcts_allocate_queue_max

The allocate queue indicated in the event element has increased to its maximum threshold.

Event_timestamp
Returned parameter

- Type: Character string
- Char Set: N/A
- Length: 8 bytes

Contains a timestamp that shows when the event occurred. This timestamp is in the format provided by the STORE CLOCK (STCK) assembler instruction.

Event_buffer_length
Supplied parameter

- Type: Integer
- Length: 32 bits

Specifies the length, in bytes, of the event buffer (the event_buffer parameter) where the event element is to be placed.

Event_buffer
Supplied/Returned parameter

- Type: Character string
- Char Set: No restriction
- Length: Variable (specified in event_buffer_length parameter)

Specifies an area of storage where the event element is to be placed. The event buffer must reside in the caller's primary address space and be accessible to the caller's PSW key.

If the buffer you specify is large enough to contain the event element, APPC/MVS places the event element in this buffer on return to the caller. APPC/MVS sets the event_element_size parameter to the length of the storage used to contain the returned element.

If the buffer you specify is not large enough, APPC/MVS fails the Get_Event request with return code 16 (atbcts_request_unsuccessful) and reason code 41 (atbcts_buffer_too_small). APPC/MVS sets the event_element_size parameter to the length of storage it requires to satisfy this request. To retrieve the event element successfully, specify an event buffer equal to the event_element_size and call the Get_Event service again.

To determine whether the event was a minimum or maximum threshold, check the value returned in the event_code parameter. Table 8 on page 41 shows the contents of event_buffer, based on the event_code value:

Table 8. Relationship Between Event_code and Event_buffer

Event_code Value:	**Event_buffer Contents:**
1	The event buffer contains two fields (12 bytes): • ***Allocate_queue_token*** (character string - 8 bytes). Indicates the allocate queue for which this event applies. • ***Minimum queue size*** (integer - 32 bits). Indicates that the specified allocate queue decreased to this size.
2	The event buffer contains two fields (12 bytes): • ***Allocate_queue_token*** (character string - 8 bytes). Indicates the allocate queue for which this event applies. • ***Maximum queue size*** (integer - 32 bits). Indicates that the specified allocate queue increased to this size.

Event_element_size

Returned parameter

- Type: Integer
- Length: 32 bits

Contains the length of the event element that was retrieved, or the length that APPC/MVS requires to satisfy this request (if the call is unsuccessful).

To determine whether the request was successful, check the value in the return_code parameter, as follows:

- If return_code contains zero, the event_buffer parameter contains the event element. APPC/MVS sets event_element_size to the length of the event element that was returned.
- If return_code contains 16 (atbcts_request_unsuccessful), and reason_code contains 41 (atbcts_buffer_too_small), event_element_size contains the length required to satisfy the request. To retrieve the event element successfully, create a larger event buffer equal to this length and call the Get_Event service again.

Reason_code

Returned parameter

- Type: Integer
- Length: 32 bits

Contains additional information about the result of the call when the return_code parameter contains a non-zero value other than decimal 64 (atbcts_appc_not_available).

Table 9 on page 42 lists the valid reason codes.

Return_code

Returned parameter

- Type: Integer
- Length: 32 bits

Contains the result of the call. If the return_code parameter contains zero or decimal 64 (atbcts_appc_not_available), there is no reason code. For other return codes, check the reason_code parameter for additional information about the result of the call.

Table 9 on page 42 lists the valid return and reason codes for the Get_Event service.

Table 9. Return and Reason Codes for Get_Event

Return Code (Decimal)	Reason Code (Decimal)	Symbolic Value
0		atbcts_ok
8		atbcts_parameter_error
	18	atbcts_inval_notify_type
	37	atbcts_inval_event_get_type
16		atbcts_request_unsuccessful
	7	atbcts_parameter_inaccessible
	8	atbcts_cannot_hold_locks
	20	atbcts_request_cancelled
	30	atbcts_no_event_available
	31	atbcts_event_notify_cancelled
	32	atbcts_get_event_outstanding
	33	atbcts_notify_not_set
	41	atbcts_buffer_too_small
32		atbcts_service_failure
	16	atbcts_appc_service_failure
64		atbcts_appc_not_available

For more detailed information about these return codes and reason codes, refer to Appendix B, “Explanation of Return and Reason Codes,” on page 79.

Abend Codes for Get_Event

The caller might encounter abend X'EC7' with either of the reason codes shown below:

Table 10. Abend Codes for Get_Event

Abend Code (Hexadecimal)	Reason Code (Hexadecimal)	Description
X'EC7'	X'00140009'	The number of parameters specified is incorrect.

Table 10. Abend Codes for Get_Event (continued)

Abend Code (Hexadecimal)	Reason Code (Hexadecimal)	Description
X'EC7'	X'0014000A'	APPC/MVS cannot access one or more of the specified parameters.

See *z/OS MVS System Codes* for an explanation and response for these codes.

Query_Allocate_Queue

APPC/MVS servers use the Query_Allocate_Queue service to obtain commonly needed information about the status of an allocate queue for which the caller is registered.

Environment for Query_Allocate_Queue

The requirements for the caller are:

Minimum authorization:	Problem state, with any PSW key
Dispatchable unit mode:	Task or SRB
Cross memory mode:	Any PASN, any HASN, any SASN
AMODE:	31-bit
ASC mode:	Primary or access register (AR)
Interrupt status:	Enabled for I/O and external interrupts
Locks:	No locks held
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Restrictions

Programs that call the Query_Allocate_Queue service while in task mode should not have any enabled unlocked task (EUT) functional recovery routines (FRRs) established. For more information about EUT FRRs, see the section on providing recovery in *z/OS MVS Programming: Authorized Assembler Services Guide*.

Input Register Information

Before calling the Query_Allocate_Queue service, the caller must ensure that the following GPRs contain the specified information:

Register
  Contents

1
  Address of the parameter list

13
  Address of a standard 18-word save area

14
  Return address

15
  Entry point address of the service being called.

Before calling the Query_Allocate_Queue service, the caller must ensure that the following access registers (ARs) contain the specified information:

Register
Contents

1
0

13 - 15
0

Output Register Information

When control returns to the caller of the Query_Allocate_Queue service, the general purpose registers (GPRs) contain:

Register
Contents

0 - 1
Used as work registers by the system

2 - 14
Unchanged

15
Used as a work register by the system

When control returns to the caller of the Query_Allocate_Queue service, the access registers (ARs) contain:

Register
Contents

0 - 1
Used as work registers by the system

2 - 14
Unchanged

15
Used as a work register by the system

Some callers depend on register contents remaining the same before and after issuing a service. If the system changes the contents of registers on which the caller depends, the caller must save them before issuing the service, and restore them after the system returns control.

Syntax Format for Query_Allocate_Queue

The figure below shows the syntax of the CALL statement for the Query_Allocate_Queue service. You must code all parameters on the CALL statement in the order shown.

```
CALL ATBQAQ2(
        Notify_type,
        Allocate_queue_token,
        TP_name_length,
        TP_name,
        Local_LU_name,
        Allocate_queue_size,
        Allocate_queue_oldest,
        Last_rec_alloc_issued
        Last_rec_alloc_returned
        Reason_code,
        Return_code
       );
```

Figure 10. Syntax for Query_Allocate_Queue

Parameters for Query_Allocate_Queue

The following section describes the parameters you specify when calling the Query_Allocate_Queue service.

Notify_type

Supplied parameter

- Type: Structure
- Length: 4-8 bytes

Specifies the type of processing and notification (synchronous or asynchronous) requested for this service. The possible types are:

- None

  No notification is requested. APPC/MVS processes this call synchronously, and returns control to the caller when processing is complete. APPC/MVS sets all returned parameters on return to the caller. To specify no notification, set this parameter to a four-byte structure that contains binary zeroes.

- ECB

  Programs can request asynchronous processing by specifying an ECB to be posted when processing completes. To specify an ECB, set this parameter to an eight-byte structure that contains a fullword binary one (X'00000001'), followed by the address of a fullword area to be used as the ECB. The ECB must reside in the caller's home address space.

  When you specify an ECB, APPC/MVS returns control to the caller before processing is complete, with only the return code set. If APPC/MVS accepts the asynchronous request, it sets the return code to 0 to show that it is processing the service asynchronously. APPC/MVS fills in the other returned parameters during asynchronous processing, and posts the specified ECB when it has set all the returned parameters. The completion code field in the ECB contains the return code for the service. APPC/MVS places the reason code, if any, in the server's reason_code parameter.

Allocate_queue_token

Supplied parameter

- Type: Character string
- Length: 8 bytes

Specifies the allocate queue token that indicates the particular allocate queue on which the query is to be performed. (The allocate queue token is returned as output from a successful call to the Register_For_Allocates service.)

TP_name_length

Returned parameter

- Type: Integer
- Length: 32 bits
- Range: 1-64

Specifies the length of the data contained in the TP_name parameter.

TP_name

Returned parameter

- Type: Character string
- Char Set: 00640 or Type A (Type A if the TP is protected by RACF)
- Length: 1 - 64 bytes

Contains the TP name specified on the Register_For_Allocates call for this allocate queue.

Local_LU_name

Returned parameter

- Type: Character string
- Char Set: Type A
- Length: 8 bytes

Contains the name of the local LU associated with the allocate requests in the allocate queue.

Allocate_queue_size
Returned parameter

- Type: Integer
- Length: 32 bits

Contains the number of allocate requests that currently reside on the specified allocate queue.

Allocate_queue_oldest
Returned parameter

- Type: Integer
- Length: 32 bits

Contains the age (in seconds) of the oldest allocate request on the allocate queue.

Last_rec_alloc_issued
Returned parameter

- Type: Character
- Char Set: N/A
- Length: 8 bytes

Contains a timestamp that shows when the caller last called the Receive_Allocate service.

The time is in the format provided by the STORE CLOCK (STCK) assembler instruction.

If the caller has not called the Receive_Allocate service for the specified allocate queue, the system sets this parameter to binary zeroes.

Last_rec_alloc_returned
Returned parameter

- Type: Character
- Char Set: N/A
- Length: 8 bytes

Contains a timestamp that shows when the Receive_Allocate service most recently returned control to the caller for the specified allocate queue (regardless of whether the call successfully returned an allocate request).

The time is in the format provided by the STORE CLOCK (STCK) assembler instruction.

If the Receive_Allocate service has not yet returned control to the caller for the specified allocate queue, the system sets this parameter to binary zeroes.

Reason_code
Returned parameter

- Type: Integer
- Length: 32 bits

Contains additional information about the result of the call when the return_code parameter contains a non-zero value other than decimal 64 (atbcts_appc_not_available).

Table 11 on page 47 lists the valid reason codes.

Return_code
Returned parameter

- Type: Integer
- Length: 32 bits

Contains the result of the call. If the return_code parameter contains zero or decimal 64 (atbcts_appc_not_available), there is no reason code. For other return codes, check the reason_code parameter for additional information about the result of the call.

Table 11 on page 47 lists the valid return and reason codes for the Query_Allocate_Queue service.

Table 11. Return and Reason Codes for Query_Allocate_Queue

Return Code (Decimal)	Reason Code (Decimal)	Symbolic Value
0		atbcts_ok
8		atbcts_parameter_error
	17	atbcts_inval_alloc_queue_token
	18	atbcts_inval_notify_type
16		atbcts_request_unsuccessful
	7	atbcts_parameter_inaccessible
	8	atbcts_cannot_hold_locks
	20	atbcts_request_cancelled
32		atbcts_service_failure
	16	atbcts_appc_service_failure
64		atbcts_appc_not_available

For more detailed information about these return codes and reason codes, see Appendix B, “Explanation of Return and Reason Codes,” on page 79.

Abend Codes for Query_Allocate_Queue

The caller might encounter abend X'EC7' with either of the reason codes shown below:

Table 12. Abend Codes for Query_Allocate_Queue

Abend Code (Hexadecimal)	Reason Code (Hexadecimal)	Description
X'EC7'	X'0014000B'	The number of parameters specified is incorrect.
X'EC7'	X'0014000C'	APPC/MVS cannot access one or more of the specified parameters.

See *z/OS MVS System Codes* for an explanation and response for these codes.

Receive_Allocate

APPC/MVS servers use the Receive_Allocate service to retrieve the oldest allocate request from a particular allocate queue. Note that servers use the Receive_Allocate service, instead of the Accept_Conversation or Get_Conversation calls, which are commonly used by scheduled transaction programs, to receive inbound conversations.

Environment for Receive_Allocate

The requirements for the caller are:

Minimum authorization:	Problem state, with any PSW key
Dispatchable unit mode:	Task or SRB
Cross memory mode:	Any PASN, any HASN, any SASN
AMODE:	31-bit
ASC mode:	Primary or access register (AR)
Interrupt status:	Enabled for I/O and external interrupts
Locks:	No locks held
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Restrictions

Programs that call the Receive_Allocate service while in task mode should not have any enabled unlocked task (EUT) functional recovery routines (FRRs) established. For more information about EUT FRRs, see the section on providing recovery in *z/OS MVS Programming: Authorized Assembler Services Guide*.

Input Register Information

Before calling the Receive_Allocate service, the caller must ensure that the following GPRs contain the specified information:

Register	Contents
1	Address of the parameter list
13	Address of a standard 18-word save area
14	Return address
15	Entry point address of the service being called.

Before calling the Receive_Allocate service, the caller must ensure that the following access registers (ARs) contain the specified information:

Register	Contents
1	0
13 - 15	0

Output Register Information

When control returns to the caller of the Receive_Allocate service, the general purpose registers (GPRs) contain:

Register	Contents
0 - 1	Used as work registers by the system
2 - 14	Unchanged

15
Used as a work register by the system

When control returns to the caller of the Receive_Allocate service, the access registers (ARs) contain:

Register	Contents
0 - 1	Used as work registers by the system
2 - 14	Unchanged
15	Used as a work register by the system

Some callers depend on register contents remaining the same before and after issuing a service. If the system changes the contents of registers on which the caller depends, the caller must save them before issuing the service, and restore them after the system returns control.

Syntax Format for Receive_Allocate

The figure below shows the syntax of the CALL statement for the Receive_Allocate service. You must code all parameters on the CALL statement in the order shown.

```
CALL ATBRAL2(
        Notify_type,
        Allocate_queue_token,
        Receive_allocate_type,
        Time_out_value,
        Conversation_ID,
        Conversation_type,
        Partner_LU_name,
        Mode_name,
        Sync_level,
        User_ID,
        Profile,
        Reason_code,
        Return_code
       );
```

Figure 11. Syntax for Receive_Allocate

Parameters for Receive_Allocate

The following section describes the parameters you specify when calling the Receive_Allocate service.

Notify_type
Supplied parameter

- Type: Structure
- Length: 4-8 bytes

Specifies the type of processing and notification (synchronous or asynchronous) requested for this service. The possible types are:

- None

  No notification is requested. APPC/MVS processes this call synchronously, and returns control to the caller when processing is complete. APPC/MVS sets all returned parameters on return to the caller. To specify no notification, set this parameter to a four-byte structure that contains binary zeroes.
- ECB

  Programs can request asynchronous processing by specifying an ECB to be posted when processing completes. To specify an ECB, set this parameter to an eight-byte structure that contains a fullword binary one (X'00000001'), followed by the address of a fullword area to be used as the ECB. The ECB must reside in the caller's home address space.

