CHAPTER 3

Program Synchronization and Control

This chapter discusses how the application program synchronizes with the transport provider and controls the execution sequence of Cisco IOS for S/390 API macro instructions. The information presented in this chapter together with the concepts and facilities presented in the preceding chapter provide the necessary background for understanding how the macro instructions operate. For more detailed information about each API macro instruction, read the *Cisco IOS for S/390 Assembler API Macro Reference*.

The chapter includes these sections:

- Task Synchronization Requirements

  Describes how to synchronize with the completion of a TPL-based service request and overlap endpoint processing with other application program activities.

- Modes of Operation

  Describes the various modes of operation for TPL-based service requests and provides detailed information on synchronization characteristics of the API macro instructions.

- Specifying and Using Exit Routines

  Describes how to specify, code, and use exit routines for processing various asynchronous events.

- Handling Errors and Special Conditions

  Describes how to detect and handle errors and other special conditions.

- Application Program Organization

  Describes facilities for program control and synchronization.

- Multitasking Operation Rules

  Describes the rules for using the API in a multitasking environment.

- Multiple Address Spaces

  Describes the facility provided to pass sessions from one address space to another.

- 24-bit and 31-Bit Addressing

  Describes use of the API with 24-bit and 31-bit addressing.

Task Synchronization Requirements

The API must be capable of providing service within a variety of application program environments; single-threaded programs serving a single user must be supported as well as multi-threaded programs supporting several users simultaneously. In the latter case, such programs may be multi-task or multi-address-space, or may be a single task with a built-in scheduling algorithm for servicing individual user requests (, CICS). Also, the API must support the capability of overlapping network I/O with other application program processing (for example, disk I/O).

An important characteristic of the API design is that, at the option of the application program, no transport service request causes the issuing task to be suspended. As a result, the processing of service requests can be easily overlapped with other application program activities, and when executing on a multi-processor, can actually be executed in parallel. To support this requirement, the API provides a mechanism for initiating a service request and synchronizing with its subsequent completion under total control of the application program.

The synchronization mechanisms implemented by the API are similar to those of VTAM, and use standard MVS facilities. Thus, a programmer who is familiar with VTAM assembler language programming has already been introduced to most of the principles presented in this section. However, it is suggested that such programmers read this section as a review of those principles.

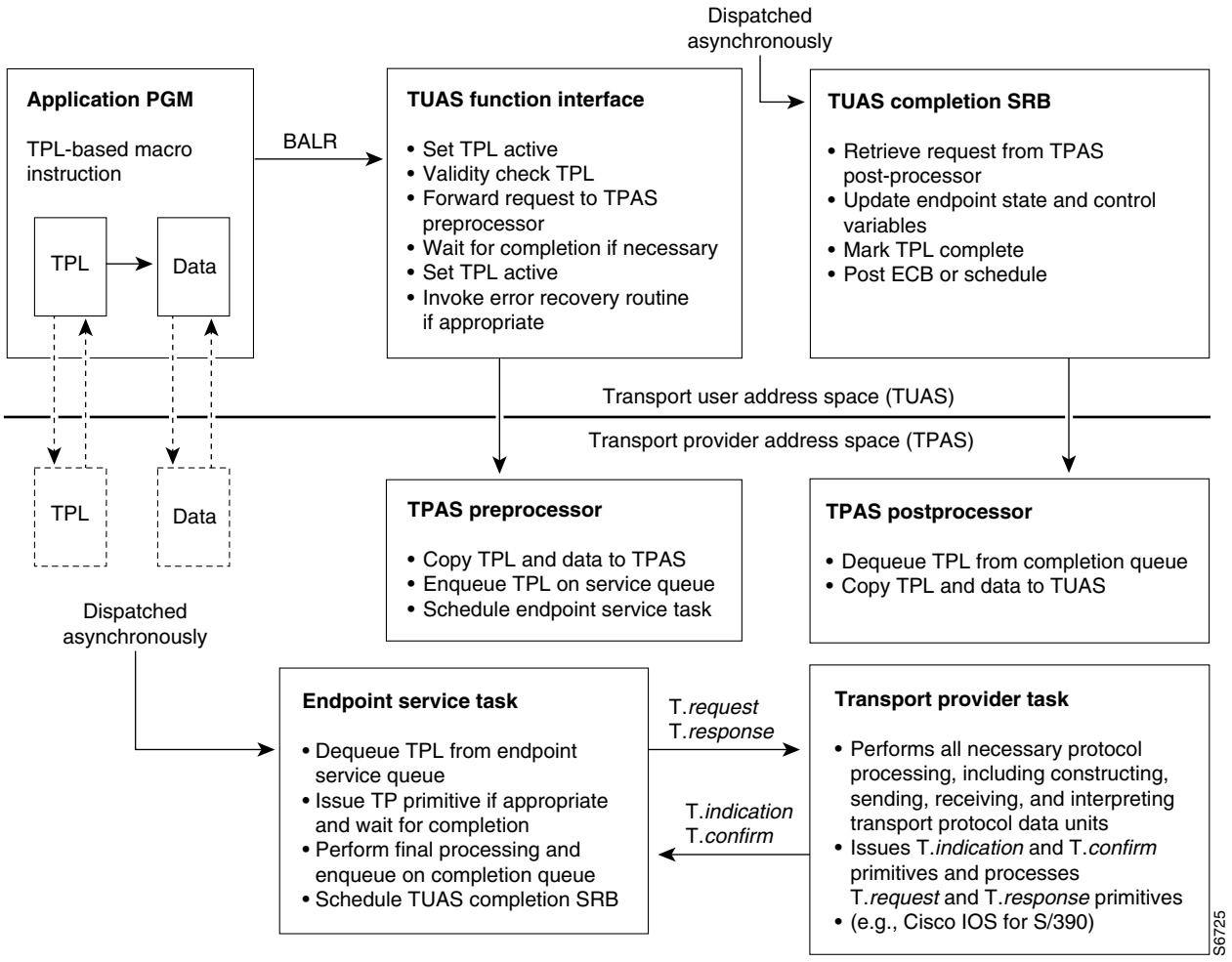
Typical Processing Flow

A service request requires processing in two MVS address spaces. The Transport User Address Space *(*TUAS*)* is the address space of the application program that initiates the request. The Transport Provider Address Space *(*TPAS*)* is the address space containing the API subsystem and the transport service provider that eventually processes the request. In some cases the transport provider may reside in an external device (for example, a specialized network I/O processor), and the transport provider depicted in this diagram is merely a stub that communicates with the external device. In either case, all other entities are present, and the actual location of the transport provider is transparent to the application program.

Execution Flow

This diagram illustrates the typical processing flow for a transport service request

Figure 3-1 Processing Flow for a Transport Service Request



Service Request Initiation

All service requests are initiated by executing an API macro instruction. This list describes the steps included in this process:

- The API macro expansion generates executable instructions that call a TUAS function interface which has been loaded into common storage.
- The TUAS function interface locates the PC number for the TPAS interface and issues the PC instruction. The TPAS PC routine is a space-switched PC. The TPAS becomes the primary address space, although execution continues under control of the application program dispatchable unit.
- The TPAS PC routine locates and calls a TLI function-dependent processing routine (TPAS function processor).
- The TPAS function processor validates the request parameters and copies the TPL and associated data into the API address space.
- The TPAS function processor calls the respective transport provider routines (in other words, connection management and data transfer routines).

- Depending upon the specific API request and/or the existing conditions of the endpoint at the time, the request may either complete immediately, or it may need to wait for a protocol event such as the arrival of data or a connect request.
- When a protocol event must occur before completing the request, either the application's dispatchable unit is suspended or the request is queued internally and control is returned to the application program. The requested mode of service (synchronous or asynchronous) determines the disposition of the event-pending request.

  When the anticipated protocol event occurs, suspended synchronous requests are resumed. An SRB routine is scheduled into the application address space to re-drive the internally queued request via the TPAS PC interface.

  During the time that an asynchronous request is queued awaiting a protocol event, the application program may perform additional processing. Eventually, the application must issue a TCHECK to resynchronize with the request.
- The TPL and associated data areas in the application address space are updated.
- The application is notified of the completed request. For synchronous requests, this simply consists of returning control to the application. For asynchronous requests, completion processing depends if the request specified ECB or EXIT for completion notification.
  - If ECB was specified, a cross-memory post is executed.
  - If EXIT was specified, the exit routine address and parameter are placed into a queue element for processing by the TUAS IRB routine. The IRB routine must also be scheduled if it is not already active. Synchronization modes are discussed in Modes of Operation.

Modes of Operation

The mode of operation affects how processing proceeds after a service request has been accepted and scheduled for processing by the API address space. The mode of operation is selected individually for each request by indicating an option code when the macro instruction is executed.

This section describes the various modes of operation for TPL-based service requests and provides detailed information on synchronization characteristics of API macro instructions.

Example

OPTCD=SYNC indicates synchronous operation, and causes the request to be completed before control is returned to the application program.

OPTCD=ASYNC indicates asynchronous operation, and causes control to be returned prior to completion if the request must await a protocol event.

Processing that precedes acceptance of a service request, and all processing within the API address space, is identical for both modes of operation. When a service request is issued, the TPL is set active and its contents are checked for validity. If the request is valid and the state of the endpoint is acceptable, the request is scheduled for processing by the API address space. When the request completes, the TPL is posted complete. The TPL remains active during this entire period of time, and if a subsequent request attempts to use the TPL, it is completed immediately with an error.

Operating Mode Differences

The difference between operating modes arises after the request has been issued, and only affects processing in the application program's address space.

- For synchronous mode, the TPAS function interface waits for the request to complete, and sets the TPL inactive before returning to the application program.
- For asynchronous mode, the TPL is left in its active state when control is returned. It is the responsibility of the application program to wait for completion and cause the TPL to be set inactive by executing a TCHECK macro instruction.

Execution of the TCHECK macro instruction is implicit in synchronous mode, and explicit in asynchronous mode.

Synchronization Characteristics of Macro Instructions

This table lists each mode of operation. Operation modes are discussed in further detail in the Synchronous Operation examples:

Table 3-1 Modes of Operation

Macro Instruction	Transport Provider Primitive Required For Completion	Notes
ACLOSE		1
AOPEN		
APCB		
TACCEPT	T-CONNECT.response	2
TADDR		
TBIND		
TCHECK		3
TCLEAR		
TCLOSE		4
TCONFIRM	T-CONNECT.confirm	
TCONNECT	T-CONNECT.request	
TDISCONN	T-DISCONNECT.request	
TDSECT		
TERROR		
TEXEC	Dependent on function code	
TEXLST		
TINFO		
TLISTEN	T-CONNECT.indication	
TOPEN		
TOPTION		
TPL		
TRECV	T-DATA.indication or T-EXPEDITED-DATA.indication	
TRECVERR		

Table 3-1 Modes of Operation (Continued)

Macro Instruction	Transport Provider Primitive Required For Completion	Notes
TRECVFR	T-UNITDATA.indication	
TREJECT	T-DISCONNECT.request	
TRELACK	T-RELEASE.indication	
TRELEASE	T-RELEASE.request	
TRETRACT		
TSEND	T-DATA.request or T-EXPEDITED-DATA.request	2
TSENDTO	T-UNITDATA.request	
TSTATE		
TUNBIND		
TUSER		

Note These notes correspond to the note numbers in the table:

1 The API issues a synchronous TCLOSE for all opened endpoints.

2 Primitive issued to a TCP-based provider may delay completion until an acknowledgment is received from the remote transport provider.

3 The API may issue a system WAIT macro instruction if pending request has not been completed.

4 The API issues T-DISCONNECT.request primitive if connection has not been released.

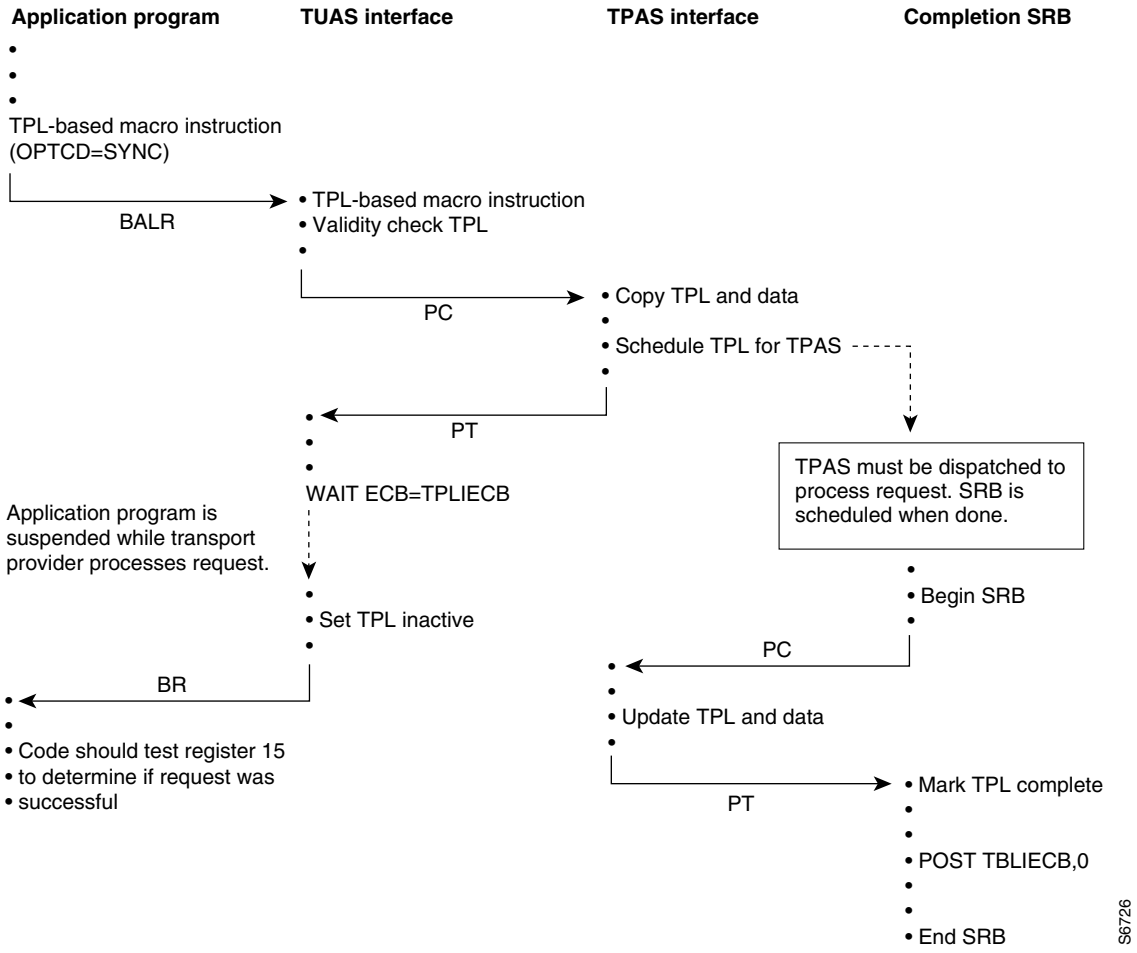
Synchronous Operation

In a synchronous program, operations are performed serially. A request for synchronous operation (OPTCD=SYNC) means that the API does not return control to the next sequential instruction in the application program task from which the macro instruction was issued until after the requested operation is completed. Execution of the application program task is suspended by issuing a system WAIT macro instruction until the API has completed the request. The program must wait for the processing of one requested operation to be completed before going on to the next.

Synchronous Operation Flow

This diagram illustrates the flow of a synchronous operation:

Figure 3-2 Synchronous Operation Flow



When control is returned to the next sequential instruction in the application program task, the TPL has been set inactive, and may be reused for another request. Synchronous mode is appropriate for application programs that manage a single endpoint and do not need to overlap endpoint operations, such as send and receive operations, with other application program activities. This suggests that the communication of data via the endpoint is half-duplex in nature, and that some arrangement exists between the communicating entities to know when to send and receive data. Since only one operation may be pending at any point in time, synchronous operation is usually less complex than asynchronous operation.

While the application program is waiting for the request to complete, an asynchronous event such as a timer interrupt could cause the program's STIMER exit routine to be entered. Similarly, asynchronous API events could cause the corresponding exit routines to be entered (see the discussion of exit routines in Specifying and Using Exit Routines). Only the application program task from which the macro instruction was issued is suspended while waiting for completion of a synchronous operation. The exit routines associated with the program are scheduled and executed whether or not the mainline program logic is awaiting completion of a synchronous operation. However, if a synchronous operation is issued from within an exit routine, executions of other exit routines may be prevented until the operation completes.

When a synchronous operation is completed, the application program must determine whether the operation was successful or unsuccessful. The program does this by testing values in registers 15 and 0 and by examining fields in the TPL used for the operation. For more information on error handling, refer to Handling Errors and Special Conditions.

Asynchronous Operation

In an asynchronous operation, control of execution may return to the application program next sequential instruction before the requested operation has completed. The transport provider may not be able to complete an operation immediately if dependent upon some event, such as the arrival of data or a protocol indication, or if it is necessary to serialize the request with other API requests for a given endpoint. In these situations, the transport provider analyzes request for errors and saves the request to await the respective event before returning to the application program. At this point, the program may issue other API requests or perform other application specific activities.

Example

An application program can issue a TRECV macro instruction on one endpoint indicating asynchronous mode, and while the input operation is being performed, the application program can send some data on the same endpoint, initiate an operation on a different endpoint, or perform other I/O activities, such as reading or writing a direct-access storage device. The application program can determine if the request was accepted by testing the value returned in register 15. If the value is 0, the request was accepted and the associated TPL is set active. If the value is non-zero, the request was not accepted, and register 15 and 0 contains the same information as if the request had been executed in synchronous mode. In this case, the TPL sets inactive, and the application program should initiate error recovery without waiting for the request to complete.

While an asynchronous operation is pending, the associated TPL is active and cannot be used with another request until it has been posted complete. If the application program issues another TPL-based macro instruction, a different TPL must be used. Also, when an operation is pending for an endpoint, other operations which can be initiated at the same endpoint are limited.

When an asynchronous operation is specified, there are two ways the API can notify the application program that the requested operation has been completed. If the application program associates an Event Control Block (ECB) with the request, the API posts the ECB when the operation is completed. Alternatively, the application program can designate that a particular TPL exit routine be executed as soon as the operation is completed. When the operation is completed, the API schedules the exit routine. The method of notification is controlled by storing the address of the ECB or exit routine in the TPL used for the request. Each method is discussed in the remainder of this section.

Regardless of whether a program waits on an ECB or uses a TPL exit routine, a TCHECK macro instruction must be executed after an asynchronous operation completes to set the TPL inactive and to make it available for another request. The TCHECK macro instruction also clears the ECB if one was provided.

It is also important to note that the TPL ECB may be posted or the TPL exit routine may be executed before the application receives control back from the API (for example, at the next sequential instruction following the TLI request macro). This is because this activity occurs upon a different unit of execution within the operating system.

Asynchronous Operation using ECBs

By using ECBs, the application program can issue one WAIT macro instruction for a combination of pending API requests in addition to non-API requests that also use ECBs.

Example

An application program can issue three TRECV requests for three different endpoints and three BSAM WRITE requests for three different data sets. By issuing one WAIT for all six ECBs, the application program resumes processing when any one of the six operations is completed. When execution resumes, the application program can determine which operation completed by determining which ECBs are posted.

The distinction between ECBs and TPL exit routines is primarily that the TPL exit routine is automatically scheduled when the requested operation is completed, thereby saving the application program the trouble of testing ECBs and branching to subroutines. On the other hand, the use of ECBs provides the program with greater control over the order in which events are to be handled.

Example

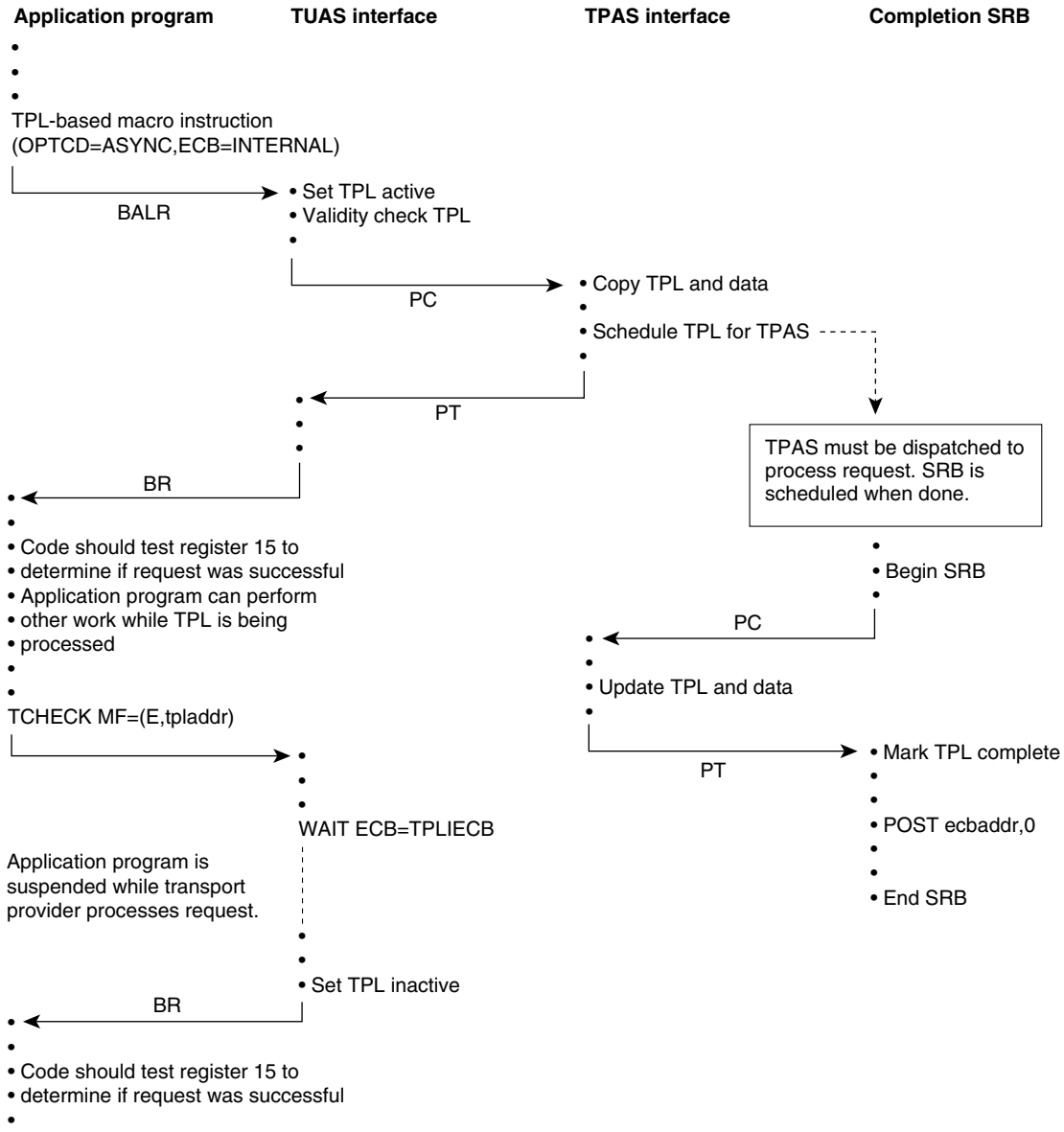
The application program can prioritize requested operations by testing some ECBs before others. The order of testing can be varied during program execution as circumstances change.

If neither an ECB address nor a TPL exit routine address is specified by the TPL-based macro instruction, the API uses the ECB-EXIT field of the TPL (TPLECBXR) as an internal ECB, and the API (for synchronous operations) or the application program (for asynchronous operations) waits on it, checks and clears it. Alternatively, the ECB-EXIT field can be set to point to an external ECB provided by the application program by using a TPL-based macro instruction that specifies ECB=ecb address. Once set, it can be reset to an internal ECB by specifying ECB=INTERNAL. If OPTCD=SYNC is specified, ECB=INTERNAL is assumed.