When you specify an ECB, APPC/MVS returns control to the caller before processing is complete, with only the return code set. If APPC/MVS accepts the asynchronous request, it sets the return code to 0 to show that it is processing the service asynchronously. APPC/MVS fills in the other returned parameters during asynchronous processing, and posts the specified ECB when it has set all the returned parameters. The completion code field in the ECB contains the return code for the service. APPC/MVS places the reason code, if any, in the server's reason_code parameter.

Allocate_queue_token

Supplied parameter

- Type: Character string
- Length: 8 bytes

Specifies the allocate queue token that indicates the particular allocate queue from which the allocate request is to be received. (The allocate queue token is returned as output from a successful call to the Register_For_Allocates service.)

Receive_allocate_type

Supplied parameter

- Type: Integer
- Length: 32 bits

Specifies whether the caller is allowed to wait to receive an allocate request if one is not immediately available (such as when the allocate queue is empty) and, if so, for how long.

Valid values for this parameter are:

Value

Meaning

1

atbcts_immediate

Control is to be returned to the caller immediately, regardless of whether an allocate request can be received. If an allocate request is not immediately available, APPC/MVS sets the return_code parameter to return code 16 (atbcts_request_unsuccessful) and the reason_code parameter to reason code 21 (atbcts_no_alloc_to_receive).

2

atbcts_wait

The call to this service does not complete until an allocate request is received.

3

atbcts_timed

This call does not complete until an allocate request is received, or the time specified in the timeout_value parameter is exceeded. If no allocate request is available before the timeout value is exceeded, APPC/MVS sets the return_code parameter to return code 16 (atbcts_request_unsuccessful) and the reason_code parameter to reason code 21 (atbcts_no_alloc_to_receive).

Set this parameter with the notify_type and time_out_value parameters, as follows:

- Notify_type specifies synchronous and asynchronous processing for this call. (For the results of specifying the receive_allocate_type parameter with the notify_type parameter, see Table 3 on page 14.)
- When the call is to wait a specified amount of time for an inbound allocate to become available, specify the amount of time (in seconds) on the time_out_value parameter.

Time_out_value

Supplied parameter

- Type: Integer
- Length: 32 bits
- Range: 0 - 1,000,000

Specifies, in seconds, the maximum amount of time the caller is allowed to wait to receive an allocate request, if one is not immediately available (because the allocate queue is empty).

The time_out_value parameter is used only when the caller sets the receive_allocate_type parameter to atbcts_timed. Otherwise, this parameter has no meaning and is ignored after APPC/MVS checks it to see that it contains a valid value. Set this parameter to zero if you are not using it.

Conversation_ID

Returned parameter

- Type: Character string
- Char Set: No restriction
- Length: 8 bytes

Contains, on return to the caller, the conversation_ID of the conversation that was received. The conversation_id, sometimes called the resource identifier, identifies a conversation to the system. The caller uses the conversation ID to hold subsequent APPC communications with the client transaction program.

Conversation_type

Returned parameter

- Type: Integer
- Length: 32 bits

Contains the type of conversation that was received.

Valid values for this parameter are:

Value

Meaning

0

Basic_conversation

In this conversation, the caller and its partner program are to format their data into separate records, with record length and data specified, before sending it.

1

Mapped_conversation

In this conversation, the caller and its partner are to rely on APPC to format the data they send.

Partner_LU_name

Returned parameter

- Type: Character string
- Char Set: Type A
- Length: 17 bytes, padded with blanks and left justified.

Contains the name of the LU at which the partner program is located. This value is the combined network_ID and network LU name (two 1-8 byte Type A character strings, concatenated by a period: *network_ID.network_LU_name*). This format is known as a **network-qualified LU name**. When VTAM is not active, the network_ID is not defined and only the network LU name is returned to the caller.

If the partner LU is a member of a VTAM generic resource group, the network-LU-name portion of the partner LU name might be a generic resource name.

Mode_name
Returned parameter

- Type: Character string
- Char Set: Type A
- Length: 8 bytes

Contains the mode name that specifies the network properties of the session allocated for this conversation.

Sync_level
Returned parameter

- Type: Integer
- Length: 32 bits

Contains the synchronization level that the caller and its partner program are to use on this conversation.

Valid values for this parameter are:

Value
Meaning

0
None

The programs will not perform confirmation processing on this conversation.

1
Confirm

The programs can perform confirmation processing on this conversation. The programs will recognize returned parameters relating to confirmation.

2
Syncpt

The programs can perform syncpoint processing on this conversation. The programs may issue Commit or Backout calls and will recognize returned parameters related to this syncpoint processing.

User_ID
Returned parameter

- Type: Character string
- Char Set: Type A
- Length: 8 bytes

Contains the user ID that is associated with the received conversation. If no user ID is associated with the conversation, this parameter contains blanks.

Profile
Returned parameter

- Type: Character string
- Char Set: Type A
- Length: 8 bytes

Contains the RACF group name associated with the conversation.

If APPC/MVS does not return a value for this parameter, and does return a value for the user_id parameter, APPC/MVS used a default RACF profile for the user ID. If neither a profile nor a user ID is returned, or there is no security environment established, no security profile is associated with the conversation and this parameter contains blanks.

Reason_code

Returned parameter

- Type: Integer
- Length: 32 bits

Contains additional information about the result of the call when the return_code parameter contains a non-zero value other than decimal 64 (atbcts_appc_not_available).

Table 13 on page 53 lists the valid reason codes.

Return_code

Returned parameter

- Type: Integer
- Length: 32 bits

Contains the result of the call. If the return_code parameter contains zero or decimal 64 (atbcts_appc_not_available), there is no reason code. For other return codes, check the reason_code parameter for additional information about the result of the call.

Table 13 on page 53 lists the valid return and reason codes for the Receive_Allocate service.

Table 13. Return and Reason Codes for Receive_Allocate

Return Code (Decimal)	Reason Code (Decimal)	Symbolic Value
0		atbcts_ok
8		atbcts_parameter_error
	17	atbcts_inval_alloc_queue_token
	18	atbcts_inval_notify_type
	19	atbcts_inval_timeout_value
	38	atbcts_inval_receive_allc_type
16		atbcts_request_unsuccessful
	7	atbcts_parameter_inaccessible
	8	atbcts_cannot_hold_locks
	20	atbcts_request_cancelled
	21	atbcts_no_alloc_to_receive
	44	atbcts_luwid_already_associated
	45	atbcts_sync_point_manager_error
32		atbcts_service_failure
	16	atbcts_appc_service_failure
64		atbcts_appc_not_available

For more detailed information about these return codes and reason codes, see Appendix B, “Explanation of Return and Reason Codes,” on page 79.

Abend Codes for Receive_Allocate

The caller might encounter abend X'EC7' with either of the reason codes shown below:

Table 14. Abend Codes for Receive_Allocate

Abend Code (Hexadecimal)	Reason Code (Hexadecimal)	Description
X'EC7'	X'00140003'	The number of parameters specified is incorrect.
X'EC7'	X'00140004'	APPC/MVS cannot access one or more of the specified parameters.

See *z/OS MVS System Codes* for an explanation and response for these codes.

Register_for_Allocates

An application program calls the Register_For_Allocates service to indicate which inbound allocate requests are to be directed to it, rather than scheduled by an APPC/MVS transaction scheduler. The program is considered to be an APPC/MVS server on the successful completion of the Register_For_Allocates service.

Environment for Register_For_Allocates

The requirements for the caller are:

Minimum authorization:	Problem state, with any PSW key
Dispatchable unit mode:	Task or SRB
Cross memory mode:	Any PASN, any HASN, any SASN
AMODE:	31-bit
ASC mode:	Primary or access register (AR)
Interrupt status:	Enabled for I/O and external interrupts
Locks:	No locks held
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Restrictions

Programs that call the Register_For_Allocates service while in task mode should not have any enabled unlocked task (EUT) functional recovery routines (FRRs) established. For more information about EUT FRRs, see the section on providing recovery in *z/OS MVS Programming: Authorized Assembler Services Guide*.

Input Register Information

Before calling the Register_For_Allocates service, the caller must ensure that the following GPRs contain the specified information:

Register
: **Contents**

1
: Address of the parameter list

13	Address of a standard 18-word save area
14	Return address
15	Entry point address of the service being called.

Before calling the Register_For_Allocates service, the caller must ensure that the following access registers (ARs) contain the specified information:

Register	**Contents**
1	0
13 - 15	0

Output Register Information

When control returns to the caller of the Register_For_Allocates service, the general purpose registers (GPRs) contain:

Register	**Contents**
0 - 1	Used as work registers by the system
2 - 14	Unchanged
15	Used as a work register by the system

When control returns to the caller of the Register_For_Allocates service, the access registers (ARs) contain:

Register	**Contents**
0 - 1	Used as work registers by the system
2 - 14	Unchanged
15	Used as a work register by the system

Some callers depend on register contents remaining the same before and after issuing a service. If the system changes the contents of registers on which the caller depends, the caller must save them before issuing the service, and restore them after the system returns control.

Syntax Format for Register_For_Allocates

The figure below shows the syntax of the CALL statement for the Register_For_Allocates service. You must code all parameters on the CALL statement in the order shown.

```
CALL ATBRFA2(
        Notify_type,
        Sym_dest_name,
        TP_name_length,
        TP_name,
        Local_LU_name,
        Partner_LU_name,
        User_ID,
        Profile,
        Allocate_Queue_Token,
        Reason_code,
        Return_code
       );
```

Figure 12. Syntax for Register_For_Allocates

Parameters for Register_For_Allocates

The following section describes the parameters you specify when calling the Register_For_Allocates service.

Notify_type
Supplied parameter

- Type: Structure
- Length: 4-8 bytes

Specifies the type of processing and notification (synchronous or asynchronous) requested for this service. The possible types are:

- None

  No notification is requested. APPC/MVS processes this call synchronously, and returns control to the caller when processing is complete. APPC/MVS sets all returned parameters on return to the caller. To specify no notification, set this parameter to a four-byte structure that contains binary zeroes.
- ECB

  Programs can request asynchronous processing by specifying an ECB to be posted when processing completes. To specify an ECB, set this parameter to an eight-byte structure that contains a fullword binary one (X'00000001'), followed by the address of a fullword area to be used as the ECB. The ECB must reside in the caller's home address space.

  When you specify an ECB, APPC/MVS returns control to the caller before processing is complete, with only the return code set. If APPC/MVS accepts the asynchronous request, it sets the return code to 0 to show that it is processing the service asynchronously. APPC/MVS fills in the other returned parameters during asynchronous processing, and posts the specified ECB when it has set all the returned parameters. The completion code field in the ECB contains the return code for the service. APPC/MVS places the reason code, if any, in the server's reason_code parameter.

Sym_dest_name
Supplied parameter

- Type: Character string
- Char Set: 01134
- Length: 8 bytes (left justified, padded with blanks)

Specifies a symbolic destination name that represents the transaction program and local LU for which you are registering. This parameter, if used, must contain a value that matches a symbolic destination name defined in the active side information data set. APPC/MVS obtains the TP name and local LU name from the side information data set (the mode name, which is also contained in the side information, is ignored).

To omit a symbolic destination name, set the sym_dest_name parameter value to 8 blanks and specify values for the local_LU_name, TP_name, and TP_name_length parameters.

If you specify a symbolic destination name, APPC/MVS obtains the local LU name from the PARTNER_LU keyword value in the symbolic destination entry. If the LU name specified in the symbolic destination entry is network-qualified (includes both the network ID and network LU name), APPC/MVS uses only the network-LU-name portion for the local LU name. If the keyword value is a VTAM generic resource name, this service fails with return code 16 (atbcts_request_unsuccessful) and reason code 12 (atbcts_inval_local_lu). If both the local_LU_name parameter and the LU name in the symbolic destination entry is blank, this service fails with return code 8 (atbcts_parameter_error), and reason code 5 (atbcts_local_lu_not_specified).

If the local LU name is obtained from the side information data set and the LU name is a network-qualified LU name, APPC/MVS ensures that the netid in the symbolic destination name is the local netid. If not, the Register_For_Allocates service returns return code 4 (atbcts_warning) and reason code 28 (atbcts_netid_does_not_match). If VTAM is not active and therefore APPC/MVS cannot determine whether the netid is valid, the Register_For_Allocates service returns return code 4 (atbcts_warning) and reason code 39 (atbcts_cannot_determine_netid). Both of these codes are warnings only; APPC/MVS continues to process the call to the Register_For_Allocates service.

If you also specify values for the local_LU_name or the TP_name/TP_name_length parameters, these values override any obtained from the side information data set.

If you specify a symbolic destination name that does not match an entry in the side information data set, the Register_For_Allocates service returns control with return code 16 (atbcts_request_unsuccessful) and reason code 11 (atbcts_sym_dest_name_unknown).

TP_name_length
Supplied parameter

- Type: Integer
- Length: 32 bits
- Range: 0-64

Specifies the length of data contained in the TP_name parameter.

If you specify a symbolic destination name in the sym_dest_name parameter, set TP_name_length to 0 to use the TP name from the side information data set.

TP_name
Supplied parameter

- Type: Character string
- Char Set: 00640 or Type A (Type A if the TP is protected by RACF)
- Length: 1 - 64 bytes

Specifies the name of transaction program that was targeted by the client TP's allocate request. This name must match the name that the client TP specified in the TP_name parameter of the Allocate service.

If you specify a symbolic destination name in the sym_dest_name parameter, set TP_name to 0 to use the TP name from the side information data set.

You can specify a SNA service transaction program name in this parameter.

If the TP is to be protected by a RACF security profile in the APPCTP class or the APPCSERV class, the TP name must consist of Type A characters only. See Appendix A, "Character Sets," on page 75 for a list of Type A characters.

Local_LU_name
Supplied parameter

- Type: Character string
- Char Set: Type A
- Length: 8 bytes (left justified, padded on right with blanks)

Specifies the name of the LU at which the transaction program specified in the TP_name parameter resides. The local LU name must match the name of the LU that the client TP specified on its allocate call (in the partner_LU_name parameter).

If you specify a symbolic destination name in the sym_dest_name parameter, set local_LU_name to 8 blanks to use the value for the PARTNER_LU keyword in the side information entry. The keyword value cannot be a VTAM generic resource name, or this service fails.

Partner_LU_name
Supplied parameter

- Type: Character string
- Char Set: Type A
- Length: 17 bytes (left justified, padded on right with blanks)

Specifies the name of the LU from which the client TP's allocate request originated. The client TP might have specified this LU on its allocate request, or have used the default local LU (by setting the local_LU_name parameter of the Allocate service to blanks). The local LU transforms this locally known LU name to an LU name used by the network. (For more information about the local_LU_name parameter of the Allocate service, see *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*.)

The partner_LU_name parameter can contain one of the following values:

- LU name only (1-8 byte Type A character string).

  This string represents the network LU name, which, if unique within the network and interconnected networks, is sufficient for most TP communications.

  Inbound allocate requests from any LU with the network LU name specified on the call to the Register_For_Allocates service can be placed on this allocate queue. APPC/MVS, however, will first check to see if there are any allocate queues with a matching network-qualified partner LU name.
- A VTAM generic resource name.

  If the partner LU is a member of a generic resource group, you may specify the 1- to 8-byte generic resource name of the group.
- Combined network ID and network LU name (two 1-8 byte Type A character strings, concatenated by a period: *network_ID.network_LU_name*). This format is known as a **network-qualified LU name**; each LU in the network and all interconnected networks can be uniquely identified by its network-qualified LU name.