Asynchronous Operation Using Internal ECB

When using an internal ECB, the application program does not wait or check the internal ECB, and lets the TCHECK function wait if necessary. The application program could have waited on the ECB explicitly knowing that it is located in the TPL location at TPLIECB (TPLECBXR). This diagram illustrates this processing flow:

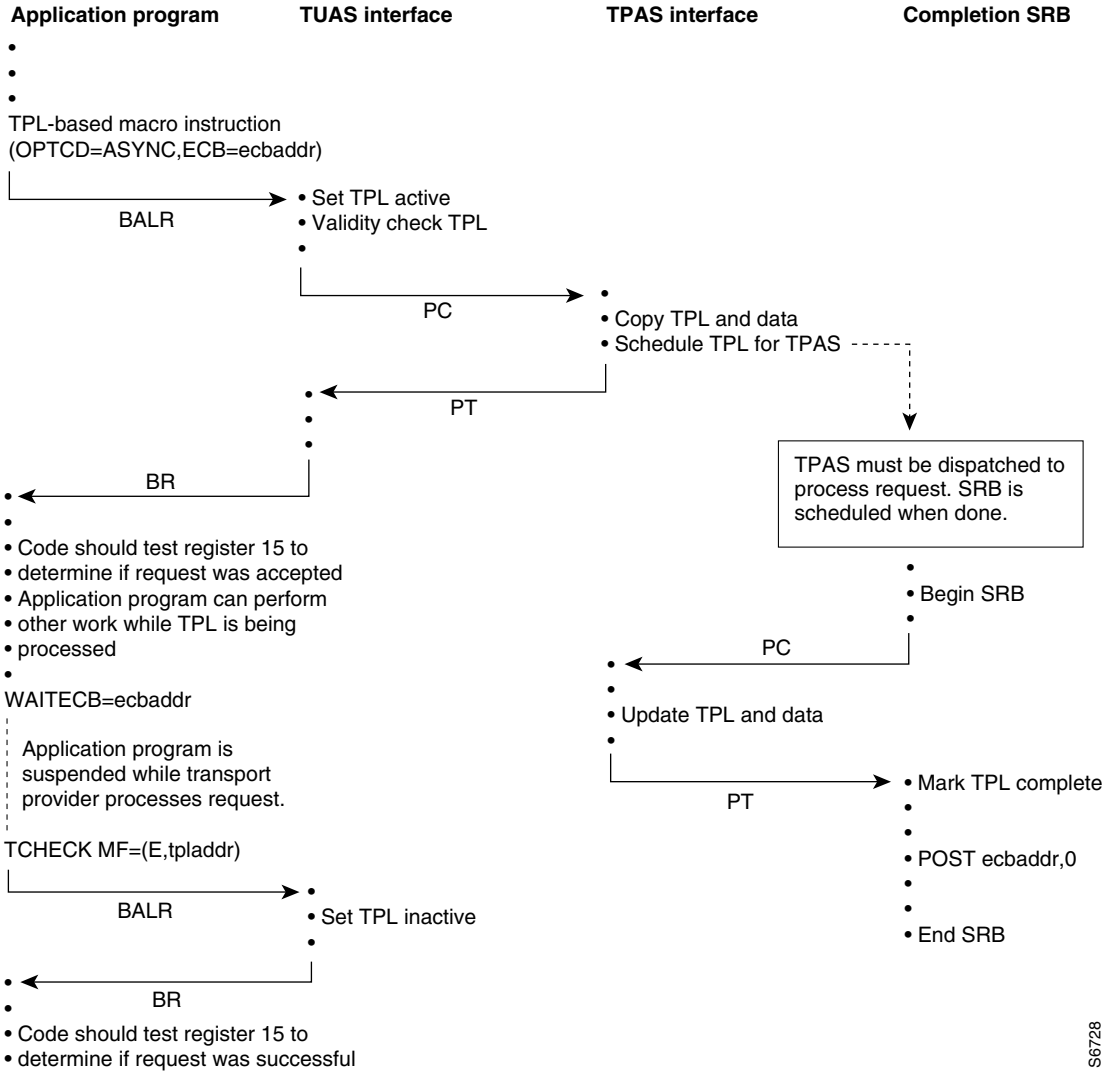
Figure 3-3 Asynchronous Operation Using Internal ECB



Asynchronous Operation Using External ECB

When using an external ECB, the application program issues a WAIT for the external ECB, but could have allowed TCHECK to perform the wait implicitly. This diagram illustrates this processing flow:

Figure 3-4 Asynchronous Operation Using External ECB



Asynchronous Operation Using Completion Exits

Instead of having the API post an ECB when a request for an asynchronous operation is completed, the application program can have the API schedule and cause control to be given to a TPL-specified asynchronous exit routine. The TPL exit routine can supply the logic that would have been branched to by the mainline program after discovering a posted ECB. A TPL exit routine is any exit routine whose symbolic name has been provided in the EXIT operand of the macro instruction or the TPL used for the request.

One advantage of using a TPL exit routine instead of an ECB is that it is easier to code for that type of processing than it is to code the logic associated with discovering a posted ECB and relating the ECB to a branch address. Also, the TPL exit routine is given control almost immediately after the associated TPL-based operation completes, and thus has priority over the mainline program. A disadvantage of a TPL exit routine is that more system instructions must be executed to schedule an exit routine than must be executed to post an ECB. Also, as discussed in the previous section,

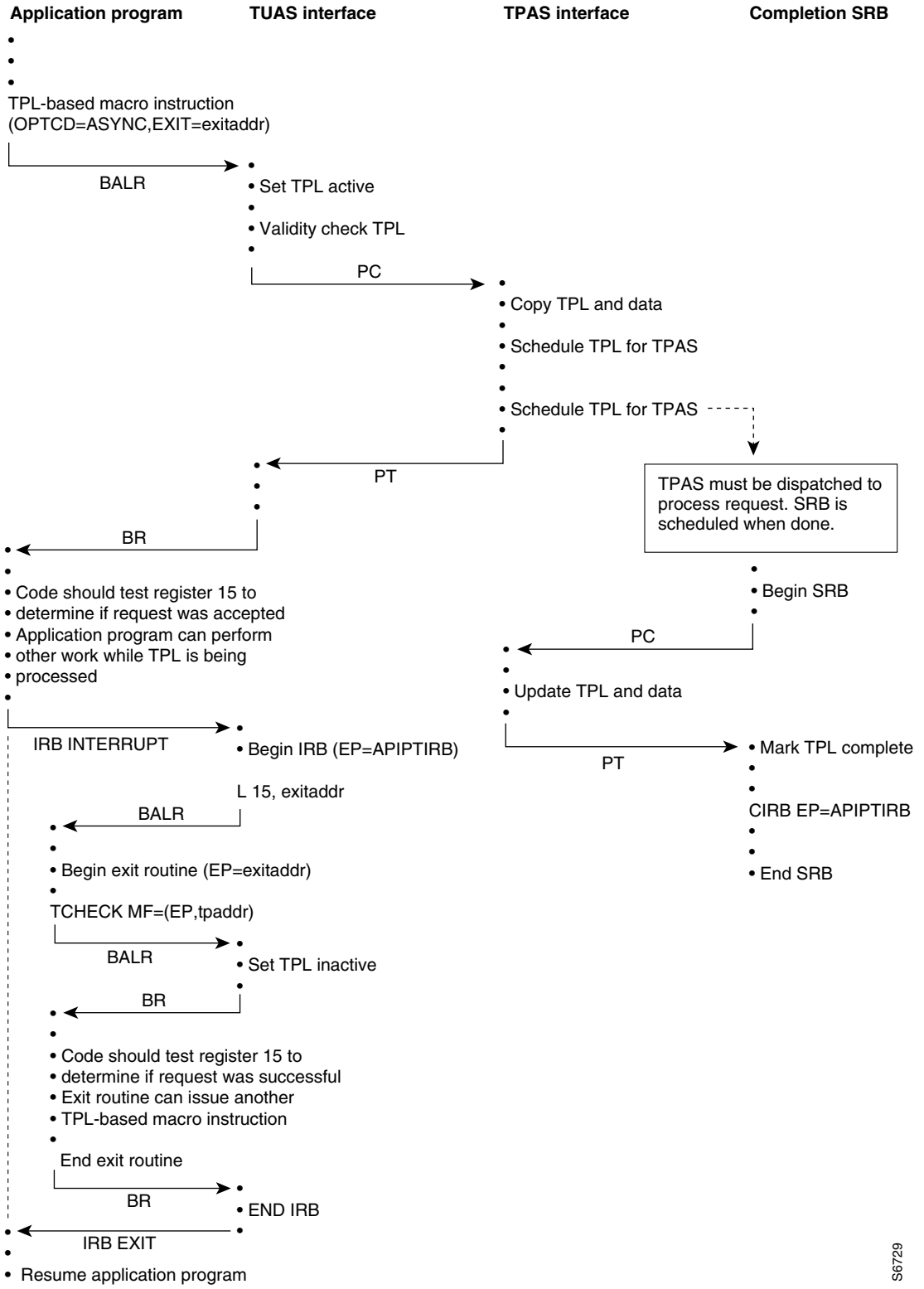
TPL-exit scheduling does not provide as much flexibility as ECB-posting for giving a higher priority to selected operations. An application program can use a combination of ECB-posting and TPL exit routines (see Mixing Synchronization Modes on mixing modes of operation).

A TPL exit routine may itself issue asynchronous requests, continue executing, and return to the API. The asynchronous request in a TPL exit routine may specify that on completion of the request, an ECB is to be posted or a TPL exit routine is to be scheduled. If the TPL exit routine option is chosen, the exit routine can be the same one in which the request was issued, or different.

A TPL exit routine may also issue synchronous requests, but the exit routine is suspended until the synchronous operation is completed. Since the API schedules exit routines serially for a given task, a synchronous request issued in an exit routine also prevents other exit routines from receiving control until control is returned from the suspended exit routine. The TPL exit routine may have run, or the TPL completion ECB may have been posted prior to the return from the asynchronous request.

This diagram illustrates this processing flow:

Figure 3-5 Asynchronous Operation Using Completion Exits



Using the TCHECK Macro Instruction

When an asynchronous request is accepted by the API, control is returned to the next sequential instruction of the application program with the TPL marked as active. No other request can be issued with an active TPL until the operation completes and the TPL is marked inactive.

A TCHECK macro instruction must be executed to set the TPL inactive. If ECB-posting was specified for a request, the application program generally waits for the ECB to be posted and then issue the TCHECK macro instruction. However, if a TCHECK macro instruction is executed before the ECB is posted, the API waits for the ECB to be posted. In either case, the ECB is cleared by the API before returning to the application program.

When a TPL exit routine is specified, the TCHECK macro instruction is usually executed by the exit routine. However, the TCHECK macro instruction may be executed by the mainline program after the exit routine has run, but this may be difficult to coordinate. If the TCHECK macro instruction is executed before the operation completes (and EXIT is specified), an error is generated.

The TCHECK macro instruction also causes error recovery exits to be entered. This is covered in more detail in Handling Errors and Special Conditions.

Mixing Synchronization Modes

Since the synchronization mode used for a request is determined at the time the request is issued, the modes of operation previously described may be mixed. That is, an application program may issue some requests in synchronous mode, and issue others in asynchronous mode. Similarly, some asynchronous requests can specify ECB-posting, and others can specify TPL exit routines.

The important point to remember is that once a synchronous request is issued, the operation must be completed before another request can be issued from the same program thread. If the program thread is the mainline program, exit routines can still run and requests can be issued from those exit routines. However, if a synchronous request is issued by an exit routine, no other API-scheduled exit routines run until the synchronous operation is complete. This fact can be used to the advantage or detriment of the application program.

Specifying and Using Exit Routines

The API lets an application program use exit routines to gain control to handle certain events. An exit routine is written to handle a specific event (for example, a SYNAD routine is written to process TPL-based errors or special conditions other than program logic errors). When the event occurs, the API gives the exit routine control as soon as possible.

This section discusses how exit routines are specified, how they are called by the API, and describes procedures to follow in using them. An overview of the API exit routines is presented, and then each exit routine is described in detail.

How Exit Routines Are Specified

The API provides for the use of two general kinds of exit routines by an application program:

- TPL request completion exit routines
- TEXLST or TEVNTLST protocol event exit routines

These two kinds of exit routines work somewhat differently and derive their names from how they are specified.

TPL Exit Routines

The instructions to execute when a TPL-based service request is completed can be written as a separate routine. This routine, called a TPL exit routine, can be specified in the TPL-based macro instruction used to initiate the request.

The address of the exit routine is specified by the EXIT operand of the macro instruction or is stored in the TPLEXIT field of the TPL. When the requested operation is completed, the API schedules and eventually causes entry to the TPL exit routine. The same TPL exit routine can be designated by more than one TPL-based macro instruction (in other words, a TPL exit routine can be established as a common exit routine).

TEXLST and TEVNTLST Exit Routines

TEXLST and TEVNTLST exit routines differ from TPL exit routines in being special-purpose (in other words, TEXLST/TEVNTLST exit routines handle special events that are well understood by both the application program and the API). Instead of being specified in a particular TPL-based macro instruction request, the identity of a TEXLST/TEVNTLST exit routine is established only when the exit list in which its name is specified is identified to the API, either when the application program session is established with the API or, for certain types of exit routines, when an endpoint is opened. In other words, TPL exit routines are associated with a particular service request, and TEXLST/TEVNTLST exit routines are associated with a transport user or a particular endpoint.

The TEXLST macro instruction is used to build an exit list, and the exit list is associated with the transport user by linking it to the APCB specified in the AOPEN macro instruction. For those exits that can be associated with a particular endpoint, the exit list must be specified in the TOPEN macro instruction. The TEXLST macro instruction keyword used to specify the exit routine's address is used in the remainder of this section when discussing particular types of exit routines.

Example

The exit routine specified by the TPEND keyword is referred to as the TPEND exit routine.

The TEVNTLST macro instruction provides capabilities similar to TEXLST. TEVNTLST, however, can only be used with the TOPEN macro instruction. TEVNTLST may not be referenced by an APCB macro instruction. Internally, the TEVNTLST macro generates a parameter list that is identical to the parameter list generated by TEXLST. TEVNTLST provides the additional capability of specifying ECB addresses in lieu of exit routines. Both ECB addresses and exit routine addresses can be used in a TEVNTLST specification, but for a given event, only one ECB or exit address may be specified.

When specifying TEXLST exit routines, the content and length of an exit list is dependent on whether the exit list is used with an AOPEN or TOPEN macro instruction, and must be indicated with the TEXLST macro instruction. An AOPEN exit list cannot be used with TOPEN, and vice versa.

The AOPEN and TOPEN event lists are hierarchical. When a particular endpoint event occurs that is being handled by the application program, the TOPEN event list associated with the endpoint is accessed first. If no exit routine or ECB address was specified in the TOPEN event list, then the AOPEN exit list is used. Only certain types of exit routines can be specified in a TOPEN event list, however all types can be specified in an AOPEN exit list. Thus, the AOPEN exit list can be used to provide default exit routines for the application program in general, and the TOPEN exit list can provide special exit routines and ECBs for particular endpoints if and when the need arises.

How Exit Routines Are Called

Exit routines are classified as synchronous or asynchronous exit routines according to how they are called by the API. The SYNAD and LERAD exit routines are synchronous, and all other exit routines are asynchronous. Exit routines have different characteristics based on their classification. This classification should not be confused with similar terminology used to describe modes of operation for service requests.

Synchronous Exit Routines

Synchronous exit routines are sometimes referred to as inline exit routines and are considered to be an extension of the part of the application program (either mainline or asynchronous exit routine) that was executing when the synchronous exit routine was invoked. Effectively a synchronous exit routine is a branch entered from a TPL-based or TCHECK macro instruction just before control is returned to the next sequential instruction in the application program.

After a synchronous exit routine has completed processing, it may return to the API. If it is coded to return, the application program receives control at the next sequential instruction immediately after the TPL-based or TCHECK macro instruction that caused the synchronous exit routine to be invoked. If the exit routine is coded not to return, it is as if the application program branched to another location after issuing the TPL-based or TCHECK macro instruction.

Register Information

When a synchronous exit routine is entered, the general-purpose registers contain this information:

Table 3-2 Synchronous Exit Routine Register Information

R0	Recovery action code
R1	Address of TPL
R2-12	Unmodified application program registers
R13	Unmodified application program save area address
R14	the API return address
R15	Address of exit routine

Synchronous Exit Routine Coding Procedures

These procedures should be followed when coding a synchronous exit routine:

- The exit routine is not obligated to return to the API. However, if it does, register 14 should be saved and restored before returning.
- The exit routine is not required to save and restore general-purpose registers 2-12, and the API does not provide a save area for the exit routine. However, register 13 does contain the address of an 18-word save area supplied by the application program.
- If the exit routine issues any TPL-based macro instructions or calls any external routines, it must provide the address of its own 18-word save area in register 13. Register 13 should be restored before returning to the API.

- If the exit routine returns to the API, it should not execute any macro instruction or call any routine that would change the contents of the 18-word save area supplied by the application program. The API restores the application program's registers from the save area before returning to the next sequential instruction.
- Registers 0 and 15 are not restored by the API and the exit routine is not required to preserve them. Contents of these registers are passed to the application program if the exit routine returns to the API.

A synchronous exit routine is generally not required to be reentrant. However, if a synchronous exit routine executes other TPL-based macro instructions, the exit routine may be re-entered (recursion). Also, if TPL-based service requests are issued from the mainline program and asynchronous exit routines, or the synchronous exit routine is shared by two or more programs, it may be reentered. In this case, the application program should be prepared to handle reentrancy issues (for example, dynamic allocation of work areas).

Synchronous Exit Routine Example

Here is an example of a synchronous exit routine that returns to the API. An 18-word save area is provided since it may issue other service requests or call external routines:

```
*************************************************************************
*  SYNCHRONOUS ERROR RECOVERY ROUTINE
*************************************************************************
          USING SYNADX,12
SYNADX    LR    12,15               ESTABLISH BASE ADDRESS
          LR    2,0                 SAVE RECOVERY ACTION CODE
          LTR   3,1                 TEST TPL ADDRESS
          BNP   RECURSIV            IGNORE IF CALLED RECURSIVELY
          LR    5,14                OTHERWISE, SAVE RETURN ADDRESS
          O     3,=X'80000000'      SET RECURSION FLAG
          ST    13,SYNADSAV+4       BACK CHAIN SAVE AREAS
          LA    13,SYNADSAV         ESTABLISH NEW SAVE AREA
           .
          . [this exit routine is not intended to be reentrant]
           .
          TERROR VERBATIM,MF=(E,(3)) GENERATE DIAGNOSTIC MESSAGE
          LTR   15,15               MESSAGE RETURNED?
          BNZ   SKIPWTO             IF NOT, SKIP WTO
          LR    4,0                 SAVE WTO LIST ADDRESS
          XR    0,0
          USING TEM,4
          WTO   MF=(E,TEMWTO)       WRITE MESSAGE TO OPERATOR
          L     0,TEMSL             LOAD LENGTH AND SUBPOOL
          FREEMAIN R,LV=(0),A=(4)  RELEASE MESSAGE STORAGE AREA
          DROP  4
SKIPWTO   DS    0H
          .
          . [perform recovery processing]
           .
RETURN    L     13,4(,13)           RESTORE PREVIOUS SAVE AREA ADDRESS
          LR    14,5                RESTORE RETURN ADDRESS
          XR    15,15               UPDATE GENERAL RETURN CODE
          XR    0,0                 CLEAR CONDITIONAL COMPLETION CODE
RECURSIV  DS    0H
          BR    14                  RETURN TO THE API
          LTORG
           .
           .
           .
SYNADSAV  DS     18F                 EXIT ROUTINE SAVE AREA
          TDSECT TEM                 GENERATE TEM DSECT
```

Asynchronous Exit Routines

Asynchronous exit routines, in contrast to synchronous exit routines, do not act as extensions to the part of the application program that was executing when the event associated with the exit routine occurred. The events that cause invocation of asynchronous exit routines are unpredictable, whereas synchronous exit routines can be invoked only at predictable points, for instance, immediately after the associated TCHECK or TPL-based macro instruction.

Asynchronous exit routines can interrupt the mainline program at any time, even if the mainline program is currently suspended (for example, because it issued a TCHECK or WAIT macro instruction). However, no API-invoked asynchronous exit routine can interrupt another asynchronous exit routine; thus each asynchronous exit routine must return to the API before the next asynchronous exit routine can be given control.

When an asynchronous event occurs, the associated exit routine (if defined by the application program) is scheduled by the API by being put on a transport user exit queue. If the mainline program is currently in control, execution of the mainline program is suspended and control is immediately given to the exit routine. If another asynchronous exit routine is in control, that exit routine must return to the API before the next exit routine on the user exit queue can be given control. If the asynchronous exit routine currently in control suspends execution (for example, by issuing TCHECK or WAIT), it prevents other asynchronous exit routines from gaining control. When the final asynchronous exit routine returns to the API (the transport user exit queue is now empty), the mainline program resumes control at the point where it was interrupted.

The API schedules exits only for the API events, and is unaware of asynchronous exits scheduled by other program products or the operating system. Therefore, the serialization of asynchronous exits described here only applies to the API exits, and only applies to exits for one transport user (in other words, task).

Example

The API asynchronous exits can be interrupted by VTAM exits, and VTAM asynchronous exits can be interrupted by the API exits. Also, the API and VTAM exits can be interrupted by asynchronous exits scheduled by MVS (for example, STIMER exits).

The application program must implement the appropriate serialization mechanisms when common data areas may be accessed by different asynchronous exit routines that are interruptible by one another.

Register Information

When an asynchronous exit routine is entered, the general-purpose registers contain this information:

R0	Unpredictable
R1	Address of TPL or TXP
R2-12	Unpredictable
R13	Unpredictable
R14	The API return address
R15	Address of exit routine

Asynchronous Exit Routine Coding Procedures

These procedures should be followed when coding an asynchronous exit routine:

- Asynchronous exit routines must return control with a BR 14 after register 14 has been restored with the address it contained when the exit routine was entered.
- Except for general-register 14, the exit routine is not required to save the API registers; register 13 does not contain the address of an API or application program save area.
- If the exit routine issues any TPL-based macro instructions or calls any external routines, it must provide the address of its own 18-word save area in register 13.
- The AOPEN or ACLOSE macro instruction cannot be executed in an asynchronous exit routine (the API or otherwise). The AOPEN or ACLOSE macro instruction must always be executed from the mainline program.
- Care must be taken not to reuse save areas still in use by other parts of the application program. For example, the save area used by the mainline program when a TPL-based macro instruction is issued is in use until the API returns to the mainline program; it should not be used by an asynchronous exit routine in the meantime.
- If the exit routine issues a TPL-based macro instruction and completion is awaited in the same routine (for example, by the use of TCHECK or WAIT) the mainline program, as well as the exit routine, is suspended until the requested operation is completed. To avoid such delays, consider using a TPL exit routine for notification of completion.

Since asynchronous exit routines are serialized by the API, they are not normally re-entered. However, if the same asynchronous exit routine is shared by two or more tasks, it must be coded to be reentrant. In this case, save areas and work areas should be allocated dynamically.

Asynchronous Exit Routine Example

Here is an example of an asynchronous exit routine. In this example, the exit routine is entered when data is available to be received. The exit routine issues a TRECV macro instruction in asynchronous mode and returns to the API. The TPL completion exit is also shown:

```
*************************************************************************
*  ASYNCHRONOUS EXIT ROUTINE TO HANDLE DATA INDICATIONS
*************************************************************************
          USING DORECV,12
DORECV    LR    12,15               ESTABLISH BASE ADDRESS
          LR    2,1                 MOVE TXP ADDRESS
          LR    3,14                SAVE RETURN ADDRESS
          LA    13,SAVEAREA         ESTABLISH SAVE AREA
          USING TXP,2
          L     4,TXPEPID           LOAD ENDPOINT ID
          LA    5,RECVTPL           LOAD RECEIVE TPL ADDRESS
          USING TPL,5
          TRECV EP=(4),MF=(E,(5))   INITIATE RECEIVE REQUEST
          LTR   15,15               REQUEST ACCEPTED?
          BZ    RECVEXIT            IF SO, RETURN TO API
RECVERR   DS    OH
           .
          . [perform error recovery processing]
           .
          DROP  2
          USING RECVDONE,15
*************************************************************************
* TPL EXIT ROUTINE FOR TRECV SERVICE REQUEST
*************************************************************************
RECVDONE L      12,=A(DORECV)       ESTABLISH COMMON BASE ADDRESS
          DROP  15
          LR    5,1                 MOVE TPL ADDRESS
          LR    3,14                SAVE RETURN ADDRESS
          LA    13,SAVEAREA         ESTABLISH SAVE AREA
          TCHECKMF=(E,(1))          CHECK TPL AND SET INACTIVE
          LTR   15,15               RECEIVE SUCCESSFUL?
          BNZ   RECVERR             IF NOT, GO HANDLE ERROR
           .
          . [received data is now available for processing]
           .
RECVEXIT LR     14,3                RESTORE RETURN ADDRESS
          BR    14                  RETURN TO API
          DROP  5,12
          LTORG
           .
           .
           .
RECVTPL  TRECV  DABUF=DATAAREA,DALEN=L'DATAAREA,                        +
                OPTCD=ASYNC,EXIT=RECVDONE,MF=L
*                                   TPL FOR RECEIVING DATA
DATAAREA  DS    XL1024              RECEIVE DATA BUFFER
SAVEAREA  DS    18F                 EXIT ROUTINE SAVE AREA
          TDSECT TXP,TPL            GENERATE DSECTS
```

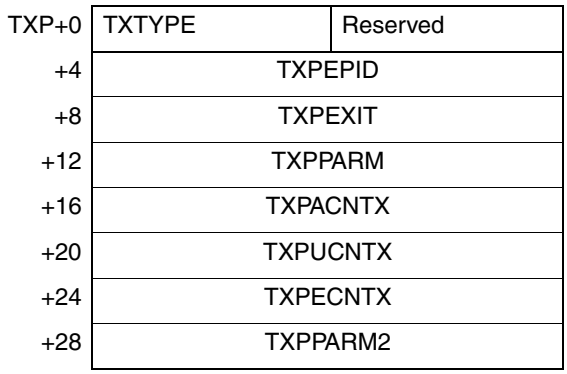
Exit Routine Parameter List

The value passed to an exit routine in register 1 is always the address of a parameter list. For the SYNAD, LERAD, and TPL exit routines, this parameter list is the TPL which is in error or has just completed. For all other exit routines, a common API data structure is used to pass exit routine parameters. This data structure is defined by the TXP dsect (read the *Cisco IOS for S/390 Assembler API Macro Reference* for more information).