  The network-LU-name portion of a network-qualified name may be a VTAM generic resource name.

  If the network-qualified LU name is specified, both the network ID and the network LU name on the inbound allocate request will have to match the value specified on the Register_For_Allocates call to be placed on the allocate queue. If VTAM is inactive, APPC/MVS is unable to determine the network ID, therefore no inbound allocate requests will be placed on an allocate queue that has a network-qualified partner LU name. When an inbound allocate request enters the system, APPC/MVS first checks for a match with the network-qualified LU name and then just the network LU name.
- Blank:

  A blank value for the partner_LU_name parameter indicates that allocate requests from any partner LU are to be accepted.

Table 15 on page 59 shows whether the partner LU name specified on a call to Register_For_Allocates will successfully match the LU associated with the inbound request:

Table 15. How APPC/MVS Handles Partner LU Name Specifications

Register_For_Allocates specifies...	**LU of the inbound request...**		
	netid.LUNAME	**net2.LUNAME**	**LUNAME**
LUNAME	Match	Match	Match
netid.LUNAME	Match	No Match	No Match
net2.LUNAME	No Match	Match	No Match

User_ID

Supplied parameter

- Type: Character string
- Char Set: Type A
- Length: 8 bytes (left justified, padded on right with blanks)

Specifies the user ID associated with the inbound allocate requests to be served.

A blank value for the user_ID parameter indicates that allocate requests from any user ID are to be accepted.

Note that APPC/MVS does not ensure that the specified user ID is a member of the security profile specified in the profile parameter. The server must perform this check itself, if needed.

Profile

Supplied parameter

- Type: Character string
- Char Set: Type A
- Length: 8 bytes (left justified, padded on right with blanks)

Specifies the security profile associated with the inbound allocate requests to be served. APPC/MVS treats the profile name as a RACF group name.

A blank value for the profile parameter indicates that allocate requests from any security profile are to be accepted.

APPC/MVS compares this profile with the profile that flows in with the allocate request, or with the user ID's default profile, if no profile flows in with the allocate request.

Allocate_queue_token

Returned parameter

- Type: Character string
- Length: 8 bytes

Returns the allocate queue token, which uniquely identifies an allocate queue. Use the allocate queue token on later calls to APPC/MVS allocate queue services to indicate the particular allocate queue on which a requested function is to be performed.

This field contains a valid token only when the return code from this service is either 0 or 4 (atbcts_warning).

Reason_code

Returned parameter

- Type: Integer
- Length: 32 bits

Contains additional information about the result of the call when the return_code parameter contains a non-zero value other than decimal 64 (atbcts_appc_not_available).

Table 16 on page 60 lists the valid reason codes.

Return_code

Returned parameter

- Type: Integer
- Length: 32 bits

Contains the result of the call. If the return_code parameter contains zero or decimal 64 (atbcts_appc_not_available), there is no reason code. For other return codes, check the reason_code parameter for additional information about the result of the call.

Table 16 on page 60 lists the valid return and reason codes for the Register_For_Allocates service.

Table 16. Return and Reason Codes for Register_For_Allocates

Return Code (Decimal)	**Reason Code (Decimal)**	**Symbolic Value**
0		atbcts_ok
4		atbcts_warning
	1	atbcts_already_registered
	28	atbcts_netid_does_not_match
	39	atbcts_cannot_determine_netid
8		atbcts_parameter_error
	2	atbcts_tp_name_not_specified
	3	atbcts_inval_tp_name
	4	atbcts_inval_tp_name_length
	5	atbcts_local_lu_not_specified
	18	atbcts_inval_notify_type
	43	atbcts_inval_partner_lu
16		atbcts_request_unsuccessful
	7	atbcts_parameter_inaccessible
	8	atbcts_cannot_hold_locks
	10	atbcts_sched_cant_register
	11	atbcts_sym_dest_name_unknown
	12	atbcts_inval_local_lu
	13	atbcts_lu_not_receiving
	14	atbcts_not_auth_to_serve_tp
	15	atbcts_not_auth_to_local_lu
	20	atbcts_request_cancelled
32		atbcts_service_failure
	16	atbcts_appc_service_failure
64		atbcts_appc_not_available

For more detailed information about these return and reason codes, see Appendix B, "Explanation of Return and Reason Codes," on page 79.

Abend Codes for Register_For_Allocates

The caller might encounter abend X'EC7' with either of the reason codes shown below:

Table 17. Abend Codes for Register_For_Allocates

Abend Code (Hexadecimal)	Reason Code (Hexadecimal)	Meaning
X'EC7'	X'00140001'	The number of parameters specified is incorrect.
X'EC7'	X'00140002'	APPC/MVS cannot access one or more of the specified parameters.

See *z/OS MVS System Codes* for an explanation and response for these codes.

Set_Allocate_Queue_Attributes

APPC/MVS servers use the Set_Allocate_Queue_Attributes service to specify whether APPC/MVS is to preserve an allocate queue during periods of time when no servers are registered for the queue. This service allows the caller to stop serving an allocate queue for some interval of time, and then resume serving the queue without interrupting the flow of inbound allocate requests.

Environment for Set_Allocate_Queue_Attributes

The requirements for the caller are:

Minimum authorization:	Problem state, with any PSW key
Dispatchable unit mode:	Task or SRB
Cross memory mode:	Any PASN, any HASN, any SASN
AMODE:	31-bit
ASC mode:	Primary or access register (AR)
Interrupt status:	Enabled for I/O and external interrupts
Locks:	No locks held
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Restrictions

Programs that call the Set_Allocate_Queue_Attributes service while in task mode should not have any enabled unlocked task (EUT) functional recovery routines (FRRs) established. For more information about EUT FRRs, see the section on providing recovery in *z/OS MVS Programming: Authorized Assembler Services Guide*.

Input Register Information

Before calling the Set_Allocate_Queue_Attributes service, the caller must ensure that the following GPRs contain the specified information:

Register
  Contents

1
Address of the parameter list

13
Address of a standard 18-word save area

14
Return address

15
Entry point address of the service being called.

Before calling the Set_Allocate_Queue_Attributes service, the caller must ensure that the following access registers (ARs) contain the specified information:

Register	Contents
1	0
13 - 15	0

Output Register Information

When control returns to the caller of the Set_Allocate_Queue_Attributes service, the general purpose registers (GPRs) contain:

Register	Contents
0 - 1	Used as work registers by the system
2 - 14	Unchanged
15	Used as a work register by the system

When control returns to the caller of the Set_Allocate_Queue_Attributes service, the access registers (ARs) contain:

Register	Contents
0 - 1	Used as work registers by the system
2 - 14	Unchanged
15	Used as a work register by the system

Some callers depend on register contents remaining the same before and after issuing a service. If the system changes the contents of registers on which the caller depends, the caller must save them before issuing the service, and restore them after the system returns control.

Syntax Format for Set_Allocate_Queue_Attributes

The figure below shows the syntax of the CALL statement for the Set_Allocate_Queue_Attributes service. You must code all parameters on the CALL statement in the order shown.

```
CALL ATBSAQ2(
        Notify_type,
        Allocate_queue_token,
        Allocate_queue_keep_time,
        Reason_code,
        Return_code
       );
```

Figure 13. Syntax for Set_Allocate_Queue_Attributes

Parameters for Set_Allocate_Queue_Attributes

The following section describes the parameters you specify when calling the Set_Allocate_Queue_Attributes service.

Notify_type

Supplied parameter

- Type: Structure
- Length: 4-8 bytes

Specifies the type of processing and notification (synchronous or asynchronous) requested for this service. The possible types are:

- None

  No notification is requested. APPC/MVS processes this call synchronously, and returns control to the caller when processing is complete. APPC/MVS sets all returned parameters on return to the caller. To specify no notification, set this parameter to a four-byte structure that contains binary zeroes.

- ECB

  Programs can request asynchronous processing by specifying an ECB to be posted when processing completes. To specify an ECB, set this parameter to an eight-byte structure that contains a fullword binary one (X'00000001'), followed by the address of a fullword area to be used as the ECB. The ECB must reside in the caller's home address space.

  When you specify an ECB, APPC/MVS returns control to the caller before processing is complete, with only the return code set. If APPC/MVS accepts the asynchronous request, it sets the return code to 0 to show that it is processing the service asynchronously. APPC/MVS fills in the other returned parameters during asynchronous processing, and posts the specified ECB when it has set all the returned parameters. The completion code field in the ECB contains the return code for the service. APPC/MVS places the reason code, if any, in the server's reason_code parameter.

Allocate_queue_token

Supplied parameter

- Type: Character string
- Length: 8 bytes

Specifies the allocate queue token that indicates the particular allocate queue for which the caller wants to set attributes. (The allocate queue token is returned as output from a successful call to the Register_For_Allocates service.)

Allocate_queue_keep_time

Supplied parameter

- Type: Integer
- Length: 32 bits

- Range: 0 - 3600

Specifies, in seconds, the amount of time APPC/MVS is to maintain the specified allocate queue during periods when no servers are registered for the queue. This time takes effect when the last server of the allocate queue calls the Unregister_For_Allocates service to stop serving the queue. If no server registers for the allocate queue before this time limit is exceeded, APPC/MVS rejects any allocate requests that reside on the allocate queue.

To have no keep time in effect for the allocate queue, set this parameter to zero (or do not call this service).

If you specify an allocate queue keep time that is outside the accepted range (0 to 3600 seconds), the Set_Allocate_Queue_Attributes service returns control with the reason_code parameter set to 8 (atbcts_parameter_error) and the reason_code parameter set to 34 (atbcts_inval_queue_keep_time).

Reason_code
Returned parameter

- Type: Integer
- Length: 32 bits

Contains additional information about the result of the call when the return_code parameter contains a non-zero value other than decimal 64 (atbcts_appc_not_available).

Table 18 on page 64 lists the valid reason codes.

Return_code
Returned parameter

- Type: Integer
- Length: 32 bits

Contains the result of the call. If the return_code parameter contains zero or decimal 64 (atbcts_appc_not_available), there is no reason code. For other return codes, check the reason_code parameter for additional information about the result of the call.

Table 18 on page 64 lists the valid return and reason codes for the Set_Allocate_Queue_Attributes service.

Table 18. Return and Reason Codes for Set_Allocate_Queue_Attributes

Return Code (Decimal)	Reason Code (Decimal)	Symbolic Value
0		atbcts_ok
8		atbcts_parameter_error
	17	atbcts_inval_alloc_queue_token
	18	atbcts_inval_notify_type
	34	atbcts_inval_queue_keep_time
16		atbcts_request_unsuccessful
	8	atbcts_cannot_hold_locks
	20	atbcts_request_cancelled
32		atbcts_service_failure
	16	atbcts_appc_service_failure
64		atbcts_appc_not_available

For more detailed information about these return codes and reason codes, see Appendix B, “Explanation of Return and Reason Codes,” on page 79.

Abend Codes for Set_Allocate_Queue_Attributes

The caller might encounter abend X'EC7' with either of the reason codes shown below:

Table 19. Abend Codes for Set_Allocate_Queue_Attributes

Abend Code (Hexadecimal)	**Reason Code (Hexadecimal)**	**Description**
X'EC7'	X'0014000D'	The number of parameters specified is incorrect.
X'EC7'	X'0014000E'	APPC/MVS cannot access one or more of the specified parameters.

See *z/OS MVS System Codes* for an explanation and response for these codes.

Set_Allocate_Queue_Notification

APPC/MVS servers use the Set_Allocate_Queue_Notification service to request to be notified when an allocate queue reaches a specified maximum or minimum number of inbound allocate requests (or to cancel a previous request for such notification).

Environment for Set_Allocate_Queue_Notification

The requirements for the caller are:

Minimum authorization:	Problem state, with any PSW key
Dispatchable unit mode:	Task or SRB
Cross memory mode:	Any PASN, any HASN, any SASN
AMODE:	31-bit
ASC mode:	Primary or access register (AR)
Interrupt status:	Enabled for I/O and external interrupts
Locks:	No locks held
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Restrictions

Programs that call the Set_Allocate_Queue_Notification service while in task mode should not have any enabled unlocked task (EUT) functional recovery routines (FRRs) established. For more information about EUT FRRs, see the section on providing recovery in *z/OS MVS Programming: Authorized Assembler Services Guide*.

Input Register Information

Before calling the Set_Allocate_Queue_Notification service, the caller must ensure that the following GPRs contain the specified information:

Register
    Contents

1
    Address of the parameter list

13
    Address of a standard 18-word save area

14
Return address

15
Entry point address of the service being called.

Before calling the Set_Allocate_Queue_Notification service, the caller must ensure that the following access registers (ARs) contain the specified information:

Register	**Contents**
1	0
13 - 15	0

Output Register Information

When control returns to the caller of the Set_Allocate_Queue_Notification service, the general purpose registers (GPRs) contain:

Register	**Contents**
0 - 1	Used as work registers by the system
2 - 14	Unchanged
15	Used as a work register by the system

When control returns to the caller of the Set_Allocate_Queue_Notification service, the access registers (ARs) contain:

Register	**Contents**
0 - 1	Used as work registers by the system
2 - 14	Unchanged
15	Used as a work register by the system

Some callers depend on register contents remaining the same before and after issuing a service. If the system changes the contents of registers on which the caller depends, the caller must save them before issuing the service, and restore them after the system returns control.

Syntax Format for Set_Allocate_Queue_Notification

The figure below shows the syntax of the CALL statement for the Set_Allocate_Queue_Notification service. You must code all parameters on the CALL statement in the order shown.

```
CALL ATBSAN2(
        Notify_type,
        Allocate_queue_token,
        Event_notification_type,
        Event_code,
        Event_qualifier,
        Reason_code,
        Return_code
       );
```

Figure 14. Syntax for Set_Allocate_Queue_Notification

Parameters for Set_Allocate_Queue_Notification

The following section describes the parameters you specify when calling the Set_Allocate_Queue_Notification service.

Notify_type
Supplied parameter

- Type: Structure
- Length: 4-8 bytes

Specifies the type of processing and notification (synchronous or asynchronous) requested for this service. The possible types are:

- None

  No notification is requested. APPC/MVS processes this call synchronously, and returns control to the caller when processing is complete. APPC/MVS sets all returned parameters on return to the caller. To specify no notification, set this parameter to a four-byte structure that contains binary zeroes.
- ECB

  Programs can request asynchronous processing by specifying an ECB to be posted when processing completes. To specify an ECB, set this parameter to an eight-byte structure that contains a fullword binary one (X'00000001'), followed by the address of a fullword area to be used as the ECB. The ECB must reside in the caller's home address space.

  When you specify an ECB, APPC/MVS returns control to the caller before processing is complete, with only the return code set. If APPC/MVS accepts the asynchronous request, it sets the return code to 0 to show that it is processing the service asynchronously. APPC/MVS fills in the other returned parameters during asynchronous processing, and posts the specified ECB when it has set all the returned parameters. The completion code field in the ECB contains the return code for the service. APPC/MVS places the reason code, if any, in the server's reason_code parameter.

Allocate_queue_token
Supplied parameter

- Type: Character string
- Length: 8 bytes

Specifies the allocate queue token that indicates the particular allocate queue for which you are requesting notification, or cancelling a previous notification request. (The allocate queue token is returned as output from a successful call to the Register_For_Allocates service.)

Event_notification_type
Supplied parameter

- Type: Integer
- Length: 32 bits

Specifies one-time or continuous notification, or cancels notification that the server requested previously through one or more calls to this service.

Valid values for this parameter are:

Value
Meaning

1
atbcts_set_one_time_notify

Request one-time notification for the event. APPC/MVS notifies the server after the first occurrence of the event and stops monitoring for the event. The server can restore this notification request if necessary by calling the Set_Allocate_Queue_Notification service again with the same parameter values.

2
: atbcts_set_continuous_notify

Request continuous notification for an event. APPC/MVS notifies the server every time the event occurs until the server cancels this request or the APPC/MVS address space ends.

3
: atbcts_cancel_notify

Cancel a particular notification request (specified in this call by the combination of the event_code and event_notification_type parameters). APPC/MVS also deletes from the server's event queue any unreceived event elements that were queued because of the request to be cancelled. If this is the last active notification request for the server and the server has an outstanding call to the Get_Event service, APPC/MVS cancels the Get_Event service. The server receives, as output from the Get_Event service, a return code of 16 (atbcts_request_unsuccessful) and a reason code of 31 (atbcts_event_notify_cancelled).

4
: atbcts_cancel_all_notify

Cancel *all* active event notification requests from the server for a particular allocate queue (specified by the allocate_queue_token parameter). APPC/MVS also deletes any unreceived event elements related to the allocate queue from the server's event queue. If this is the server's last active notification request, and the server has an outstanding call to the Get_Event service, APPC/MVS cancels the call to the Get_Event service. The server receives, as output from the Get_Event service, a return code of 16 (atbcts_request_unsuccessful) and a reason code of 31 (atbcts_event_notify_cancelled).

Event_code

Supplied parameter

- Type: Integer
- Length: 32 bits

Specifies the type of threshold (minimum or maximum) for which the server is to be notified, or the notification request to be cancelled. When the event_notification_type parameter is set to 1 or 2, this is the type of threshold for which the server is requesting notification. When the event_notification_type parameter is set to 3 or 4, this is the type of threshold for which the server is cancelling notification.

Valid values for this parameter are:

Value
: **Meaning**

1
: atbcts_allocate_queue_min

Specifies a minimum threshold. For a notification request (event notification type 1 or 2), APPC/MVS is to notify the server when the allocate queue (specified by the allocate_queue_token parameter) decreases to the number of allocate requests specified by the event_qualifier parameter.

For a cancel notification request (event notification type 3 or 4), APPC/MVS is to cancel notification for this minimum threshold.

2
: atbcts_allocate_queue_max

Specifies a maximum threshold. For a notification request (event notification type 1 or 2), APPC/MVS is to notify the server when the allocate queue (specified by the allocate_queue_token parameter) increases to the number of allocate requests specified by the event_qualifier parameter.

For a cancel notification request (event notification type 3 or 4), APPC/MVS is to cancel notification for this maximum threshold.

Event_qualifier
Supplied parameter

- Type: Integer
- Length: 32 bits

Specifies the number (in decimal) of allocate requests for the minimum or maximum threshold. The range of possible values for this parameter depends on whether the threshold is a minimum or maximum (specified through the event_code parameter). The threshold ranges are as follows:

- For a minimum threshold, this parameter may be set to a value from 0 to ((2**32)-2)
- For a maximum threshold, this parameter may be set to a value from 1 to ((2**32)-1).

Reason_code
Returned parameter

- Type: Integer
- Length: 32 bits

Contains additional information about the result of the call when the return_code parameter contains a non-zero value other than decimal 64 (atbcts_appc_not_available).

Table 20 on page 69 lists the valid reason codes.

Return_code
Returned parameter

- Type: Integer
- Length: 32 bits

Contains the result of the call. If the return_code parameter contains zero or decimal 64 (atbcts_appc_not_available), there is no reason code. For other return codes, check the reason_code parameter for additional information about the result of the call.

Table 20 on page 69 lists the valid return and reason codes for the Set_Allocate_Queue_Notification service.

Table 20. Return and Reason Codes for Set_Allocate_Queue_Notification

Return Code (Decimal)	**Reason Code (Decimal)**	**Symbolic Value**
0		atbcts_ok
8		atbcts_parameter_error
	17	atbcts_inval_alloc_queue_token
	18	atbcts_inval_notify_type
	26	atbcts_inval_event_notif_type
	27	atbcts_inval_event_code
	29	atbcts_inval_event_code_qual
16		atbcts_request_unsuccessful
	8	atbcts_cannot_hold_locks
	20	atbcts_request_cancelled

Table 20. Return and Reason Codes for Set_Allocate_Queue_Notification (continued)

Return Code (Decimal)	Reason Code (Decimal)	Symbolic Value
32		atbcts_service_failure
	16	atbcts_appc_service_failure
64		atbcts_appc_not_available

For more detailed information about these return codes and reason codes, see Appendix B, “Explanation of Return and Reason Codes,” on page 79.

Abend Codes for Set_Allocate_Queue_Notification

The caller might encounter abend X'EC7' with either of the reason codes shown below:

Table 21. Abend Codes for Set_Allocate_Queue_Notification

Abend Code (Hexadecimal)	Reason Code (Hexadecimal)	Description
X'EC7'	X'00140007'	The number of parameters specified is incorrect.
X'EC7'	X'00140008'	APPC/MVS cannot access one or more of the specified parameters.

See *z/OS MVS System Codes* for an explanation and response for these codes.

Unregister_For_Allocates

The Unregister_For_Allocates service is used by a server to specify that it will no longer serve specific inbound transaction program requests. The service indicates that the server is no longer registered for an allocate queue and therefore the allocate queue token is no longer valid for the server.

Environment for Unregister_For_Allocates

The requirements for the caller are:

Minimum authorization:	Problem state, with any PSW key
Dispatchable unit mode:	Task or SRB
Cross memory mode:	Any PASN, any HASN, any SASN
AMODE:	31-bit
ASC mode:	Primary or access register (AR)
Interrupt status:	Enabled for I/O and external interrupts
Locks:	No locks held
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Restrictions

Programs that call the Unregister_For_Allocates service while in task mode should not have any enabled unlocked task (EUT) functional recovery routines (FRRs) established. For more information about EUT FRRs, see the section on providing recovery in *z/OS MVS Programming: Authorized Assembler Services Guide*.

Input Register Information

Before calling the Unregister_For_Allocates service, the caller must ensure that the following GPRs contain the specified information:

Register
: **Contents**

1
: Address of the parameter list

13
: Address of a standard 18-word save area

14
: Return address

15
: Entry point address of the service being called.

Before calling the Unregister_For_Allocates service, the caller must ensure that the following access registers (ARs) contain the specified information:

Register
: **Contents**

1
: 0

13 - 15
: 0

Output Register Information

When control returns to the caller of the Unregister_For_Allocates service, the general purpose registers (GPRs) contain:

Register
: **Contents**

0 - 1
: Used as work registers by the system

2 - 14
: Unchanged

15
: Used as a work register by the system

When control returns to the caller of the Unregister_For_Allocates service, the access registers (ARs) contain:

Register
: **Contents**

0 - 1
: Used as work registers by the system

2 - 14
: Unchanged

15
: Used as a work register by the system

Some callers depend on register contents remaining the same before and after issuing a service. If the system changes the contents of registers on which the caller depends, the caller must save them before issuing the service, and restore them after the system returns control.

Syntax Format for Unregister_For_Allocates

The figure below shows the syntax of the CALL statement for the Unregister_For_Allocates service. You must code all parameters on the CALL statement in the order shown.

```
CALL ATBURA2(
        Notify_type,
        Allocate_queue_token,
        Reason_code,
        Return_code
       );
```

Figure 15. Syntax for Unregister_For_Allocates

Parameters for Unregister_For_Allocates

The following section describes the parameters you specify when calling the Unregister_For_Allocates service.

Notify_type

Supplied parameter

- Type: Structure
- Length: 4-8 bytes

Specifies the type of processing and notification (synchronous or asynchronous) requested for this service. The possible types are:

- None

  No notification is requested. APPC/MVS processes this call synchronously, and returns control to the caller when processing is complete. APPC/MVS sets all returned parameters on return to the caller. To specify no notification, set this parameter to a four-byte structure that contains binary zeroes.

- ECB

  Programs can request asynchronous processing by specifying an ECB to be posted when processing completes. To specify an ECB, set this parameter to an eight-byte structure that contains a fullword binary one (X'00000001'), followed by the address of a fullword area to be used as the ECB. The ECB must reside in the caller's home address space.

  When you specify an ECB, APPC/MVS returns control to the caller before processing is complete, with only the return code set. If APPC/MVS accepts the asynchronous request, it sets the return code to 0 to show that it is processing the service asynchronously. APPC/MVS fills in the other returned parameters during asynchronous processing, and posts the specified ECB when it has set all the returned parameters. The completion code field in the ECB contains the return code for the service. APPC/MVS places the reason code, if any, in the server's reason_code parameter.

Allocate_queue_token

Supplied parameter

- Type: Character string
- Length: 8 bytes

Specifies the allocate queue token that indicates the particular allocate queue that the caller is to stop serving. (The allocate queue token is returned as output from a successful call to the Register_For_Allocates service.)

To unregister the caller for all allocate queues for which it is registered, set this parameter to binary zeroes.

Reason_code

Returned parameter

- Type: Integer

- Length: 32 bits

Contains additional information about the result of the call when the return_code parameter contains a non-zero value other than decimal 64 (atbcts_appc_not_available).

Table 22 on page 73 lists the valid reason codes.

Return_code
Returned parameter

- Type: Integer
- Length: 32 bits

Contains the result of the call. If the return_code parameter contains zero or decimal 64 (atbcts_appc_not_available), there is no reason code. For other return codes, check the reason_code parameter for additional information about the result of the call.

Table 22 on page 73 lists the valid return and reason codes for the Unregister_For_Allocates service.

Table 22. Return and Reason Codes for Unregister_For_Allocates

Return Code (Decimal)	Reason Code (Decimal)	Symbolic Value
0		atbcts_ok
4		atbcts_warning
	36	atbcts_unreg_all_no_registers
8		atbcts_parameter_error
	17	atbcts_inval_alloc_queue_token
	18	atbcts_inval_notify_type
16		atbcts_request_unsuccessful
	8	atbcts_cannot_hold_locks
	20	atbcts_request_cancelled
32		atbcts_service_failure
	16	atbcts_appc_service_failure
64		atbcts_appc_not_available

For more detailed information about these return codes and reason codes, see Appendix B, "Explanation of Return and Reason Codes," on page 79.

Abend Codes for Unregister_For_Allocates

The caller might encounter abend X'EC7' with either of the reason codes shown below:

Table 23. Abend Codes for Unregister_For_Allocates

Abend Code (Hexadecimal)	Reason Code (Hexadecimal)	Description
X'EC7'	X'00140011'	The number of parameters specified is incorrect.
X'EC7'	X'00140012'	APPC/MVS cannot access one or more of the specified parameters.

See *z/OS MVS System Codes* for an explanation and response for these codes.

Appendix A. Character Sets

APPC/MVS makes use of character strings composed of characters from one of the following character sets:

- Character set 01134, which is composed of the uppercase letters A through Z and numerals 0-9.
- Character set Type A, which is composed of the uppercase letters A through Z, numerals 0-9, national characters (@, $, #), and must begin with either an alphabetic or a national character.
- Character set 00640, which is composed of the uppercase and lowercase letters A through Z, numerals 0-9, and 19 special characters. Note that APPC/MVS does not allow blanks in 00640 character strings.

These character sets, along with hexadecimal and graphic representations, are provided in the following table:

Table 24. Character Sets 01134, Type A, and 00640

Hex Code	Graphic	Description	Character Set		
			01134	Type A	00640
40		Blank			
4B	.	Period			X
4C	<	Less than sign			X
4D	(	Left parenthesis			X
4E	+	Plus sign			X
50	&	Ampersand			X
5B	$	Dollar sign		X (Note 1)	
5C	*	Asterisk			X (Note 2)
5D	)	Right parenthesis			X
5E	;	Semicolon			X
60	–	Dash			X
61	/	Slash			X
6B	,	Comma			X (Note 3)
6C	%	Percent sign			X
6D	_	Underscore			X
6E	>	Greater than sign			X
6F	?	Question mark			X
7A	:	Colon			X
7B	#	Pound sign		X (Note 1)	
7C	@	At sign		X (Note 1)	
7D	'	Single quote			X
7E	=	Equals sign			X
7F	"	Double quote			X
81	a	Lowercase a			X

Table 24. Character Sets 01134, Type A, and 00640 (continued)

Hex Code	Graphic	Description	Character Set		
			01134	**Type A**	**00640**
82	b	Lowercase b			X
83	c	Lowercase c			X
84	d	Lowercase d			X
85	e	Lowercase e			X
86	f	Lowercase f			X
87	g	Lowercase g			X
88	h	Lowercase h			X
89	i	Lowercase i			X
91	j	Lowercase j			X
92	k	Lowercase k			X
93	l	Lowercase l			X
94	m	Lowercase m			X
95	n	Lowercase n			X
96	o	Lowercase o			X
97	p	Lowercase p			X
98	q	Lowercase q			X
99	r	Lowercase r			X
A2	s	Lowercase s			X
A3	t	Lowercase t			X
A4	u	Lowercase u			X
A5	v	Lowercase v			X
A6	w	Lowercase w			X
A7	x	Lowercase x			X
A8	y	Lowercase y			X
A9	z	Lowercase z			X
C1	A	Uppercase A	X	X	X
C2	B	Uppercase B	X	X	X
C3	C	Uppercase C	X	X	X
C4	D	Uppercase D	X	X	X
C5	E	Uppercase E	X	X	X
C6	F	Uppercase F	X	X	X
C7	G	Uppercase G	X	X	X
C8	H	Uppercase H	X	X	X
C9	I	Uppercase I	X	X	X
D1	J	Uppercase J	X	X	X

Table 24. Character Sets 01134, Type A, and 00640 (continued)

Hex Code	Graphic	Description	Character Set		
			01134	Type A	00640
D2	K	Uppercase K	X	X	X
D3	L	Uppercase L	X	X	X
D4	M	Uppercase M	X	X	X
D5	N	Uppercase N	X	X	X
D6	O	Uppercase O	X	X	X
D7	P	Uppercase P	X	X	X
D8	Q	Uppercase Q	X	X	X
D9	R	Uppercase R	X	X	X
E2	S	Uppercase S	X	X	X
E3	T	Uppercase T	X	X	X
E4	U	Uppercase U	X	X	X
E5	V	Uppercase V	X	X	X
E6	W	Uppercase W	X	X	X
E7	X	Uppercase X	X	X	X
E8	Y	Uppercase Y	X	X	X
E9	Z	Uppercase Z	X	X	X
F0	0	Zero	X	X	X
F1	1	One	X	X	X
F2	2	Two	X	X	X
F3	3	Three	X	X	X
F4	4	Four	X	X	X
F5	5	Five	X	X	X
F6	6	Six	X	X	X
F7	7	Seven	X	X	X
F8	8	Eight	X	X	X
F9	9	Nine	X	X	X

Note:

1. Avoid these characters because they display differently depending on the national language code page in use.
2. Avoid using the asterisk in TP names because it causes a subset list request when entered on panels of the APPC administration dialog and in DISPLAY APPC commands.
3. Avoid using the comma in TP names because it acts as a parameter delimiter when entered in DISPLAY APPC commands.