Transport Endpoint Exit Parameters (TXP)

This diagram illustrates the transport endpoint exit parameters (TXP):

Figure 3-6 Transport Endpoint Exit Parameters:



TXP Information

The TXP dsect can be generated by the TDSECT macro instruction.

This table describes the contents of the TXP:

Table 3-3 TXP Information

TXPTYPE	A halfword integer value indicating the type of exit. The exit type defines the contents of the TXPPARM field.
TXPEPID	Identifies the endpoint associated with the event. If there is no endpoint associated with the event (for example, TPEND exit), the value of this field is zero.
TXPEXIT	Contains the entry point address of the exit routine. The API uses this field to schedule the exit routine.
TXPPARM	Contains a fullword parameter that is exit-dependent. For protocol event exits, the parameter identifies a particular protocol event. For the TPEND exit, this field contains a reason code.
TXPACNTX	Contains a word of context associated with the application program. This value is copied from the APCB and is specified by the ACNTX keyword on the APCB macro instruction.
TXPUCNTX	Contains a word of context associated with the endpoint. This value is copied from an internal data structure and is equal to the value specified by the UCNTX keyword on the TOPEN macro instruction.
TXPECNTX	Contains a word of context associated with the language environment. If the language environment is assembler language, this value is equal to the value specified by the ECNTX keyword on the APCB macro instruction.
TXPPARM2	Contains additional exit-dependent information. In the case of data indication, it is the amount of available window space.

The TXP is dynamically formatted in key-0 storage and cannot be modified by an exit routine executing with its normal private-area key.

How Exit Routines Are Used

This subsection summarizes all API exit routines and provides an exit routine register usage summary. Usage of each exit routine is also discussed.

Exit Routine Summary

This table lists each API exit, the type of exit, how and when it is specified, and the purpose for which it is used:

Table 3-4 Exit Routine Summary

Exit	Type	How Specified*	When Specified	Purpose
CONFIRM	Asynchronous	TEXLST or TEVNTLST	AOPEN or TOPEN	Used to receive a confirm indication. Exit routine should respond by issuing a TCONFIRM macro instruction.
CONNECT	Asynchronous	TEXLST or TEVNTLST	AOPEN or TOPEN	Used to receive a connect indication. Exit routine should respond by issuing a TLISTEN macro instruction.
DATA	Asynchronous	TEXLST or TEVNTLST	AOPEN or TOPEN	Used to receive a normal data (COTS) or datagram (CLTS) indication. Exit routine should respond by issuing a TRECV or TRECVFR macro instruction.
DGERR	Asynchronous	TEXLST or TEVNTLST	AOPEN or TOPEN	Used to receive a datagram error indication. Exit routine should respond by issuing a TRECVERR macro instruction.
DISCONN	Asynchronous	TEXLST or TEVNTLST	AOPEN or TOPEN	Used to receive a disconnect indication. Exit routine should respond by issuing a TCLEAR macro instruction.
LERAD	Synchronous	TEXLST	AOPEN only	Used to recover from program logic errors that typically occur during program debugging.
RELEASE	Asynchronous	TEXLST or TEVNTLST	AOPEN or TOPEN	Used to receive a release indication. Exit routine should respond by issuing a TRELACK macro instruction.
SENDWIND	Asynchronous	TEXLST or TEVNTLST	AOPEN or TOPEN	Used to notify MODE=SOCKETS endpoints that the send window has opened.
SYNAD	Synchronous	TEXLST	AOPEN only	Used to recover from physical errors and exceptional conditions.
TPEND	Asynchronous	TEXLST or TEVNTLST	AOPEN or TOPEN	Called when the transport provider or the API is stopped or terminated. Exit routine should initiate shutdown procedures.
APEND	Asynchronous	TEXLST	AOPEN only	Called when the API is stopped or terminated. Exit routine should initiate shutdown procedures.
TPL	Asynchronous	EXIT operand on TPL-based macro instruction	Any TPL- based service request	Called when the requested operation is complete. Exit routine should issue TCHECK macro instruction to set TPL inactive.

Exit	Type	How Specified*	When Specified	Purpose
XDATA	Asynchronous	TEXLST or TEVNTLST	AOPEN or TOPEN	Used to receive an expedited data indication. Exit routine should respond by issuing a TRECV macro instruction.

Note * Any exit that can be specified on a TOPEN TEXLST can be specified on a TOPEN TEVNTLST as well.

Register Usage Summary

This table summarizes the register usage for each type of exit routine

Table 3-5 Register Usage Summary:

	Contents Of General-Purpose Registers On Entry To Exit Routine					
Exit	**Register 0**	**Register 1**	**Register 2-12**	**Register 13**	**Register 14**	**Register 15**
CONFIRM	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine
CONNECT	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine
DATA	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine
DGERR	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine
DISCONN	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine
LERAD	Recovery action code	Address of TPL	Unmodified	Unmodified	API return address	Address of exit routine
RELEASE	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine
SENDWIND	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine
SYNAD	Recovery action code	Address of TPL	Unmodified	Unmodified	API return address	Address of exit routine
TPEND	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine
APEND	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine
TPL	Unpredictable	Address of TPL	Unpredictable	Unpredictable	API return address	Address of exit routine
XDATA	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine

Note For APEND exits, the TXPUCNTX field is not set; the TXPAPCB field is set.

TPL Completion Exit

A TPL exit routine is entered after a TPL-based operation completes if the TPL or the macro instruction using it specified the exit routine address in the EXIT operand. Specifying an exit routine forces the synchronization mode to be asynchronous (OPTCD=ASYNC).

The TPL exit routine is entered with the address of the TPL in register 1. The TPL is marked complete, but remains active until a TCHECK macro instruction is executed. The TCHECK macro instruction may be issued by the TPL exit routine, or later by posting an ECB on which the mainline program is waiting. If the TCHECK macro instruction is issued before the TPL exit routine is entered, TCHECK returns with an error.

If the application program identified a SYNAD or LERAD exit routine in the exit list provided to AOPEN, the appropriate exit routine is entered if the TPL service request was completed with an error. Otherwise, the TPL exit routine should check register 15 returned by the TCHECK macro instruction, or check the return code fields of the TPL to determine if the request was completed successfully.

The exit routine may initiate additional API service requests using the same TPL, or may supply a different TPL. If additional TPL-based requests are executed, it is best that the exit routine not wait for completion, otherwise entry into other asynchronous exit routines may be delayed. Newly initiated requests should specify posting of an ECB or a TPL exit routine which, in the latter case, may be the same exit routine. When processing is complete, control must be returned to the API by branching to the address contained in register 14 when the exit routine was entered.

Protocol Event Exits and ECBs

Certain protocol events can be handled asynchronously by the application program. These protocol events generally correspond to T.indication or T.confirm primitives issued by the transport provider.

All protocol event exits are entered with register 1 containing the address of a TXP. The TXP identifies the exit type (TXPTYPE) as a protocol event (TXPTPROT), TXPEPID contains the endpoint ID for the endpoint associated with the protocol event, and TXPPARM contains a fullword value identifying the particular event. The event code is in multiples of four and can be used as a branch or table index to locate the appropriate processing routine.

These protocol event codes are defined:

Table 3-6 Protocol Event Codes

Name	Dec	Hex	Exit	Protocol Event
TXPECONN	0	X’00’	CONNECT	Connect indication
TXPECONF	4	X’04’	CONFIRM	Confirm indication
TXPEDATA	8	X’08’	DATA	Normal data (or datagram) indication
TXPEXPDT	12	X’0C’	XDATA	Expedited data indication
TXPERROR	16	X’10’	DGERR	Datagram error indication
TXPEDISC	20	X’14’	DISCONN	Disconnect indication
TXPERLSE	24	X’18’	RELEASE	Orderly release indication
TXPESWND	28	1C	SENDWIND	Send Window Open

The event code is used by the API to determine which exit routine address to load from the exit list. Therefore, the application program can supply a different exit routine for each protocol event, or supply a common exit routine that handles all events, or a particular subset of events. If no exit routine is specified in the TOPEN or AOPEN exit list, the application program cannot receive asynchronous notification of the corresponding protocol event.

A protocol event exit routine generally issues an appropriate TPL-based macro instruction to clear the pending indication. Since the indication serves to notify the application program that some awaited event has occurred, it is generally safe to issue the responding macro instruction in synchronous mode without fear of suspending the exit routine for an indefinite period of time. However, the exit routine must be suspended long enough for the API address space to complete the service request. Therefore, for optimum performance, it may be better to execute the request in asynchronous mode and supply a TPL exit routine to handle the subsequent completion.

The exit routine is not obliged to clear the pending indication in the exit routine. Instead, an ECB could be posted on which the mainline program is waiting. When the exit routine completes its processing, control must be returned to the API by branching to the address contained in register 14 when the exit routine was entered.

CONNECT Event

The CONNECT exit routine is scheduled or the ECB is posted when a connect indication is generated and no TLISTEN service request is outstanding (in other words, the TLISTEN request has been issued and is awaiting completion). A connect indication is generated by the API on receiving a T-CONNECT.indication primitive issued by the transport provider. Connect indications can only be generated on endpoints in the enabled (TSENABLD) or connect-indication-pending (TSINCONN) state.

The CONNECT exit routine should execute a TLISTEN macro instruction to receive the connect indication. Once a connect indication has been received with TLISTEN, it is said to be pending until accepted or rejected with a TACCEPT or TREJECT macro instruction. The exit routine is not obliged to issue the TLISTEN request, and may defer this responsibility to the mainline program by posting an ECB. Similarly, the TACCEPT or TREJECT macro instruction may be issued by the mainline program or in a subsequent TPL completion exit to prevent suspension of the CONNECT exit routine.

Once the CONNECT exit routine has been scheduled, it is not rescheduled until all available connect indications have been received by a TLISTEN macro instruction. If another T-CONNECT.indication is issued by the transport provider, it is queued without rescheduling the CONNECT exit routine. Therefore, once the CONNECT exit routine has been entered, sufficient TLISTEN service requests should be issued to receive all of the indications that have been queued. The COUNT field (TPLCOUNT) returned in the TPL by TLISTEN indicates the number of additional connect indications to be received. Also, a bit (TOMORE) is set in the option code field (TPLOPCD2) to indicate more connect indications are available. The application program should receive and accept or reject connect indications quickly since a limited number may be queued (see the QLSTN parameter on the TBIND macro instruction in the *Cisco IOS for S/390 Assembler API Macro Reference*).

When associations are being used with an endpoint operating in connectionless mode (CLTS), datagrams arriving from a new source address causes a connect indication to be simulated. The datagram is queued until the connect indication is received and accepted. Thus, the CONNECT exit routine can be used to form CLTS associations in the same manner it is used to establish COTS connections.

CONFIRM Event

The CONFIRM exit routine is scheduled when a confirm indication is generated and no TCONFIRM service request is outstanding. A confirm indication is generated by the API on receiving a T-CONNECT.confirm primitive issued by the transport provider. Confirm indications can only be generated on endpoints in the connect-in-progress (TSOUCONN) state (in other words, a TCONNECT macro instruction has been successfully executed at the endpoint and the application program is awaiting connect confirmation).

The CONFIRM exit routine should execute a TCONFIRM macro instruction to receive the confirm indication.

If the TCONFIRM macro instruction completes successfully, the connection has been established and the state of the endpoint becomes connected (TSCONNCT).

If the application program no longer desires to establish the connection, a TDISCONN macro instruction should be executed instead.

The exit routine is not obliged to receive (or clear) the confirm indication and may defer this responsibility to the mainline program by posting an ECB on which it is waiting.