Appendix B. Explanation of Return and Reason Codes

Refer to the following sections for diagnostic information related to the APPC/MVS allocate queue services:

- "Return Codes" on page 79
- "Reason Codes" on page 80
- "Symptom Records for APPC Service Failures" on page 83

Return Codes

The following table lists the possible return codes, their symbolic equates, and their meanings, for the APPC/MVS allocate queue services.

Table 25. Return Codes and Their Meanings

Return Code (Decimal)	Symbolic	Meaning
0	atbcts_ok	The service completed as requested.
4	atbcts_warning	The service completed but possibly not as expected. See the reason_code parameter for a description of the warning condition.
8	atbcts_parameter_error	A user-supplied parameter was found to be in error. For example, a parameter contains characters not in the required character set. See the reason_code parameter to determine which parameter is in error.
16	atbcts_request_unsuccessful	The service was unsuccessful. The cause is most likely a parameter error other than a syntax error, or an environmental error. For example, a syntactically valid LU name was specified, but the LU is not defined to APPC/MVS. An example of an environmental error is that the caller called the service while holding locks. See the reason_code parameter for the specific cause of the error, and to determine whether the error can be corrected and the service re-issued.
32	atbcts_service_failure	APPC/MVS service failure. Record the return and reason code, and give them to your systems programmer, who should contact the appropriate IBM support personnel.
64	atbcts_appc_not_available	APPC/MVS is not currently active. Call the service again after APPC is available.

Reason Codes

The following table lists the possible reason codes for the APPC/MVS allocate queue services. Because all reason code values are unique, a reason code alone is sufficient to identify an error condition.

Table 26. Reason Codes and Their Meanings

Reason Code (Decimal)	Symbolic	Meaning
1	atbcts_already_registered	The address space issued a Register_For_Allocates call that duplicated a previous Register_For_Allocates call (that is, the values specified for TP name, local LU name, partner LU name, user ID, and profile all matched those specified on a previous call to the Register_For_Allocates service).
2	atbcts_tp_name_not_specified	A TP name is required, but none was specified.
3	atbcts_inval_tp_name	The specified TP name contains characters that are not valid.
4	atbcts_inval_tp_name_length	The specified TP name length is outside the allowable range.
5	atbcts_local_lu_not_specified	A local LU name is required, but none was specified.
7	atbcts_parameter_inaccessible	An asynchronous call failed because a specified parameter was found to be inaccessible.
8	atbcts_cannot_hold_locks	The caller held one or more locks when calling the service.
10	atbcts_sched_cant_register	A transaction scheduler called the Register_For_Allocates service, which is not allowed.
11	atbcts_sym_dest_name_unknown	The specified symbolic destination name could not be found in the side information data set.
12	atbcts_inval_local_lu	Either the specified local LU is undefined, or the VTAM generic resource name for the local LU was specified on the Register_For_Allocates call or in a side information entry.
13	atbcts_lu_not_receiving	The specified local LU is not receiving inbound allocate requests.
14	atbcts_not_auth_to_serve_tp	The Register_For_Allocates service was called, but the caller is not authorized to serve the specified TP name on the specified local LU.

Table 26. Reason Codes and Their Meanings (continued)

Reason Code (Decimal)	**Symbolic**	**Meaning**
15	atbcts_not_auth_to_local_lu	The specified local LU is inaccessible to the caller. For a discussion of which LUs a server can use, refer to "For Which Local LUs Can a Server Register?" on page 12.
16	atbcts_appc_service_failure	The service failed because of an APPC failure. APPC provides symptom records for this type of error. For more information, see "Symptom Records for APPC Service Failures" on page 83.
17	atbcts_inval_alloc_queue_token	The specified allocate queue token does not represent an allocate queue for which this address space is registered.
18	atbcts_inval_notify_type	The specified notify type is not valid.
19	atbcts_inval_timeout_value	The specified timeout value is not valid.
20	atbcts_request_cancelled	The request was cancelled while in progress. This could have been caused by a call to the Unregister_For_Allocates service, or the termination of the caller's address space.
21	atbcts_no_alloc_to_receive	A Receive_Allocate call completed, but no allocate request was available to be received.
26	atbcts_inval_event_notif_type	The specified event notification type is not valid.
27	atbcts_inval_event_code	The specified event code is not supported or is not valid for this service.
28	atbcts_netid_does_not_match	The netid retrieved from the side information data set does not match the local netid.
29	atbcts_inval_event_code_qual	The specified event code qualifier is not valid or supported.
30	atbcts_no_event_available	The Get_Event call completed, but no event element was available to be received.
31	atbcts_event_notify_cancelled	The call to the Get_Event service was interrupted because all event notification requests were cancelled for this address space.
32	atbcts_get_event_outstanding	The call to the Get_Event service was rejected because a previous Get_Event call is currently outstanding.

Table 26. Reason Codes and Their Meanings (continued)

Reason Code (Decimal)	Symbolic	Meaning
33	atbcts_notify_not_set	The Get_Event call was rejected because no event notification is in effect for this address space.
34	atbcts_inval_queue_keep_time	The specified allocate queue keep time is outside the allowable range.
36	atbcts_unreg_all_no_registers	A call to the Unregister_For_Allocates service specified "unregister all" (that is, the allocate_queue_token was set to binary zeroes), but this address space is not registered for any allocate queues.
37	atbcts_inval_event_get_type	The specified event get type is not valid.
38	atbcts_inval_receive_allc_type	The specified receive allocate type is not valid.
39	atbcts_cannot_determine_netid	APPC/MVS cannot determine if the specified netid is valid.
41	atbcts_buffer_too_small	The service failed because the supplied buffer was not large enough to contain the requested information.
43	atbcts_inval_partner_lu	The service failed because the supplied partner LU name is not valid.
44	atbcts_luwid_already_associated	A Receive_Allocate service failed because a protected conversation (a conversation with a synchronization level of syncpt) and a logical unit of work identifier (LUWID) were already associated with the context for the dispatchable unit of work that issued the service call. APPC/MVS abnormally terminates the inbound conversation that the Receive_Allocate service tried to process. On the next conversation service it calls, the program that tried to allocate the conversation will receive a return code indicating TP_not_available_retry.
45	atbcts_sync_point_manager_error	A Receive_Allocate service failed because APPC/MVS could not register an inbound protected conversation (a conversation with a synchronization level of syncpt) as a protected logical unit of work with the system syncpoint manager (RRS). APPC/MVS abnormally terminates the inbound conversation that the Receive_Allocate service tried to process. On the next conversation service it calls, the program that tried to allocate the conversation will receive a return code indicating TP_not_available_retry.

Symptom Records for APPC Service Failures

If your program encounters return code 32 (atbcts_service_failure), and reason code 16 (atbcts_appc_service_failure), an APPC service failure has been detected. The system writes symptom records that describe the error to the logrec data set. See *z/OS MVS Diagnosis: Tools and Service Aids* for more information on retrieving and reading the logrec data set software record reports.

Section 3 of the symptom record contains the primary symptom string for APPC service failures:

Table 27. Symptom String for Service Failure Errors. (Section 3 of the Symptom Record in the Logrec Data Set.)

Symptom	Description
PIDS/5752SCACB	Product identifier
RIDS/ATBxxxxx	CSECT name
RIDS/ATBxxxxx#L	Load module name
LVLS/ddd	Product level
PCSS/ATBxxxx	The allocate queue service that caused the error. This field is omitted if the error was not caused by an allocate queue service.
PRCS/dddddddd	The return code returned to the caller of the service. This field is omitted if the error was not caused by an allocate queue service.
FLDS/REASON VALU/Hdddddddd	The unique reason code that identifies the APPC service failure.

Section 5 of the symptom record contains the following information for the APPC service failure:

- The job or user name, in EBCDIC, for the home address space of the caller
- An EBCDIC description of the error (up to 80 characters).

Look for symptom FLDS/REASON VALU/Hdddddddd in section 3 of the symptom record for the reason code identifying the error, which is one of the following:

Table 28. Reason Codes for Service-Failure Errors

Reason Code	Message Text	Explanation
00000006	ERROR IN TRYING TO VERIFY APPCSERV AUTHORITY.	An error occurred when APPC/MVS tried to verify the APPCSERV authority of the caller. This error occurred while APPC was processing a call to the Register_For_Allocates service.

Table 28. Reason Codes for Service-Failure Errors (continued)

Reason Code	Message Text	Explanation
00000007	ERROR RETRIEVING SECURITY INFORMATION.	An error occurred when APPC/MVS tried to obtain information about the caller's security environment. This error occurred while APPC was processing a call to the Register_For_Allocates service. The RACF return code and reason code appear in section 5 of the symptom record. **System Programmer Response:** Ensure that the correct level of the security product is installed and is active. Also check security-related parameters specified on the Register_For_Allocates call (such as user_ID and profile) for proper authorization.
00000008	ERROR RETRIEVING SIDE INFORMATION.	An error occurred when APPC/MVS tried to obtain side information from the side information file. This error occurred while APPC was processing a call to the Register_For_Allocates service.
00000009	AN INTERNAL FAILURE OCCURRED IN APPC PROCESSING.	An internal failure occurred during APPC processing of a call to the Register_For_Allocates service.
0000000A	AN INTERNAL FAILURE OCCURRED IN APPC PROCESSING.	An internal failure occurred during APPC processing of a call to the Register_For_Allocates service.
0000000B	FAILURE IN SERIALIZING SERVER FACILITIES RESOURCES.	Serialization of APPC/MVS server facilities resources failed during APPC processing of the Register_For_Allocates service.
0000000C	FAILURE IN SERIALIZING SERVER FACILITIES RESOURCES.	Serialization of APPC/MVS server facilities resources failed during APPC processing of the Register_For_Allocates service.
0000000D	AN INTERNAL FAILURE OCCURRED IN APPC PROCESSING.	An internal failure occurred during APPC processing of a call to the Register_For_Allocates service.
0000000E	AN INTERNAL FAILURE OCCURRED IN APPC PROCESSING.	An internal failure occurred during APPC processing of a call to either the Receive_Allocate or Unregister_For_Allocates service.
0000000F	AN INTERNAL FAILURE OCCURRED IN APPC PROCESSING.	An internal failure occurred during APPC processing of a call to either the Receive_Allocate or Unregister_For_Allocates service.
00000010	AN INTERNAL FAILURE OCCURRED IN APPC PROCESSING.	An internal failure occurred during APPC processing of a call to the Receive_Allocate service.

Table 28. Reason Codes for Service-Failure Errors (continued)

Reason Code	**Message Text**	**Explanation**
00000011	AN INTERNAL FAILURE OCCURRED IN APPC PROCESSING.	An internal error occurred in APPC/MVS processing.
00000012	AN INTERNAL FAILURE OCCURRED IN APPC PROCESSING.	An internal error occurred in APPC/MVS processing.
00000013	AN INTERNAL FAILURE OCCURRED IN APPC PROCESSING.	An internal error occurred in APPC/MVS processing.

Appendix C. Sample APPC/MVS Server

This program is the server part of a client/server application, which is written in the C programming language.