A confirm indication can only occur as the result of a successful TCONNECT service request. Therefore, only one confirm indication can be received on a given endpoint until the connection is released, and another TCONNECT request is issued. If the endpoint is operating in connectionless mode with associations, a confirm indication is simulated when the TCONNECT request is issued. Thus, the CONFIRM exit routine can be used to form CLTS associations in a manner similar to COTS connections.

DATA Arrival Event

The DATA exit routine is scheduled when a normal data indication is generated and no TRECV service request is outstanding. A normal data indication is generated by the API on receiving a T-DATA.indication primitive from the transport provider. Normal data indications can only be generated on endpoints in the connected (TSCONNCT) or release-in-progress (TSOURLSE) state.

The DATA exit routine should execute a TRECV macro instruction to receive the data buffered for the endpoint. Alternatively, the DATA exit routine may defer this responsibility to the mainline program by posting an ECB on which it is waiting. However, optimum performance is achieved by receiving the data as soon as possible.

Once the DATA exit routine has been scheduled, it is not rescheduled until all available data has been received by the application program. If another T-DATA.indication primitive is issued by the transport provider and some previous data has not yet been received, new data is buffered at the endpoint without rescheduling the DATA exit routine. Therefore, once the DATA exit routine has been entered, sufficient TRECV service requests should be issued to receive all of the available data. A bit (TOMORE) is set in the option code field (TPLOPCD2) of the TPL to indicate more data is available to be received (see the description of the TRECV macro instruction in the Cisco IOS for S/390 Assembler API Macro Reference).

Note Expedited data may arrive at the endpoint and be received by subsequent TRECV macro instructions. A bit (TOEXPDTE) is set in the option code field (TPLOPCD2) to indicate expedited data was received. Read XDATA Event for more information on expedited data.

A CLTS endpoint operates in a similar fashion; the DATA exit routine is scheduled when a datagram indication is generated and no TRECVFR service request is outstanding. In this case, a datagram indication is generated by the API on receiving a T-UNITDATA.indication primitive from the

transport provider. Datagram indications can only be generated on CLTS endpoints that are in the disabled (TSDSABLD) state. Once the DATA exit routine has been scheduled, it is not rescheduled until all buffered datagrams have been received by the application program.

DGERR Event

The DGERR exit routine is scheduled whenever the transport provider indicates to the API that it could not deliver a datagram previously transferred with a TSENDTO service request. The transport provider detected the error after the TSENDTO request was completed successfully. Datagram error indications can only occur on CLTS endpoints that are in the disabled (TSDSABLD) state.

The DGERR exit routine should execute a TRECVERR macro instruction to receive the protocol address and protocol options associated with the datagram in error. A protocol-dependent datagram error code is also returned (read Cisco IOS for S/390 Assembler API Macro Reference for more information).

Note Connectionless-mode service is unreliable. The fact that the DGERR exit routine is not entered does not imply that a particular datagram was delivered successfully. Generally, errors that cause the DGERR exit routine to be entered are detected locally before the datagram is transmitted onto the network, such as an invalid protocol address.

DISCONN Event

The DISCONN exit routine is scheduled in response to receiving a T-DISCONNECT.indication primitive from the transport provider. A disconnect indication can be generated for a COTS endpoint that is in one of these states:

- Connect-indication-pending (TSINCONN)
- Connect-in-progress (TSOUCONN)
- Connected (TSCONNCT)
- Release-indication-pending (TSINRLSE)
- Release-in-progress (TSOURLSE)

The disconnect indication serves to notify the application program that an established connection has been released or a connection attempt has been abandoned. The exit routine should execute a TCLEAR macro instruction that returns the endpoint to the enabled or disabled state. Alternatively, the exit routine may post an ECB waited on by the mainline program, and the mainline program can issue the TCLEAR macro instruction. The TCLEAR macro instruction receives information associated with the disconnect indication, including disconnect user data and a protocol-dependent disconnect reason code.

Only one disconnect indication can be generated per established connection or connection attempt. A disconnect indication is never generated for a CLTS endpoint.

RELEASE Event

The RELEASE exit routine is scheduled when a release indication is generated and no TRELACK service request is outstanding. A release indication is generated by the API on receiving a T-RELEASE.indication primitive from a transport provider that supports orderly release of a connection. Release indications can only be generated on endpoints in the connected (TSCONNCT) or release-in-progress (TSOURLSE) state.

The RELEASE exit routine should execute a TRELACK macro instruction to acknowledge the release indication. Alternatively, the exit routine may post an ECB on which the mainline program is waiting, and the mainline program can acknowledge the release indication.

The RELEASE exit routine is not scheduled until all available data has been received by the application program. After the release indication has been received, no more TRECV macro instructions should be executed at the endpoint. However, the endpoint can continue to send data until a TRELEASE macro instruction has been executed.

Only one release indication is generated per established connection, and can only be generated on endpoints for which the orderly release option was selected. If the endpoint is operating in connectionless mode using associations, the optional orderly release service is automatically selected. In this case, a release indication is simulated when the application program executes a TRELEASE macro instruction.

Send Window Opened Protocol Event

The Send Window Opened protocol event is scheduled when an endpoint is defined with MODE=SOCKETS and the event is defined in the TEVENTLST or TEXLST macro. The exit may be specified on the TEVNTLST or TEXLST macro, where SENDWIND=*parameter*. The ECB may be specified on the TEVNTLST macro, where SENDWIND=(<*addr*>, ECB).

After a series of TSEND or TSENDTO requests have used all of the available send-buffer space, the Send Window Opened protocol event is armed. The protocol event is triggered when send buffer space becomes available. TCP acknowledgment (ACK) of sent data causes buffer space to become available. In the case of UDP, buffer space is released as soon as datagrams using the buffer space are placed on the network.

XDATA Event

The XDATA exit routine is scheduled when an expedited data indication is generated and no TRECV service request is outstanding. An expedited data indication is generated by the API on receiving a T-EXPEDITED-DATA.indication primitive from the transport provider. Expedited data indications can only be generated on COTS endpoints in the connected (TSCONNCT) or release-in-progress (TSOURLSE) state.

The XDATA exit routine should execute a TRECV macro instruction to receive the data buffered for the endpoint. Alternatively, the XDATA exit routine may defer this responsibility to the mainline program by posting an ECB on which it is waiting. However, optimum performance is achieved by receiving the data as soon as possible.

Once the XDATA exit routine has been scheduled, it is not rescheduled until all available data has been received by the application program. If another T-EXPEDITED-DATA.indication primitive is issued by the transport provider and some previous data has not yet been received, new data is buffered at the endpoint without rescheduling the XDATA exit routine. Therefore, once the XDATA exit routine has been entered, sufficient TRECV service requests should be issued to receive all of the available data. A bit (TOMORE) is set in the option code field (TPLOPCD2) to indicate more data is available to be received (read Cisco IOS for S/390 Assembler API Macro Reference for more information).

For the purpose of scheduling exit routines, expedited data is treated as normal data if no XDATA exit routine has been specified. If the DATA and XDATA exit routines have both been specified, an expedited data indication is generated if no expedited data is available to be received and new expedited data arrives. However, if expedited data is pending, newly arriving normal data does not generate a normal data indication, and the DATA exit routine is not scheduled.

Scheduling of DATA and XDATA Exit Routines

This table summarizes when the DATA and XDATA exit routines are scheduled:

Note The column labeled **Data Pending** indicates whether or not data of the specified type is already available when new data arrives at the endpoint. Expedited data is not supported for CLTS endpoints operating in association mode.

Table 3-7 DATA and XDATA Exit Routines

Exit Routine Specified		Data Pending			Exit Routine
Data	**Xdata**	**Normal**	**Expedited**	**New Data**	**Scheduled**
No	No	No/Yes	No/Yes	Any	None
No	Yes	No/Yes	No/Yes	Normal	None
		No/Yes	No	Expedited	XDATA
		No/Yes	Yes		None
Yes	No	No	No	Any	DATA
		No	Yes		None
		Yes	No		None
		Yes	Yes		None
Yes	Yes	No	No	Normal	DATA
		No	Yes		None
		Yes	No		None
		Yes	Yes		None
		No	No	Expedited	XDATA
		No	Yes		None
		Yes	No		XDATA
		Yes	Yes		None

SYNAD/LERAD – Synchronous Error Recovery Exits

The API supports two synchronous error recovery exit routines:

- The SYNAD exit routine

  Used to handle physical errors and exceptional conditions

- The LERAD exit routine

  Used to handle program logic errors

If the appropriate error recovery exit routine is not provided by the application program, the detection and recovery from such errors must be handled inline with the macro instruction. The SYNAD and LERAD exit routines are specified in the AOPEN exit list and apply to all endpoints opened by a given transport user (in other words, the application program task).

The SYNAD or LERAD exit routine can be entered at two different points during the execution of any TPL-based service request:

- If the initial request for the operation fails (in other words, after it is rejected)

- If the request is accepted, after the operation fails (in other words, after it is completed abnormally)

If the TPL-based macro instruction is executed in asynchronous mode, then the first point occurs just prior to returning control to the next sequential instruction following the TPL-based macro instruction, and the second point occurs just prior to returning control to the next sequential instruction following the corresponding TCHECK macro instruction.

If the TPL-based macro instruction is executed in synchronous mode, then the TCHECK function is embedded, and the application program cannot distinguish at which point entry was made.

In either case, entry into a SYNAD or LERAD exit routine is made no more than once for any given instance of a TPL-based macro instruction.

The SYNAD or LERAD exit routine is entered with the TPL address in register 1, and a recovery action code in register 0. The recovery action code can be used to determine error recovery procedures. This is the same recovery action code that is returned to the application program if a SYNAD or LERAD exit routine is not specified. The value of the recovery action code also determines which exit routine is entered. Therefore, the SYNAD and LERAD exit list addresses can specify a common exit routine, or separate exit routines tailored for their individual use.

Register 14 contains an address in the API that the exit routine should branch to if it desires to return control to the next sequential instruction in the application program. If control is returned, the contents of registers 0 and 15 is passed through to the application program, and the contents of registers 2-12 is restored from the save area whose address is in register 13. The exit routine should take care not to modify the contents of the save area or the save area address. If the exit routine requires its own save area, one should be allocated, and the contents of register 13 should be restored before returning to the API.

On entry to the SYNAD or LERAD exit routine, registers 2-12 contain the application program's general-purpose registers at the time the initial request was issued or when the TCHECK macro instruction was executed, depending at which point the exit routine was called. The exit routine can take advantage of this fact if it knows how those registers are used by the application program. The exit routine is free to use the registers for any purpose without restoring them before returning to the API.

If the TCHECK or TPL-based macro instruction that caused entry into the SYNAD or LERAD exit routine used one of the general registers between 2 and 12 inclusive to contain the TPL address, bit 0 of that register is copied into bit 0 of register 1. If the mainline program assures that bit 0 is always reset when issuing a TPL-based request, and the exit routine assures bit 0 is set when issuing such a request, recursion can be detected by testing register 1. Any SYNAD or LERAD exit routine that issues TPL-based requests should test for recursion or risk entering an endless loop.

SYNAD/LERAD Errors

There are some circumstances when the SYNAD or LERAD exit routine cannot be entered. This happens when a fatal error occurs before the proper environment can be established to call the exit routine. In this case, control is always returned to the next sequential instruction in the application program. The general return code in register 15 is greater than 4, indicating that a fatal error occurred. This usually occurs before a TPL-based request is accepted, but can also occur on a TCHECK macro instruction if the TPL or some internal control block has become corrupted.

These are examples of fatal errors that prevent scheduling of the SYNAD or LERAD exit routine:

- An invalid function code was detected
- The TPL does not contain a valid control block identifier, or resides in store-protected memory
- The TPL does not contain a valid endpoint ID

- The APCB or internal API control blocks have become corrupted
- The APCB is closed

If control is returned to the API, register 15 should contain a general return code which the application program can test for success or failure. If the exit routine was able to recover from the error, a 0 should be returned to indicate success. This appears to the application program as if no error occurred. Otherwise, a non-zero value should be returned. We recommend that the exit routine use values consistent with the design of the API.

If the exit routine chooses not to return, control continues as if there were a branch to the exit routine immediately following the TCHECK or TPL-based macro instruction. However, this is permitted only if the TCHECK or TPL-based macro instruction was executed by the mainline program, or an extension thereof. If the SYNAD or LERAD exit routine was entered from a macro instruction executed by an asynchronous exit routine, be careful to always return to the API so the asynchronous exit routine can also return control.

LERAD Exit Routine

The LERAD exit routine is entered when a program logic error occurs. Errors of this type are generally detected early before a TPL-based request is accepted by the API. The recovery action code in register 0 is set to one of these values:

Table 3-8 LERAD Exit Routine Recovery Action Codes

Name	Dec	Hex	Type Of Error
TAFORMAT	16	X'10'	Format or specification error
TAPROCED	20	X'14'	Sequence or procedural error
TATPLERR	24	X'20'	Logic error with no TPL return code

TAFORMAT and TAPROCED errors enter the LERAD exit routine with the TPL set inactive and the TPLRTNCD field containing valid return-code information. The specific error code (TPLERRCD) and diagnostic code (TPLDGNCD) contain additional information regarding the error.

If the recovery action code is set to TATPLERR, the TPL is still active and return-code information could not be stored. This recovery action code can occur as the result of one of these different circumstances:

- A TPL-based request was issued and the associated TPL was active
- A TCHECK macro instruction was executed before the TPL was marked complete.

The second circumstance can only occur for asynchronous requests for which a TPL exit routine has been specified (in other words, the TCHECK macro instruction was executed before the exit routine was entered). For this recovery action code there is no way for the exit routine to differentiate the cause of the error.

Program logic errors typically occur while an application program is being developed and debugged. Hopefully, once debugging is complete, errors of this type do not occur. Therefore, it is unusual for a LERAD exit routine to attempt to recover from such errors. The best course of action is to dump the address space and terminate the application program.

SYNAD Exit Routine

The SYNAD exit routine is entered when a physical error or exceptional condition occurs. Unlike program logic errors, which tend not to occur after a program has been debugged, these errors can happen at any time and often occur after a TPL-based request has been accepted by the API.

The recovery action code in register 0 is set to one of these values:

Table 3-9 SYNAD Exit Routine Recovery Action Codes

Name	Dec	Hex	Type Of Error
TAEXCPTN	4	X'04'	Exceptional condition
TAINTEG	8	X'08'	Connection or data integrity error
TAENVIRO	12	X'0C'	Environmental error

The SYNAD exit routine is always entered with the TPL set inactive and the TPLRTNCD field containing valid return-code information stored by the API. A copy of the recovery action code is stored in the TPL (TPLACTCD) along with a specific error code (TPLERRCD) and a diagnostic code (TPLDGNCD). This information can be used to determine more precisely the particular cause of the error.

The SYNAD exit routine may choose to merely log the occurrence of an error and record pertinent diagnostic information, or may attempt to recover from the error. If necessary, additional TPL-based macro instructions may be issued to affect recovery. However, the exit routine should implement an appropriate mechanism (see SYNAD/LERAD – Synchronous Error Recovery Exits) to detect recursion caused by reentry into the exit routine. If recovery from the error is successful, registers 15 and 0 can be set accordingly so that the mainline program is unaware of the occurrence of the error, and processing can continue as usual.

TPEND Exit Routine or ECB

The TPEND exit routine is entered when the transport provider terminates (or is about to terminate) and can no longer provide service to the application program. The TPEND exit routine is entered asynchronously with register 1 containing the address of a TXP. The TXP identifies the exit type (TXPTYPE) as a TPEND exit (TXPTPEND), the endpoint ID (TXPEPID) is set to zero since no endpoint is associated with this event, and the exit routine parameter (TXPPARM) gives the reason for the termination. The reason code is in multiples of 4 and can be used as a branch or table index to locate the appropriate processing routine.

These reason codes are defined:

Table 3-10 TPEND Exit Routine Reason Codes

Name	Dec	Hex	Reason For Termination
TXPRDRAN	0	X'00'	Operator drained subsystem
TXPRSTOP	4	X'04'	Operator stopped subsystem
TXPRTERM	8	X'08'	Subsystem abnormally terminated

Entering the TPEND Exit Routine

The TPEND exit routine is normally entered three times.

- The first time occurs when the system operator enters a command from the operator's console to drain the API subsystem. This command is entered in anticipation of shutting down the subsystem. Existing transport service users are allowed to continue normal operation, but new transport users are not allowed to establish sessions with the transport provider (in other words, AOPEN macro instructions are completed with an error (APCBEDRA)).

  When the TPEND exit routine is entered with the reason code set to TXPRDRAN, a variable should be set to prevent the application program from attempting to define any new transport service users (in other words, prevent AOPEN macro instructions from being issued). If the application program is serving interactive users, they should be informed by whatever means are appropriate that the network subsystem may be shut down soon, and should be encouraged to complete their use of any network resources accessed via the API. This entry into the TPEND exit routine only serves as a warning that a shutdown of the subsystem is about to happen. Any TPL-based request issued after the TPEND exit routine is entered are completed conditionally (in other words, if no other errors occur and the function is allowed in drain mode, the request is completed normally with TRSTOP set in the conditional completion code returned in the TPL (and register 0)). Any request that is not allowed in drain mode is completed with TRFAILED/TEDRAIN.

- The second entry into the TPEND exit routine occurs after the TPEND exit has been entered in drain mode for all endpoints. In this case, the shutdown process has actually begun, and the application program has a limited time to complete its use of the API services. No new endpoints can be opened, but existing endpoints are allowed to continue issuing certain TPL-based service requests that are required to shutdown the application. Any allowable TPL-based request issued after the TPEND exit routine is entered is completed conditionally (in other words, if no other errors occur, the request is completed normally with TCSTOP set in the conditional completion code returned in the TPL (and register 0)). Any request that is not allowed during the stop phase is completed with TRFAILED/TESTOP.

- The third and last entry into the TPEND exit routine occurs when the API subsystem actually terminates, either normally or abnormally. A second stop command may be issued to force this phase of termination. At this point the application program is only allowed to clean up resources in its address space by issuing TCLOSE and ACLOSE macro instructions. In the case of a graceful shutdown, the application program should have already closed its endpoints and terminated its session with the API by the time subsystem termination occurs.

The TPEND exit is not guaranteed to be entered all three times.

Example

If the system operator stops the API subsystem without first issuing a drain command, the first entry does not occur. Similarly, if the subsystem is canceled or abnormally terminates without a stop command, the second entry does not occur.

If the application program does not specify a TPEND exit routine in its AOPEN exit list, no asynchronous notification is given for the events previously described. The application program has to rely on return-code information stored in the TPL to detect the occurrence of a shutdown or subsystem termination.

The TPEND exit routine for the API differs from the VTAM TPEND exit in that all instances of the TPEND exit routine are scheduled by the API at the same priority, whereas VTAM schedules the termination instance at a higher priority (in other words, when the reason code is X'08', VTAM schedules the TPEND exit routine so that it preempts other asynchronous exit routines). For the API, all asynchronous exit routines are executed serially, including all instances of the TPEND exit routine.

APEND Exit Routine

The APEND exit routine is entered when the application subsystem is shut down and can no longer provide service to the application program. The APEND exit routine is entered asynchronously with register 1 containing the address of a TXP. The TXP identifies the exit type (TXPTYPE) as an APEND exit (TXPAPEND). Instead of an endpoint ID TXPAPCB is set to the APCB associated with this event, and the exit routine parameter (TXPPARM) gives the reason for the termination. The reason code is in multiples of 4 and can be used as a branch or table index to locate the appropriate processing routine.

These reason codes are defined:

Table 3-11 APEND Exit Routine Reason Codes

Name	Dec	Hex	Reason For Termination
TXPRDRAN	0	X'00'	Operator drained subsystem
TXPRSTOP	4	X'04'	Operator stopped subsystem
TXPRTERM	8	X'08'	Subsystem abnormally terminated

Entering the APEND Exit Routine

The APEND exit routine is normally entered three times.

- The first time occurs when the system operator enters a command from the operator's console to drain the API subsystem. This command is entered in anticipation of shutting down the subsystem. Existing transport service users are allowed to continue normal operation, but new transport users are not allowed to establish sessions with the transport provider (in other words, AOPEN macro instructions are completed with an error (APCBEDRA)).

  When the APEND exit routine is entered with the reason code set to TXPRDRAN, a variable should be set to prevent the application program from attempting to define any new transport service users (in other words, prevent AOPEN macro instructions from being issued). If the application program is serving interactive users, they should be informed by whatever means are appropriate that the network subsystem may be shut down soon, and should be encouraged to complete their use of any network resources accessed via the API. This entry into the APEND exit routine only serves as a warning that a shutdown of the subsystem is about to happen. Any TPL-based request issued after the APEND exit routine is entered are completed conditionally (in other words, if no other errors occur and the function is allowed in drain mode, the request is completed normally with TRSTOP set in the conditional completion code returned in the TPL (and register 0)). Any request that is not allowed in drain mode is completed with TRFAILED/TEDRAIN.

- The second entry into the APEND exit routine occurs after the APEND exit has been entered in drain mode for all endpoints. In this case, the shutdown process has actually begun, and the application program has a limited time to complete its use of the API services. No new endpoints can be opened, but existing endpoints are allowed to continue issuing certain TPL-based service requests that are required to shutdown the application. Any allowable TPL-based request issued after the APEND exit routine is entered is completed conditionally (in other words, if no other errors occur, the request is completed normally with TCSTOP set in the conditional completion code returned in the TPL (and register 0)). Any request that is not allowed during the stop phase is completed with TRFAILED/TESTOP.

- The third and last entry into the APEND exit routine occurs when the API subsystem actually terminates, either normally or abnormally. A second stop command may be issued to force this phase of termination. At this point the application program is only allowed to clean up resources

in its address space by issuing TCLOSE and ACLOSE macro instructions. In the case of a graceful shutdown, the application program should have already closed its endpoints and terminated its session with the API by the time subsystem termination occurs.

The APEND exit is not guaranteed to be entered all three times.

Example

If the system operator stops the API subsystem without first issuing a drain command, the first entry does not occur. Similarly, if the subsystem is canceled or abnormally terminates without a stop command, the second entry does not occur.

If the application program does not specify an APEND exit routine in its AOPEN exit list, no asynchronous notification is given for the events previously described. The application program has to rely on return-code information stored in the TPL to detect the occurrence of a shutdown or subsystem termination.

The APEND exit routine for the API differs from the VTAM TPEND exit in that all instances of the APEND exit routine are scheduled by the API at the same priority, whereas VTAM schedules the termination instance at a higher priority (in other words, when the reason code is X'08', VTAM schedules the TPEND exit routine so that it preempts other asynchronous exit routines). For the API, all asynchronous exit routines are executed serially, including all instances of the APEND exit routine.

Deriving Context in Exit Routines

Generally, the information passed to an exit routine is sufficient for providing the context the application program needs for processing the event.

Example

The TPL address passed to SYNAD, LERAD, and TPL exit routines identifies the particular request causing entry into the exit routine.

Similarly, the endpoint ID provided in the TXP for protocol events identifies the endpoint at which the event occurred. However, when exit routines are shared between multiple tasks, additional context may be required.

Asynchronous exit routines that are provided the address of a TXP in register 1 can acquire additional context from the TXP itself.

Example

TXPACNTX is a word of context related to the transport user. It is acquired from the APCB and is specified by the ACNTX parameter. It can be an arbitrary fullword value, and is not interpreted by the API.

Similarly, TXPUCNTX is related to the endpoint and is specified when the endpoint is opened by coding the UCNTX parameter on the TOPEN macro instruction. The API merely copies this information into the TXP before the exit routine is entered.

The TXPECNTX field provides context for the language environment. If a higher-level language environment is not being used (in other words, ENVIRO=ASM is coded on the APCB macro instruction), this field can be used by the application program by specifying the ECNTX parameter on the APCB macro instruction.

For those exit routines that are not provided a TXP, a different technique may be used. In this case, the exit routine is always provided the address of a TPL. Since the application program provides the TPL, it can arrange to have the TPL located relative to other information.

Example

The address of the TPL can be decremented by a known amount to obtain the base of an application program control block (in other words, the TPL is located at some fixed offset within the control block), or the application program can place information at the end of the TPL (in other words, the TPL is extended in length to include application program information).

Handling Errors and Special Conditions

This section describes two types of TPL-based requests:

- Synchronous

For synchronous TPL-based requests, a single macro instruction is issued. On return to the application program, error or exceptional-condition information about the requested operation is available.

- Asynchronous

  For asynchronous requests, two TPL-based macro instructions are required:

  — A request macro instruction

  — A TCHECK macro instruction

  Error and exceptional-condition information can thus be returned at two different stages, as a result of the request for the operation being accepted and, if the request is accepted, as a result of the operation completing successfully or unsuccessfully.