For your reference, the client half of this application appears in Appendix D, “Sample Client Program,” on page 97. Also, Appendix E, “Sample Error Routine and Header File,” on page 103 shows the source code for the error routine (SRVERROR) used by this application, and the C header file used to define error codes for the SRVERROR routine.

```
/*********************************************************************/
/*    This program is a sample server for a client/server pair       */
/*    written in C/370 to demonstrate the use of APPC/MVS allocate   */
/*    queue services.                                                */
/*                                                                   */
/*  COPYRIGHT --                                                     */
/*                                                                   */
/*  (C) Copyright IBM Corp. 1992                                     */
/*  All rights reserved.                                             */
/*  U.S, Government Users Restricted Rights -- Use,                  */
/*  duplication, or disclosure restricted by GSA ADP Schedule        */
/*  Contract with IBM Corp.    Program Property of IBM.              */
/*                                                                   */
/*  This program is provided to you for tutorial purposes only.      */
/*  You may not use the program for commercial purposes.             */
/*  This program is a sample working solution intended               */
/*  to show the use of APPC/MVS allocate queue services.             */
/*  Independent of its particular use, this program is               */
/*  supplied as an example and provided "as is" without              */
/*  warranty of any kind, either express or implied, including,      */
/*  but not limited to, the implied warranties of                    */
/*  merchantability and fitness for a particular purpose.            */
/*  The entire risk about the quality and performance of the         */
/*  program is with you.  Should the program prove defective,        */
/*  you assume the entire cost of all necessary servicing,           */
/*  repair, or correction.                                           */
/*                                                                   */
/*  In no event will IBM be liable to you for any damages or         */
/*  any lost profits, lost savings or other incidental or            */
/*  consequential damages arising out of the use of or               */
/*  inability to use the program even if IBM had been advised        */
/*  of the possibility of such damages, or for any claim by any      */
/*  other party.                                                     */
/*                                                                   */
/*                                                                   */
/*  MODULE NAME --                                                   */
/*                                                                   */
/*       SRV1MAIN.C                                                  */
/*                                                                   */
/*                                                                   */
/*  ENTRY POINTS --                                                  */
/*                                                                   */
/*       Normal C entry executed on MVS.                             */
/*                                                                   */
/*                                                                   */
/*  STATUS --                                                        */
/*                                                                   */
/*       Version 1, Release 0                                        */
/*                                                                   */
/*                                                                   */
/*  FUNCTION --                                                      */
/*                                                                   */
/*       This program is provided as an example of APPC/MVS allocate */
/*       queue services.  This program is the server half of         */
/*       a client/server pair.  The server uses the APPC/MVS         */
/*       Register_For_Allocates service to establish itself          */
/*       as an APPC/MVS server.  It then proceeds to process inbound */
/*       requests received through the Receive_Allocate service.     */
/*       Each client should immediately grant the server send        */
/*       control.  The server will then send data to the client      */
/*       and deallocate the conversation.  The server will           */
/*       terminate processing when the Receive_Allocate service      */
/*       shows that no requests have been received for 5 minutes.    */
/*       (Note that the 5 minute time-out value is set by a constant */
/*       in this program.)                                            */
/*                                                                    */
/*                                                                    */
/*  INPUT --                                                          */
/*                                                                    */
/*       None                                                         */
/*                                                                    */
/*  OUTPUT --                                                         */
/*                                                                    */
/*       None.                                                        */
/*                                                                    */
/*                                                                    */
/*  RETURN INFORMATION --                                             */
/*                                                                    */
/*       None.                                                        */
/*                                                                    */
/*                                                                    */
/*  CHANGE HISTORY --                                                 */
/*                                                                    */
/*       08/31/92 - Module created.                                   */
/*                                                                    */
/**********************************************************************/
/**********************************************************************/
/*                                                                    */
/*  Include the header files that define the services used by         */
/*  this program.  STDIO and STRING are standard C libraries.         */
/*  ATBCMC is the interface definition file (IDF) for the CPI-C       */
/*  services.  ERRCDE is the header file for the SRVERROR function    */
/*  that handles error conditions detected by this program.           */
/*  ATBCTC is the IDF for APPC/MVS callable transaction services      */
/*  which include APPC/MVS allocate queue services.                   */
/*                                                                    */
/*  The '#pragma runopts(execops)' is a C/370 option which permits    */
/*  the caller to specify runtime options to C/370 before             */
/*  specifying parameters to this program.  This was done to permit   */
/*  the caller to specify /NOSPIE and /NOSTAE to prevent C/370        */
/*  from suppressing a user abend which might be generated by the     */
/*  SRVERROR function.                                                */
/*                                                                    */
/**********************************************************************/
 #pragma runopts(execops)
 #include <STDIO.H>
 #include <STRING.H>
 #include <ATBCMC.H>
 #include <ERRCDE.H>
 #include <ATBCTC.H>

 /*********************************************************************/
 /*                                                                   */
 /*                  MAINLINE FUNCTION                                */
 /*                                                                   */
 /*********************************************************************/
main()

{
 /*********************************************************************/
 /*                                                                   */
 /*         VARIABLES USED FOR ALLOCATE QUEUE FUNCTIONS               */
 /*                                                                   */
 /*   server_ok - Used as a flag to indicate whether the server       */
 /*               is to continue processing.  This flag is set to     */
 /*               'false' for errors that affect the server's         */
 /*               ability to function.  Failure of the server in      */
 /*               processing a single client does not cause the       */
 /*               server to shut down.                                */
 /*                                                                   */
 /*   register_successful - Indicates whether the server's call to    */
 /*                         the Register_For_Allocates service was    */
 /*                         successful.                               */
 /*                                                                   */
 /*   atbrfa2_tp_name - This is the TP name that the server passes    */
 /*                     to the Register For Allocates service.        */
 /*                     Actually, this is just a "place holder"       */
 /*                     parameter because we are getting the          */
 /*                     TP name from the side information file.       */
 /*                     We will set the TP name length parameter      */
 /*                     to 0 and initialize this variable to blanks. */
 /*                                                                   */
 /*   atbrfa2_tp_name_length - The TP name length passed to Register */
 /*                            For Allocates.  Set to zero because  */
 /*                            we will use the TP name that is      */
 /*                            defined in the side information file.*/
 /*                                                               */
 /*   atbrfa2_local_lu_name - The local LU name passed to Register   */
 /*                            For Allocates.  Set to blanks because */
 /*                            we will use the local LU name that is */
 /*                            defined in the side information file. */
 /*                                                               */
 /*   atbrfa2_partner_lu_name - The 3 filter parameters for         */
 /*   atbrfa2_userid              Register For Allocates.  Set to     */
 /*   atbrfa2_profile             blanks because we don't want to     */
 /*                               filter                              */
 /*                                                               */
 /*   atbrfa2_sym_dest_name - The symbolic destination name passed   */
 /*                           to Register For Allocates.  This will  */
 /*                           be used to retrieve the TP name and    */
 /*                           local LU name from the side information*/
 /*                           data set.                              */
 /*                                                               */
 /*   qtoken - The allocate queue token returned by APPC/MVS to the  */
 /*            caller of the Register_For_Allocates service.         */
 /*             This token is used in all later APPC/MVS functions   */
 /*             that relate to this allocate queue.                  */
 /*                                                               */
 /*   return_code - The return code from an allocate queue service   */
 /*                                                               */
 /*   reason_code - The reason code from an allocate queue service   */
 /*                                                               */
 /*  Note:  The string parameters, such as local_lu_name, are        */
 /*         declared as character longer than necessary because we   */
 /*         are using the strcpy function to set them, and this will */
 /*         append a null ('00'x) to the string.  Declaring an extra */
 /*         character prevents this null character from being        */
 /*         considered part of the passed value.                     */
 /*                                                               */
 /*****************************************************************/
  int server_ok;             /* on-> server has met no errors        */
  int register_successful;
  char atbrfa2_tp_name[65];
  long int atbrfa2_tp_name_length;
  char atbrfa2_local_lu_name[9];
  char atbrfa2_partner_lu_name[18];
  char atbrfa2_userid[9];
  char atbrfa2_profile[9];
  char atbrfa2_sym_dest_name[9];
  char qtoken[8];
  long int return_code;
  long int reason_code;

 /*****************************************************************/
 /*  receive_allocate_type - The receive allocate type parameter     */
 /*             passed to the Receive Allocate service.  Initialized */
 /*             to 'TIMED' to indicate to APPC/MVS that we want to   */
 /*             wait until an allocate request has arrived on the    */
 /*             allocate queue or the time specified by the time_out */
 /*             parameter to have been exceeded.                     */
 /*                                                               */
 /*  time_out - The time_out_value parameter passed to the Receive   */
 /*              Allocate service.                                   */
 /*              Because we set the receive_allocate_type to         */
 /*              'TIMED', this parameter contains the amount         */
 /*              of time we want to wait for a Receive Allocate call */
 /*              to complete.                                        */
 /*                                                               */
 /*  convid   - The conversation ID that is returned by the Receive  */
 /*              Allocate service.  This identifier will be used     */
 /*              on later communication service calls for this       */
 /*              conversation.                                       */
 /*                                                               */
 /*  conv_type - Returned by the Receive Allocate service, this      */
 /*               parameter indicates whether the inbound            */
 /*               conversation is sending data only (mapped) or      */
 /*               data preceded by a two byte length indicator       */
 /*               (basic).                                           */
 /*                                                               */
 /*  partner_lu_name - Returned by the Receive Allocate service,     */
 /*         this parameter indicates the logical unit (LU) where the */
 /*         client transaction program is located.                   */
 /*                                                               */
 /*  mode - Returned by the Receive Allocate service, this parameter */
 /*          indicates the logon mode name used by the TP which      */
 /*          allocated this conversation.                            */
 /*                                                               */
 /*  sync_level - Returned by the Receive Allocate service, this     */
 /*              parameter indicates whether the conversation has    */
 /*              synchronization level NONE or CONFIRM.              */
 /*                                                                  */
 /*  userid - Returned by the Receive Allocate service, this         */
 /*           parameter indicates the user ID that was used by       */
 /*           the partner when the conversation was allocated.       */
 /*                                                                  */
 /*  profile - This returned parameter indicates the profile (or     */
 /*             RACF group name) that was used by the partner when   */
 /*             the conversation was allocated.                      */
 /*                                                                  */
 /********************************************************************/
  unsigned long int receive_allocate_type = ATBCTS_TIMED;
  unsigned long int time_out = 300;
  char convid[8];
  long int conv_type;
  char partner_lu_name[17];
  char mode[8];
  long int sync_level;
  char userid[8];
  char profile[8];

 /********************************************************************/
 /*                   NOTIFY TYPE STRUCTURE                          */
 /*                                                                  */
 /*   Allocate queue services can be issued asynchronously by        */
 /* supplying the address of an ECB in the notify_type parameter.    */
 /* This sample program does not use the asynchronous capability     */
 /* of these APPC/MVS services.  This structure is included, however,*/
 /* as an example of how you would set up the parameters to request  */
 /* asynchronous processing.                                         */
 /*                                                                  */
 /*   To issue the Receive Allocate service asynchronously,          */
 /* pass an eight byte notify type to the service.                   */
 /* Set the first 4 bytes of this field must be set to an integer 1. */
 /* The second four bytes must be set to the address of the ECB      */
 /* that is to be posted when the service completes.  The prototype  */
 /* for the service, however, expects a 'char *' for this parameter. */
 /*   So we declare the following UNION.  The first part of the      */
 /* UNION is an 8 character field that we will pass on the parameter */
 /* list to APPC/MVS.  The second part is a structure containing an  */
 /* integer (4 bytes) that is the notify type itself and a pointer   */
 /* (4 bytes) to the ECB.                                            */
 /*   When using this UNION, we will either set the notify_type_value*/
 /* to a '1' (indicating that we request asynchronous processing)    */
 /* and RAL_ECB will be set to the address of the ECB, or the        */
 /* notify_type_value will be set to '0' (indicating that we request */
 /* synchronous processing).  We don't have to set the RAL_ECB       */
 /* field when notify_type_value is zero (synchronous).              */
 /*                                                                  */
 /********************************************************************/
  union {
        char notify_type_char[8];
        struct {
                int notify_type_value;
                int *RAL_ECB;
               } nt_struc;
        } the_notify_type;
 /********************************************************************/
 /*              VARIABLES USED FOR COMMUNICATION SERVICES           */
 /*                                                                  */
 /*   buffer - A buffer into which we will receive data from the     */
 /*             partner TP.                                          */
 /*                                                                  */
 /*   requested_length - A passed parameter to the Receive service   */
 /*                       which indicates the size of our buffer.    */
 /*                                                                  */
 /*   data_received - A returned parameter from Receive that will    */
 /*                    indicate whether we received any data.        */
 /*                                                                  */
 /*   received_length - If we received data, this parameter will     */
 /*                      tell us how much.                           */
 /*                                                                  */
 /*   status_received - This returned parameter will tell us if we   */
 /*                      received any status from our partner.       */
 /*                                                                  */
 /*   rts_received - This returned parameter will tell us if our     */
 /*                   client has requested SEND control.  Not        */
 /*                   particularly relevant in this application      */
 /*                   because the client always immediately grants   */
 /*                   the server SEND control.                       */
 /*                                                                  */
 /*   communication_return_code - The return code from APPC/MVS      */
 /*                               communication services.            */
 /*                                                                  */
 /*   deallocate_type - used with the Set_Deallocate_Type (CMSDT)    */
 /*                     service to set the deallocate type to        */
 /*                     deallocate_flush.                            */
 /*                                                                  */
 /*   send_length - The amount of data being sent to the client.     */
 /*                 Set to the size of the buffer.                   */
 /*                                                                  */
 /********************************************************************/
  char buffer[19];
  long int requested_length;
  long int data_received;
  long int received_length;
  long int status_received;
  long int rts_received;
  long int communication_return_code;
  long int deallocate_type = CM_DEALLOCATE_FLUSH;
  long int send_length = sizeof(buffer);
 /********************************************************************/
 /*                VARIOUS OTHER LOCAL VARIABLES                     */
 /*                                                                  */
 /*   true - a generic constant indicating successful processing     */
 /*                                                                  */
 /*   false - a generic constant indicating a problem                */
 /*                                                                  */
 /*   srverror_return_code - used to test the decision made by       */
 /*                 the SRVERROR() function.                         */
 /*                                                                  */
 /********************************************************************/
  const int TRUE = 1;
  const int FALSE = 0;
  int srverror_return_code;

 /********************************************************************/
 /*                 BEGIN THE ACTUAL FUNCTION                        */
 /********************************************************************/

 /********************************************************************/
 /*                                                                  */
 /*  First we indicate that all is well...so far.                    */
 /*                                                                  */
 /********************************************************************/

  server_ok = TRUE;


    /*****************************************************************/
    /*                                                               */
    /*  Then we set the passed parameters.                           */
    /*                                                               */
    /*   - Notify_type is set to indicate synchronous processing     */
    /*   - TP_name length is set to zero                             */
    /*   - Local_LU_name is set to blanks                            */
    /*   - Partner_LU_name filter is set to blanks                   */
    /*   - User_ID filter is set to blanks                           */
    /*   - Profile filter is set to blanks                           */
    /*   - Sym_dest_name is set to the name of the entry in the      */
    /*       side information file that defines the TP name and      */
    /*       local LU name that we want to serve.                    */
    /*                                                               */
    /*****************************************************************/

    the_notify_type.nt_struc.notify_type_value = 0;
    atbrfa2_tp_name_length = 0;
    strcpy(atbrfa2_local_lu_name,"        ");
    strcpy(atbrfa2_partner_lu_name,"                 ");
    strcpy(atbrfa2_userid,"        ");
    strcpy(atbrfa2_profile,"        ");
    strcpy(atbrfa2_sym_dest_name,"SRVORDER");
    /*****************************************************************/
    /*                                                               */
    /*  Call the Register_for_Allocates service.                     */
    /*                                                               */
    /*  If the service is not completely successful, invoke the      */
    /*  SRVERROR function to find out what we should do.             */
    /*                                                               */
    /*****************************************************************/

    atbrfa2(the_notify_type.notify_type_char,
            atbrfa2_sym_dest_name,
              &atbrfa2_tp_name_length,
              atbrfa2_tp_name,
              atbrfa2_local_lu_name,
              atbrfa2_partner_lu_name,
              atbrfa2_userid,
              atbrfa2_profile,
              qtoken,
              &reason_code,
              &return_code);

      if (return_code != ATBCTS_OK)
       {
        error_description.problem = ATBRFA2_RET_CODE_ERROR;
        error_description.error_reason.rc_problem.expected_return_code =
             ATBCTS_OK;
        error_description.error_reason.rc_problem.actual_return_code =
             return_code;
        server_ok = (srverror(error_description) < server_failure);
       }
   /*********************************************************************/
   /*                                                                   */
   /*  Set the register_successful flag to indicate whether the service*/
   /*  was successful.   This flag is used during cleanup processing   */
   /*  to determine whether the Unregister_For_Allocates service is    */
   /*  called.                                                          */
   /*                                                                   */
   /*********************************************************************/
      register_successful = server_ok;

   /*********************************************************************/
   /*                     MAIN LOOP                                     */
   /*                                                                   */
   /*  This is the main loop wherein we will receive allocate requests */
   /* from the queue and process them.  We exit the loop when the      */
   /* Receive_Allocate return code shows that the time-out value       */
   /* has been exceeded.                                                */
   /*                                                                   */
   /*********************************************************************/

    while (server_ok)
     {
      /*******************************************************************/
      /*                                                                 */
      /* Set-up and call a synchronous Receive_Allocate.                 */
      /*                                                                 */
      /*   - Set the notify type to request synchronous processing       */
      /*   - Call the service..note that the allocate queue token        */
      /*      is the one returned by the Register_For_Allocates          */
      /*      service.  The receive_allocate_type has been set to        */
      /*      'TIMED'.                                                   */
      /*                                                                 */
      /*******************************************************************/

       the_notify_type.nt_struc.notify_type_value = 0;

       atbral2(the_notify_type.notify_type_char,
               qtoken,
               &receive_allocate_type,
               &time_out,
               convid,
               &conv_type,
               partner_lu_name,
               mode,
               &sync_level,
               userid,
               profile,
               &reason_code,
               &return_code);

      if (return_code != ATBCTS_OK)
       if (return_code == ATBCTS_REQUEST_UNSUCCESSFUL)
        if (reason_code == ATBCTS_NO_ALLOC_TO_RECEIVE)
         server_ok = FALSE;
        else
         {
          error_description.problem = ATBRAL2_REASON_CODE_ERROR;
          error_description.error_reason.reason_problem.
               expected_reason_code = ATBCTS_NO_ALLOC_TO_RECEIVE;
          error_description.error_reason.reason_problem.
               actual_reason_code = reason_code;
          server_ok = (srverror(error_description) < server_failure);
         }
       else
         {
          error_description.problem = ATBRAL2_RET_CODE_ERROR;
          error_description.error_reason.rc_problem.
               expected_return_code = ATBCTS_OK;
          error_description.error_reason.rc_problem.actual_return_code=
               return_code;
          server_ok = (srverror(error_description) < server_failure);
         }
      /* else return code is ok then all is well...proceed */

       if (server_ok)
        {
           /*********************************************************/
           /*                                                       */
           /*  Receive SEND control from the client                 */
           /*    - Set the requested_length to zero because we are  */
           /*      expecting to receive send control from the client. */
           /*                                                       */
           /*********************************************************/

           requested_length = 0;

           cmrcv (convid,
                 buffer,
                 &requested_length,
                 &data_received,
                 &received_length,
                 &status_received,
                 &rts_received,
                 &communication_return_code);

           /*********************************************************/
           /*                                                       */
           /*  Verify that the results are as expected.             */
           /*    - Did the service complete successfully?           */
           /*    - If so, did we receive SEND control?              */
           /*                                                       */
           /*********************************************************/

           if (communication_return_code != CM_OK)
            {
             error_description.problem = CMRCV_RET_CODE_ERROR;
             error_description.error_reason.rc_problem.
                  expected_return_code = CM_OK;
             error_description.error_reason.rc_problem.
                  actual_return_code = communication_return_code;
             srverror_return_code = srverror(error_description);
            }
           else if (status_received != CM_SEND_RECEIVED)
            {
             error_description.problem = CMRCV_STATUS_ERROR;
             error_description.error_reason.status_problem.
                  expected_status = CM_SEND_RECEIVED;
             error_description.error_reason.status_problem.
                  actual_status = status_received;
             srverror_return_code = srverror(error_description);
            }
           /*********************************************************/
           /*                                                       */
           /*  If all is still well, then send the client the data.   */
           /*                                                       */
           /*********************************************************/
           if (server_ok)
            {

             strcpy(buffer,"123456789012345678");

             cmsend(convid,
                    buffer,
                    &send_length,
                    &rts_received,
                    &communication_return_code);

             if (communication_return_code != CM_OK)
              {
               error_description.problem = CMSEND_RET_CODE_ERROR;
               error_description.error_reason.rc_problem.
                    expected_return_code =
                    CM_OK;
               error_description.error_reason.rc_problem.
```

```
                  actual_return_code = communication_return_code;
             srverror_return_code = srverror(error_description);
            }

         /*********************************************************/
         /*                                                       */
         /*  At this point, we could wait for some return data from */
         /*  the client, but since this is just a sample we will    */
         /*  go ahead and deallocate the conversation.  The default */
         /*  deallocate type is deallocate_sync_level.  So, if the  */
         /*  partner (who allocated the conversation) has set the   */
         /*  sync_level to confirm, the deallocate request will     */
         /*  wait until the partner issues Confirmed.  Since we     */
         /*  don't want to delay the server waiting for the client  */
         /*  to acknowledge that we are done, we change the         */
         /*  deallocate type to deallocate_flush using the CMSDT    */
         /*  (Set Deallocate Type) service.                         */
         /*                                                       */
         /*********************************************************/
           cmsdt(convid,
                 &deallocate_type,
                 &communication_return_code);

           if (communication_return_code != CM_OK)
            {
             error_description.problem = CMSEND_RET_CODE_ERROR;
             error_description.error_reason.rc_problem.
                  expected_return_code = CM_OK;
             error_description.error_reason.rc_problem.
                  actual_return_code = communication_return_code;
             srverror_return_code = srverror(error_description);
            }
         /*********************************************************/
         /*                                                       */
         /*  And then deallocate the conversation.                */
         /*                                                       */
         /*********************************************************/
           cmdeal(convid,
                  &communication_return_code);

           if (communication_return_code != CM_OK)
            {
             error_description.problem = CMDEAL_RET_CODE_ERROR;
             error_description.error_reason.rc_problem.
                  expected_return_code = CM_OK;
             error_description.error_reason.rc_problem.
                  actual_return_code = communication_return_code;
             srverror_return_code = srverror(error_description);
            }
          }
      }

 /*******************************************************************/
 /*                                                                 */
 /*  At the bottom of the loop, check to see if any of the          */
 /*  conversation services have received a return code from SRVERROR */
 /*  indicating that the server should shut down.                   */
 /*                                                                 */
 /*******************************************************************/

    if (srverror_return_code==server_failure)
      server_ok = false;

   }
 /*******************************************************************/
 /*                                                                 */
 /*  After exiting the loop either because of an error or because   */
 /* Receive_Allocate timed out, we will clean up after ourselves by  */
 /* unregistering from the allocate queue we created.  We do this    */
 /* by calling the Unregister For Allocates service.  Note that      */
 /* this is done only if a flag shows that the Register_For_Allocates*/
 /* call was successful.                                             */
 /*                                                                 */
 /*  The only input parameters are the notify_type and the allocate  */
 /* queue token.  The notify type is set to indicate synchronous     */
 /* processing and the queue token is still set from the Register    */
 /* For_Allocates service.                                           */
 /*                                                                 */
 /*******************************************************************/

  if (register_successful)
   {
     the_notify_type.nt_struc.notify_type_value = 0;

     atbura2(the_notify_type.notify_type_char,
             qtoken,
             &reason_code,
             &return_code);

     if (return_code != ATBCTS_OK)
      {
       error_description.problem = ATBURA2_RET_CODE_ERROR;
       error_description.error_reason.rc_problem.expected_return_code =
            CM_OK;
       error_description.error_reason.rc_problem.actual_return_code =
            return_code;
       srverror_return_code = srverror(error_description);
      }
    }
  /*******************************************************************/
  /*                                                                 */
  /*              That's it!                                         */
  /*                                                                 */
  /*******************************************************************/

  return 0;
 }
```