Macro Information

Following a TPL-based macro instruction, information is available to the application program about the acceptability of the request or about the completion of the operation. This information may be provided by the API; or, if an error or exceptional condition was detected and the API invoked the program's SYNAD or LERAD exit routine, register information is provided by the exit routine. This information consists of a return code in register 15 (termed the general return code), in some cases a return code in register 0 (termed either a recovery action code or a conditional completion code), and information in the TPL. The information stored in the TPL consists of a copy of the recovery action code and conditional completion code, and when an error occurs, also consists of a specific error code and a diagnostic code.

If an error or exceptional condition occurs, it can be analyzed and handled in either the mainline program or exit routine in which the TPL-based request was issued, or in the SYNAD or LERAD exit routine designated by the AOPEN exit list. In either case, the analysis is performed by examining the return code information provided in registers and stored in the TPL. These return codes are discussed in more detail in General Return Codes.

General Return Codes

This table lists the general return codes:

Table 3-12 General Return Codes

Name	Dec	Hex	Meaning
TROKAY	0	X'00'	Request accepted or completed successfully
TRFAILED	4	X'04'	Request not accepted or completed abnormally
TRFATLFC	8	X'08'	Fatal error: invalid function code
TRFATLPL	12	X'0C'	Fatal error: invalid TPL
TRFATLAM	16	X'10'	Fatal error: internal access method error
TRFATLAP	20	X'14'	Fatal error: APCB is closed or invalid

The general return code also indicates what other information is available in register 0 and the return code field of the TPL.

Summary of Register and TPL Return Codes

This table summarizes the relationship of the general return code in register 15 and other information returned to the application program:

Table 3-13 Register and TPL Return Codes

		TPL Return Codes		SYNAD or LERAD Exit Routine
Register 15	Register 0	TPLACTCD	TPLERRCD	Entered
TROKAY X'00'	Conditional completion code	Recovery action code (X'00')	Conditional completion code	No
TRFAILED X'04'	Recovery action code or value from exit routine	Recovery action code (X'04' - X'14')	Specific error code	Yes
TRFATLFC X'08'	TPL function code	No information stored	No information stored	No
TRFATLPL X'0C'	Diagnostic code	No information stored	No information stored	No
TRFATLAM X'10'	Diagnostic code	No information stored	No information stored	No
TRFATLAP X'14'	Diagnostic code	No information stored	No information stored	No

Conditional Completion Codes

When a TPL-based request completes successfully (the general return code is zero), a conditional completion code may be returned in register 0.

- If the value returned in register 0 is zero, then the request is said to have completed normally.
- If the value returned in register 0 is non-zero, then the request is said to have completed conditionally, and the value in register 0 is the conditional completion code.

The purpose of the conditional completion code is to notify the application program of the occurrence of some condition which, although it did not prevent the successful completion of the requested operation, may need to be acted on by the application program. More than one condition may be present at one time, and as such, each bit in the conditional completion code represents a different condition.

These conditions are recognized by the API:

Table 3-14 Conditional Completion Codes

Name	Dec	Hex	Meaning
TCOKAY	0	X’00’	Normal (unconditional) completion
TCVERIFY	128	X’80’	One or more protocol options did not verify
TCNEGOT	64	X’40’	Protocol options negotiated to inferior value
TCTRUNC	32	X’20’	Data truncated to fit in storage area
TCSTOP	8	X’08’	Subsystem shutdown in progress

When a TPL-based request completes successfully, the recovery action code stored in the TPL (TPLACTCD) is set to zero (X'00'), and the conditional completion code is stored in the specific error code field (TPLERRCD). Assertion of a conditional completion code does not cause the SYNAD or LERAD exit routine to be entered.

Recovery Action Codes

Recovery action codes serve these purposes:

- They are used by the API to determine whether the SYNAD or LERAD exit routine should be entered following the unsuccessful completion of a request
- They provide the exit routine a way distinguish a class of errors and to dispatch the appropriate processing routine. These recovery actions codes are used by the API:

Table 3-15 Recovery Action Codes

Name	Dec	Hex	Meaning
TAOKAY	0	X’00’	Request completed successfully
TAEXCPTN	4	X’04’	Failed due to exceptional condition
TAINTEG	8	X’08’	Connection or data integrity error
TAENVIRO	12	X’0C’	Environmental error
TAFORMAT	16	X’10’	Format or specification error
TAPROCED	20	X’14’	Sequence or procedural error
TATPLERR	24	X’18’	Logic error with no TPL return code

Recovery Action Code Classification for Errors

Errors are classified by recovery action code as follows:

Errors with the following recovery action codes are classified as physical errors or exceptional conditions that cause the SYNAD exit routing to be entered (if one is specified):

- TAXCPTN
- TAINTEG
- TAENVIRO

Errors with the following recovery action codes are classified as program logic errors that cause the LERAD exit routine to be entered:

- TAFORMAT
- TAPROCED
- TATPLERR

TAOKAY is the recovery action code used for successful completion.

Normally, the recovery action code is stored in the TPLACTCD field of the TPL in addition to being returned in register 0. However, the TATPLERR recovery action code is a special case. For all errors of this class, the TPL is busy with another request and cannot be modified. Therefore, the recovery action code and the accompanying specific error and diagnostic codes are not stored in the TPL.

Specific Error Codes

The specific error code is used to indicate a particular error within the class of errors defined by the recovery action code. The specific error code is only returned in the TPLERRCD field of the TPL. Since return codes are not stored for the TATPLERR recovery action code, specific error codes in this class can never be returned to the application program.

Specific error codes have been carefully selected to have generic applicability across a variety of transport providers.

Example

TEPROTO is the symbolic name of the specific error code used to indicate a protocol error. This code does not indicate a specific protocol error, but only that a protocol error occurred. Therefore, application programs that make use of the specific error code can expect a degree of commonality from one transport provider to another.

Specific error codes used by the API are documented in the *Cisco IOS for S/390 Unprefixed Messages and Codes Reference*.

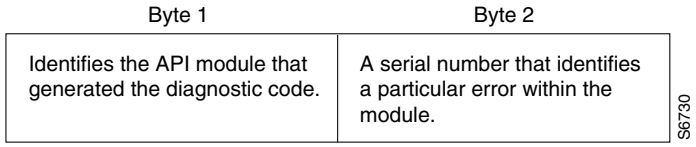
Diagnostic Codes

The diagnostic code is used to indicate a particular instance of a specific error code.

Diagnostic codes are provider-dependent, and an application program that is intended to be portable between transport providers should not analyze this code. However, it is a good idea to include the diagnostic code with any information recorded for diagnostic purposes.

The diagnostic code is four bytes in length. The first 2 bytes are the module ID, followed by a 2-byte serial number that identifies a particular error within the module:

Figure 3-7 Diagnostic Code



The four byte diagnostic code appears in API traces and is recorded in the extended diagnostic area if TPL extensions are used.

When returned to the standard TPL, the diagnostic code is uniquely mapped to a 2 byte code.

The first byte identifies the module that encountered the error. The second byte identifies a specific error instance identifier within the module.

Diagnostic codes that can accompany specific error codes are listed in the *Cisco IOS for S/390 Unprefixed Messages and Codes Reference*.

AOPEN and ACLOSE Errors

After issuing the AOPEN or ACLOSE macro instruction, register 15 should be tested.

- If the return code in register 15 is zero, the APCB has been opened or closed as requested.
- If the return code in register 15 is not equal to zero, the APCB was not properly opened or closed. When this occurs, register 0 contains an error code.

This error code may also be returned in the APCBERRC field of the APCB. Whether or not the error code is stored depends on the value returned in register 15. If the error code is stored, it is accompanied by a diagnostic code stored in the APCBDGNC field.

The API uses these AOPEN and ACLOSE return codes:

Table 3-16 AOPEN and ACLOSE Return Codes

Dec	Hex	Meaning
0	X'00'	The request operation was successful
4	X'04'	No operation was performed (no error code stored)
8	X'08'	A temporary failure occurred
12	X'0C'	A permanent failure occurred
16	X'10'	A fatal error occurred (no error code stored)

AOPEN and ACLOSE errors must be handled in the mainline program (in other words, the SYNAD or LERAD exit routine is not entered when such errors occur).

- If the failure was temporary, the request may be retried after some delay.
- If the failure was permanent, the request should not be retried unless the permanent error flag set in the APCB is cleared before reissuing the request.

AOPEN and ACLOSE error codes are documented in *Cisco IOS for S/390 Unprefixed Messages and Codes*.

Application Program Organization

The API provides many facilities for program control and synchronization. Most of these facilities are optional and are not required to be used by every application program. The facilities that are used by a particular program depends in large part on the environment in which it operates and how it is organized.

Types of Organization

An application program is generally organized along one of these lines:

- Using TPL completion ECBs as the primary mechanism for synchronization and control.

  All TPL-based requests should be executed from the mainline program and little use (if any) should be made of exit routines except for error recovery. An application program organized in this manner issues a service request in anticipation of some event, and then waits for it to complete.

  This would be useful when a TRECV macro instruction is executed without knowing whether or not data was available, and at some point the program must wait for completion.

- Organizing the application program to respond to events instead of anticipating their occurrence.

  An application program organized in this manner primarily uses asynchronous exit routines and/or protocol event ECBs.

  This would happen when a DATA exit routine is defined, and when data arrives the exit routine issues the TRECV macro instruction.

An application program can also combine both approaches. However, when using ECB posting with asynchronous exits, the program must be prepared to handle certain anomalies that may occur. In particular, the time sequence of events may appear out of order.

Example

Assume a DATA exit has been specified for an endpoint operating in connection mode. Further assume that a TCONNECT macro instruction has completed successfully, and a TCONFIRM macro instruction has been issued that specifies ECB posting. When the connect confirmation arrives, the TCONFIRM request is completed and the ECB is posted. However, if data arrives at the endpoint before the application program's address space is dispatched, the DATA exit routine may be entered before the program detects that the TCONFIRM request has been completed. The application program must be able to handle this situation, or should be coded to avoid it.

Multitasking Operation Rules

The API has been designed to support both multitasking operation and multiple address space operation.

These rules apply to using the API in a multitasking environment:

- The task that issues the ACLOSE macro instruction must be the task that opened the APCB.
- All asynchronous exit routines are entered from an IRB on the TCB of the task that opened the APCB to which the exit list is linked.

- All asynchronous exit routines are entered from an IRB on the TCB of the task that opened the endpoint to which the exit list is linked.
- All asynchronous exit routines associated with a given task are executed at the same priority (in other words, asynchronous exit routines are serialized at the task level).
- If desired, other tasks may issue TPL-based requests (other than TOPEN and TCLOSE) for any given endpoint. However, exits continue to execute on the owning task, as mentioned in the previous two rules.
- Exits associated with an endpoint that is passed with a TCLOSE OPTCD=PASS macro will be driven in the task that issues the corresponding TOPEN OPTCD=OLD when the TOPEN completes.
- A task that relinquishes control of an endpoint by issuing a TCLOSE macro instruction (with OPTCD=PASS) is treated as if it had never opened the endpoint at all.
- The task that receives control of an endpoint by issuing a TOPEN macro instruction (with OPTCD=OLD) becomes the owning task, and is treated as if it originally opened the endpoint.

Multiple Address Spaces

The API supports operation across multiple address spaces. In an address space, the rules for multitasking operation, described in *Cisco IOS for S/390 Assembler API Macro Reference* apply.

An additional facility is provided to pass sessions from one address space to another. Using this facility involves establishing a session up to some state (with no active data transfer in progress) and issuing a TCLOSE macro with OPTCD=PASS. Optionally, the user may specify which address space, by ASCB address, can receive the session. The receiving program issues a TOPEN macro specifying the original endpoint ID, the OPTCD=OLD, the ASCB address of the address space that is passing the endpoint, and optionally the TCB address of the task that is passing the endpoint. The endpoint does not need to be in common storage.

Do not assume that the endpoint ID remains the same after the TOPEN OPTCD=OLD is issued; as in any TOPEN invocation, the endpoint ID token should be copied on completion of the TOPEN and used in all subsequent requests that refer to the endpoint.

The application may also specify ASCB=ANY on the TCLOSE OPTCD=PASS macro to indicate that the endpoint is eligible for passing to any address space that issues a matching TOPEN OPTCD=OLD request.

Initialization of the second address space, passing the required information, and synchronization of the two address spaces is up to the user.

24-bit and 31-Bit Addressing

The API supports operation in either 24-bit addressing mode or 31-bit addressing mode. The addressing mode in effect at the time the AOPEN macro is issued determines the addressing mode for all further operations.

Exits also run in this addressing mode. Interface routines are placed in 24-bit memory in order to accept calls in either mode. Control blocks are allocated in the addressing mode in effect when the AOPEN macro is issued.