Appendix D. Sample Client Program

This program is the client part of the sample client/server application that began in Appendix C, “Sample APPC/MVS Server,” on page 87.

Appendix E, “Sample Error Routine and Header File,” on page 103 shows the source code for the error routine (SRVERROR) used by this application, and the C language header file used to define error codes for the SRVERROR routine.

```
/***********************************************************************/
/*    This program is a sample client for a client/server pair        */
/*    written in C/370 to demonstrate the use of APPC/MVS allocate    */
/*    queue services.  This client program uses the CPI-C             */
/*    interface and uses no MVS-specific services.                    */
/*                                                                    */
/*                                                                    */
/*  COPYRIGHT --                                                      */
/*                                                                    */
/*  (C) Copyright IBM Corp. 1992                                      */
/*  All rights reserved.                                              */
/*  U.S, Government Users Restricted Rights -- Use,                   */
/*  duplication, or disclosure restricted by GSA ADP Schedule         */
/*  Contract with IBM Corp.    Program Property of IBM.               */
/*                                                                    */
/*  This program is provided to you for tutorial purposes only.       */
/*  You may not use the program for commercial purposes.              */
/*  This program is a sample working solution intended                */
/*  to show the use of APPC/MVS allocate queue services.              */
/*  Independent of its particular use, this program is                */
/*  supplied as an example and provided "as is" without               */
/*  warranty of any kind, either express or implied, including,       */
/*  but not limited to, the implied warranties of                     */
/*  merchantability and fitness for a particular purpose.             */
/*  The entire risk about the quality and performance of the          */
/*  program is with you.  Should the program prove defective,         */
/*  you assume the entire cost of all necessary servicing,            */
/*  repair, or correction.                                            */
/*                                                                    */
/*  In no event will IBM be liable to you for any damages or          */
/*  any lost profits, lost savings or other incidental or             */
/*  consequential damages arising out of the use of or                */
/*  inability to use the program even if IBM had been advised         */
/*  of the possibility of such damages, or for any claim by any       */
/*  other party.                                                      */
/*                                                                    */
/*                                                                    */
/*  MODULE NAME --                                                    */
/*                                                                    */
/*       DRV1.C                                                       */
/*                                                                    */
/*                                                                    */
/*  ENTRY POINTS --                                                   */
/*                                                                    */
/*       Normal C entry executed on the client platform.              */
/*                                                                    */
/*                                                                    */
/*  STATUS --                                                         */
/*                                                                    */
/*       Version 1, Release 0                                         */
/*                                                                    */
/*                                                                    */
/*  FUNCTION --                                                       */
/*                                                                    */
/*       This program is provided as an example of APPC/MVS           */
/*       allocate queue services.  This program is the client half    */
/*       of a client/server pair.  It invokes the server and waits    */
/*       for the server to respond by sending data.                   */
/*                                                                    */
/*                                                                    */
/*  INPUT --                                                          */
/*                                                                    */
/*       None                                                         */
/*                                                                    */
/*  OUTPUT --                                                         */
/*                                                                    */
/*       None.                                                      */
/*                                                                  */
/*                                                                  */
/*  RETURN INFORMATION --                                           */
/*                                                                  */
/*       None.                                                      */
/*                                                                  */
/*                                                                  */
/*  CHANGE HISTORY --                                               */
/*                                                                  */
/*       08/31/91 - Module created.                                 */
/*                                                                  */
/********************************************************************/
/********************************************************************/
/*                                                                  */
/*  Include the header files which define the services used by      */
/*  this program.  STDIO and STRING are standard C libraries.       */
/*  ATBCMC is the interface definition file (IDF) for the CPI-C     */
/*  services.  ERRCDE is the header file for the srverror function  */
/*  that handles error conditions detected by this program.         */
/*                                                                  */
/*  The '#pragma runopts(execops)' is a C/370 option which permits  */
/*  the caller to specify runtime options to C/370 before           */
/*  specifying parameters to this program.  This was done to permit */
/*  the caller to specify /NOSPIE and /NOSTAE to prevent C/370      */
/*  from suppressing a user abend which might be generated by the   */
/*  srverror function.                                              */
/*                                                                  */
/********************************************************************/
 #pragma runopts(execops)
 #include <STDIO.H>
 #include <STRING.H>
 #include <ATBCMC.H>
 #include <ERRCDE.H>
/********************************************************************/
/*                                                                  */
/*                      MAINLINE CODE                               */
/*                                                                  */
/********************************************************************/
main()

{

/********************************************************************/
/*                                                                  */
/*                      DECLARE VARIABLES                           */
/*                                                                  */
/*  conv_id - conversation identifier returned by APPC on the CMINIT */
/*             call and used on subsequent calls                    */
/*                                                                  */
/*  sym_dest - symbolic destination name identifying the server     */
/*             (TP name, LU name, and logon mode).  Note that this  */
/*             variable is one character longer than the symbolic   */
/*             destination name parameter.  Since a value is placed */
/*             in this parameter using the strcpy() function we must */
/*             provide an extra character for the null since it     */
/*             must not be a part of the passed value.              */
/*                                                                  */
/*  return_code - used to hold return codes from APPC services      */
/*                                                                  */
/*  buffer - The buffer which is used to receive data from the      */
/*            partner program (server).  The buffer is 19 characters */
/*            long because we know that is how much data the server */
/*            will be sending.                                      */
/*                                                                  */
/*  requested_length - The length of the buffer we will provide to  */
/*                      the receive (CMRCV) service.                */
/*                                                                  */
/*  data_received - A returned parameter from the CMRCV service which */
/*                   will indicate whether any data was received from */
/*                   the server.                                    */
/*                                                                  */
/*  received_length - A returned parameter from the CMRCV server which*/
/*                     will indicate the amount of data placed into */
/*                     our buffer by APPC.                          */
/*                                                                  */
/*  status_received - A returned parameter from the CMRCV service   */
/*                     which will indicate whether any status was   */
/*                     received from the server.                    */
/*                                                                  */
/*  rts_received - A returned parameter from the CMRCV service which */
/*                  indicates whether the partner has requested send */
/*                  control.  Should always be set to                */
/*                  rts_not_received in this application since this   */
/*                  program always immediately grants the server      */
/*                  send control.                                     */
/*                                                                    */
/*  srverror_return_code - The return code from the srverror function.*/
/*                                                                    */
/**********************************************************************/
  char conv_id[8];
  char sym_dest[9];
  long int return_code;
  char buffer[19];
  long int requested_length;
  long int data_received;
  long int received_length;
  long int status_received;
  long int rts_received;
  int srverror_return_code;

/**********************************************************************/
/*                                                                    */
/*  Set the sym_dest variable to the symbolic destination name.       */
/*  There must be an entry defined in the side information table      */
/*  for this value.  It must contain an LU name and TP name which     */
/*  correspond to the values for which the server has registered.     */
/*  See APPC/MVS Planning and Management for information about adding */
/*  Side Information.                                                 */
/*                                                                    */
/*  The srverror function return code is initialized to zero.
/*                                                                    */
/**********************************************************************/
  strcpy(sym_dest,"SRVORDER");
  srverror_return_code = 0;

/**********************************************************************/
/*                                                                    */
/*                   INITIALIZE THE CONVERSATION                      */
/*                                                                    */
/*  Call the Initialize_Conversation service (CMINIT), providing      */
/*  the symbolic destination name defined just above.  If all goes    */
/*  well, a conversation identifier will be returned in the conv_id   */
/*  variable.                                                         */
/*                                                                    */
/*  If all is not well (i.e. the return code is not CM_OK) then       */
/*  invoke the srverror function providing a description of the       */
/*  problem encountered (CMINIT return code error), the expected      */
/*  return code value, and the actual return code received.           */
/*  The srverror function will then act appropriately and             */
/*  return a return code indicating whether to continue processing.   */
/*                                                                    */
/*  Of course, in this case any non-zero return code should result    */
/*  in termination of processing; the srverror function, however,     */
/*  can update an error log, issue an operator message, or take other */
/*  appropriate action.                                               */
/*                                                                    */
/**********************************************************************/
  CMINIT(conv_id,
         sym_dest,
         &return_code);
  if (return_code != CM_OK)
   {
    error_description.problem = CMINIT_RET_CODE_ERROR;
    error_description.error_reason.rc_problem.expected_return_code =
         CM_OK;
    error_description.error_reason.rc_problem.actual_return_code =
         return_code;
    srverror_return_code = srverror(error_description);
   }

/**********************************************************************/
/*                                                                    */
/*                  ALLOCATE THE CONVERSATION                         */
/*                                                                    */
/*  If all is well, allocate the conversation by calling the CMALLC   */
/*  service.  The only input parameter is the conversation identifier */
/*  returned by CMINIT.                                               */
/*                                                                    */
/*  If the allocate function fails (non-zero return code), we again   */
/*  check with the srverror function to find out if we should         */
/*  continue processing.                                              */
/*                                                                    */
/**********************************************************************/
```

```
  if (srverror_return_code==0)
   {
    CMALLC(conv_id,
          &return_code);

    if (return_code != CM_OK)
     {
      error_description.problem = CMALLC_RET_CODE_ERROR;
      error_description.error_reason.rc_problem.expected_return_code =
           CM_OK;
      error_description.error_reason.rc_problem.actual_return_code =
           return_code;
      srverror_return_code = srverror(error_description);
     }
   }
/*********************************************************************/
/*                                                                   */
/*                  RECEIVE DATA FROM THE SERVER                     */
/*                                                                   */
/*  Next two tasks are accomplished by calling one function.  When   */
/*  the CMRCV service is called from send state, notification is     */
/*  first sent to the server that it has been granted send control   */
/*  and then this program waits for the server to send data.         */
/*  Note that we set the requested_length parameter to the size of   */
/*  the receive buffer.                                              */
/*                                                                   */
/*  Next the returned parameters will be examined and the srverror   */
/*  function invoked if any unexpected results occur.                */
/*                                                                   */
/*********************************************************************/
  if (srverror_return_code==0)
   {
    requested_length = sizeof(buffer);

    cmrcv (conv_id,
          buffer,
          &requested_length,
          &data_received,
          &received_length,
          &status_received,
          &rts_received,
          &return_code);
/*********************************************************************/
/*                                                                   */
/*  The first returned parameter to examine is naturally the return  */
/*  code.  Two values may be expected on this call.  Since we know   */
/*  the partner will deallocate the conversation after sending the   */
/*  data, we expect to get a return code of CM_DEALLOCATED_NORMAL.   */
/*  It is possible, however, that the return code may not have arrived*/
/*  yet, so a return code of CM_OK might be returned.  If neither    */
/*  value is found, the srverror function is invoked.                */
/*                                                                   */
/*********************************************************************/
    if (return_code != CM_OK)
      if (return_code != CM_DEALLOCATED_NORMAL)
       {
        error_description.problem = CMRCV_RET_CODE_ERROR;
        error_description.error_reason.rc_problem.
             expected_return_code = CM_OK;
        error_description.error_reason.rc_problem.
             actual_return_code = return_code;
        srverror_return_code = srverror(error_description);
       }

/*********************************************************************/
/*                                                                   */
/*  If the return code is OK or DEALLOCATED_NORMAL, then we can      */
/*  examine the data received field to determine if any data was     */
/*  received.  If data was received, then we can examine the         */
/*  received_length field to determine how much data was received.   */
/*  If we received the expected length, then we can proceed to       */
/*  examine the data itself to verify it is as expected.             */
/*                                                                   */
/*  Note that in a real application the actual value of the expected */
/*  data would probably not be known, but this check can be easily   */
/*  replaced with a check verifying that the data is in some expected */
/*  format (for example, if the expected data were inventory record  */
/*  updates you might expect the data to consist of item identifiers  */
/*  and quantities in four byte integer pairs).                      */
/*                                                                   */
/*  If any returned values are not as expected, the srverror function */
/*  is invoked.                                                     */
/*                                                                  */
/********************************************************************/
    if ((return_code==CM_OK)|(return_code==CM_DEALLOCATED_NORMAL))
     {
      if (data_received!=CM_COMPLETE_DATA_RECEIVED)
       {
        error_description.problem = CMRCV_DATA_RCV_ERROR;
        error_description.error_reason.data_rcv_problem.
             expected_data_rcv = CM_COMPLETE_DATA_RECEIVED;
        error_description.error_reason.data_rcv_problem.
             actual_data_rcv = data_received;
        srverror_return_code = srverror(error_description);
       }

      if (srverror_return_code == 0)
       {
        if (received_length != requested_length)
         {
          error_description.problem = CMRCV_RCVD_LEN_ERROR;
          error_description.error_reason.length_problem.
               expected_length = requested_length;
          error_description.error_reason.length_problem.
               actual_length = received_length;
          srverror_return_code = srverror(error_description);
         }

        if (srverror_return_code == 0)
         {
          if (strcmp(buffer,"123456789012345678"))
           {
            error_description.problem = CMRCV_BUFFER_ERROR;
            error_description.error_reason.data_problem.
                 expected_data = "123456789012345678";
            error_description.error_reason.data_problem.
                 actual_data = buffer;
            srverror_return_code = srverror(error_description);
           }
         }
       }
/********************************************************************/
/*                                                                  */
/*  At this point we have verified that all the information we      */
/*  expected to receive has arrived.  We have not examined the status */
/*  received field since we expected to receive no status.  Just to   */
/*  be complete, we will verify that we did in fact receive no status */
/*  and invoke the srverror function if status did turn up.          */
/*  Note that this check occurs inside a conditional which ensures    */
/*  that we only examine the status_received field when the return    */
/*  code  is CM_OK since the status field is not set for the          */
/*  CM_DEALLOCATED_NORMAL return code (or other non-zero return codes)*/
/*                                                                  */
/********************************************************************/

      if (srverror_return_code == 0)
       {
        if (return_code == CM_OK)
         {
          if (status_received != CM_NO_STATUS_RECEIVED)
           {
            error_description.problem = CMRCV_STATUS_ERROR;
            error_description.error_reason.status_problem.
                 expected_status = CM_NO_STATUS_RECEIVED;
            error_description.error_reason.status_problem.
                 actual_status = status_received;
            srverror_return_code = srverror(error_description);
           }
         }
       }
     }
   }
/********************************************************************/
/*                                                                  */
/*  At this point, the client has received the data from the server   */
/*  and could perform any processing required such as updating a      */
/*  local database with new information.                            */
/*                                                                  */
/********************************************************************/
/********************************************************************/
/*                                                                  */
/*  As mentioned above, it is possible the notification of the end    */
/*  of the conversation might not have arrived on the first receive   */
/*  as a DEALLOCATED_NORMAL return code.  In this case, we need to    */
/*  issue another CMRCV to get this return code.  Note that we set    */
/*  the requested_length to zero for this receive since we expect no  */
/*  data to arrive.  After the receive completes, the return code     */
/*  is inspected and the srverror function is invoked if an           */
/*  unexpected value is found.  We also examine the data_received     */
/*  field to verify it is set to CM_NO_DATA_RECEIVED.                 */
/*                                                                    */
/**********************************************************************/
  if (srverror_return_code == 0)
   {
    if (return_code == CM_OK)
     {
      requested_length = 0;

      cmrcv (conv_id,
            buffer,
            &requested_length,
            &data_received,
            &received_length,
            &status_received,
            &rts_received,
            &return_code);

      if (return_code != CM_DEALLOCATED_NORMAL)
       {
        error_description.problem = CMRCV_RET_CODE_ERROR;
        error_description.error_reason.rc_problem.
             expected_return_code = CM_DEALLOCATED_NORMAL;
        error_description.error_reason.rc_problem.
             actual_return_code = return_code;
        srverror_return_code = srverror(error_description);
       }

      if (srverror_return_code == 0)
       {
        if (data_received != CM_NO_DATA_RECEIVED)
         {
          error_description.problem = CMRCV_DATA_RCV_ERROR;
          error_description.error_reason.data_rcv_problem.
               expected_data_rcve = CM_NO_DATA_RECEIVED;
          error_description.error_reason.data_rcv_problem.
               actual_data_rcv = data_received;
          srverror_return_code = srverror(error_description);
         }
       }
     }
   }

/**********************************************************************/
/*                                                                    */
/*  The function of the client is complete.                           */
/*                                                                    */
/**********************************************************************/
 return srverror_return_code;
}
```

Appendix E. Sample Error Routine and Header File

This program is the error routine that can be called by either the sample server (see Appendix C, “Sample APPC/MVS Server,” on page 87) or sample client program (see Appendix D, “Sample Client Program,” on page 97) when an error is detected. When called, SRVERROR writes an error message ("An error has occurred") to the server's joblog and returns control.

This program uses variables defined in a separate C language header file. See “Header File” on page 103.

```
 #include <STDIO.H>
 #include <STRING.H>
 #include <ATBCMC.H>
 #include <ATBCTC.H>


int srverror();

{

 printf("An error has occurred\n");

 /*The error handling code should be placed here.                      */
 /*                                                                    */
 return;
}
```

Header File

This is the C language header file used to define error code variables for the sample error routine, SRVERROR. Compile this header file with both the sample client and server programs included in this book.

```
 /*********************************************************************/
 /*                                                                   */
 /*                      SRVERROR HEADER FILE                         */
 /*                                                                   */
 /* This is the header file for the sample error routine, SRVERROR.   */
 /*                                                                   */
 /* The calling program passes to SRVERROR information that           */
 /* indicates the type of error that occurred and the expected and    */
 /* actual value for the returned parameter which was in error.       */
 /* This information is passed in the structure described below.      */
 /*                                                                   */
 /* SRVERROR examines the information, logs the error, if             */
 /* appropriate, and returns one of the following return codes:       */
 /*                                                                   */
 /*  no_server_error - indicates no serious error has occurred and    */
 /*                    processing should continue normally.           */
 /*                                                                   */
 /*  conversation_failure - indicates a failure in processing the     */
 /*                         conversation.  No further processing      */
 /*                         should be done for this conversation.     */
 /*                         If the calling program is a server,       */
 /*                         the program does not end because of       */
 /*                         this error.                               */
 /*                                                                   */
 /*  server_error - indicates a failure in processing for the server.*/
 /*                 Server processing should be terminated.           */
 /*                                                                   */
 /*********************************************************************/
int srverror();
#define no_server_error                             0
#define conversation_failure                        4
#define server_failure                              8
 /*********************************************************************/
 /*                STRUCTURE FOR COMMUNICATING ERRORS                 */
 /*                                                                   */
 /*   This data structure is passed to the SRVERROR function          */
 /*   to describe the error encountered.  The first component         */
 /*   of the structure is the error_type, which is a numeric          */
 /*   value that describes the APPC callable service that             */
 /*   received an unexpected result and the parameter on              */
 /*   the call that appears to be in error.                           */
 /*                                                                   */
 /*   Following the error_type indicator are the expected value       */
 /*   of the parameter apparently in error and the actual             */
 /*   value returned by the call.  In the case of numeric             */
 /*   parameters, the values are passed in the structure.             */
 /*   In the case of character parameters, a pointer to the           */
 /*   character string is passed.                                     */
 /*                                                                   */
 /*********************************************************************/
  struct {
          int problem;
          union {
                struct {
                       long int expected_return_code;
                       long int actual_return_code;
                       } rc_problem;
                struct {
                       long int expected_reason_code;
                       long int actual_reason_code;
                       } reason_problem;
                struct {
                       char *expected_data;
                       char *actual_data;
                       } data_problem;
                struct {
                       long int expected_status;
                       long int actual_status;
                       } status_problem;
                struct {
                       long int expected_data_rcv;
                       long int actual_data_rcv;
                       } data_rcv_problem;
                struct {
                       long int expected_length;
                       long int actual_length;
                       } length_problem;
                } error_reason;
         } error_description;
 #define ATBRFA2_RET_CODE_ERROR                    168
 #define ATBRAL2_RET_CODE_ERROR                    176
 #define ATBRAL2_REASON_CODE_ERROR                 177
 #define ATBURA2_RET_CODE_ERROR                    188
 #define CMALLC_RET_CODE_ERROR                     202
 #define CMDEAL_RET_CODE_ERROR                     206
 #define CMINIT_RET_CODE_ERROR                     218
 #define CMRCV_BUFFER_ERROR                        220
 #define CMRCV_DATA_RCV_ERROR                      221
 #define CMRCV_RCVD_LEN_ERROR                      222
 #define CMRCV_STATUS_ERROR                        223
 #define CMRCV_RET_CODE_ERROR                      225
 #define CMSEND_RET_CODE_ERROR                     228
```

Appendix F. Accessibility

Accessible publications for this product are offered through IBM Knowledge Center (www.ibm.com/support/knowledgecenter/SSLTBW/welcome).

If you experience difficulty with the accessibility of any z/OS information, send a detailed message to the Contact z/OS web page (www.ibm.com/systems/z/os/zos/webqs.html) or use the following mailing address.

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
United States

Accessibility features

Accessibility features help users who have physical disabilities such as restricted mobility or limited vision use software products successfully. The accessibility features in z/OS can help users do the following tasks:

- Run assistive technology such as screen readers and screen magnifier software.
- Operate specific or equivalent features by using the keyboard.
- Customize display attributes such as color, contrast, and font size.

Consult assistive technologies

Assistive technology products such as screen readers function with the user interfaces found in z/OS. Consult the product information for the specific assistive technology product that is used to access z/OS interfaces.

Keyboard navigation of the user interface

You can access z/OS user interfaces with TSO/E or ISPF. The following information describes how to use TSO/E and ISPF, including the use of keyboard shortcuts and function keys (PF keys). Each guide includes the default settings for the PF keys.

- *z/OS TSO/E Primer*
- *z/OS TSO/E User's Guide*
- *z/OS ISPF User's Guide Vol I*

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users who access IBM Knowledge Center with a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line because they are considered a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or `3.1` or `3.1.1`. To hear these numbers correctly, make sure that the screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number `3.1`) are mutually exclusive alternatives. If you hear the lines `3.1 USERID` and `3.1 SYSTEMID`, your syntax can include either `USERID` or `SYSTEMID`, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The `*` symbol is placed next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element `*FILE` with dotted decimal number 3 is given the format `3 \* FILE`. Format `3* FILE` indicates that syntax element `FILE` repeats. Format `3* \* FILE` indicates that syntax element `* FILE` repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol to provide information about the syntax elements. For example, the lines `5.1*, 5.1 LASTRUN, and 5.1 DELETE` mean that if you use more than one of the `LASTRUN` and `DELETE` syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the `%` symbol, it indicates a reference that is defined elsewhere. The string that follows the `%` symbol is the name of a syntax fragment rather than a literal. For example, the line `2.1 %OP1` means that you must refer to separate syntax fragment OP1.

The following symbols are used next to the dotted decimal numbers.

? indicates an optional syntax element
: The question mark (?) symbol indicates an optional syntax element. A dotted decimal number followed by the question mark symbol (?) indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the `?` symbol is displayed on the same line as the syntax element, (for example `5? NOTIFY`). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines `5 ?, 5 NOTIFY`, and `5 UPDATE`, you know that the syntax elements `NOTIFY` and `UPDATE` are optional. That is, you can choose one or none of them. The `?` symbol is equivalent to a bypass line in a railroad diagram.

! indicates a default syntax element
: The exclamation mark (`!`) symbol indicates a default syntax element. A dotted decimal number followed by the `!` symbol and a syntax element indicate that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the dotted decimal number can specify the ! symbol. For example, if you hear the lines `2? FILE, 2.1! (KEEP), and 2.1 (DELETE)`, you know that `(KEEP)` is the default option for the `FILE` keyword. In the example, if you include the `FILE` keyword, but do not specify an option, the default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the `FILE` keyword is omitted, the default `FILE(KEEP)` is used. However, if you hear the lines `2? FILE, 2.1, 2.1.1! (KEEP)`, and `2.1.1 (DELETE)`, the default option KEEP applies only to the next higher dotted decimal number, `2.1` (which does not have an associated keyword), and does not apply to `2? FILE`. Nothing is used if the keyword `FILE` is omitted.

*** indicates an optional syntax element that is repeatable**
: The asterisk or glyph (*) symbol indicates a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line `5.1* data area`, you know that you can include one data area, more than one data area, or no data area. If you

hear the lines `3* , 3 HOST, 3 STATE`, you know that you can include `HOST, STATE`, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write `HOST STATE`, but you cannot write `HOST HOST`.
3. The * symbol is equivalent to a loopback line in a railroad syntax diagram.

+ indicates a syntax element that must be included

The plus (+) symbol indicates a syntax element that must be included at least once. A dotted decimal number followed by the + symbol indicates that the syntax element must be included one or more times. That is, it must be included at least once and can be repeated. For example, if you hear the line `6.1+ data area`, you must include at least one data area. If you hear the lines `2+, 2 HOST, and 2 STATE`, you know that you must include `HOST, STATE,` or both. Similar to the * symbol, the + symbol can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loopback line in a railroad syntax diagram.

Notices

Terms and conditions for product documentation

Applicability

Personal use

Commercial use

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies that collect each user's name, email address, phone number, or other personally identifiable information for purposes of enhanced user usability and single sign-on configuration. These cookies can be disabled, but disabling them will also eliminate the functionality they enable.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at ibm.com/privacy and IBM's Online Privacy Statement at ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies," and the "IBM Software Products and Software-as-a-Service Privacy Statement" at ibm.com/software/info/product-privacy.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, JES2, JES3, and MVS™, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: IBM Lifecycle Support for z/OS (www.ibm.com/software/support/systemsz/lifecycle)
- For information about currently-supported IBM hardware, contact your IBM representative.

Programming Interface Information

This book is intended to help the customer to design and write APPC/MVS servers. This book documents General-use Programming Interface and Associated Guidance Information provided by z/OS.

General-use programming interfaces allow the customer to write programs that obtain the services of z/OS.

Trademarks

Index

SA23-1396-40