

IATE API

Reference Manual

December 2001

Copyright © 1998-2001

InnoSys
INCORPORATED

3095 Richmond Pkwy, Ste 207
Richmond, CA 94806

+1 510 222-7717

This manual and the software described in it are copyrighted, with all rights reserved. Under the copyright laws, this manual or the software may not be copied, in whole or in part, without the written consent of InnoSys Incorporated.

NO WARRANTIES OF ANY KIND ARE EXTENDED BY THIS DOCUMENT. The information herein and the IATE™ products themselves are furnished only pursuant to and subject to the terms and conditions of a duly executed Product License.

InnoSys SPECIFICALLY DISCLAIMS ALL WARRANTIES, WHETHER IMPLIED OR EXPRESSED, INCLUDING BUT NOT LIMITED TO THOSE OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. InnoSys has no responsibility, financial or otherwise, for any result of the use of this document and/or the associated product, including direct, indirect, special and/or consequential damages. The information contained herein is subject to change without notice.

IATE is a trademark of InnoSys, Inc. Microsoft®, Windows®, Windows NT®, Windows® 2000, Windows® ME, Windows® 98, Windows® 95, and Visual Basic are either registered trademarks or trademarks of Microsoft Corp. Sun™, Sun Microsystems™, Solaris™, Netra™, Sun Enterprise™, and Ultra™ are either registered trademarks or trademarks of Sun Microsystems, Inc. SPARC® and UltraSPARC® are registered trademarks of SPARC International, Inc., licensed exclusively to Sun Microsystems, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. UNIX is a registered trademark of The Open Group. Apple and Macintosh are registered trademarks of Apple Computer, Inc. All other product names are either registered trademarks or trademarks of their respective holders.

INSCC-QP cards have been tested and found to comply with the limits for CE conformity; for Class B digital devices, pursuant to Part 15 of the FCC Rules; for the Japanese VCCI standards; and for similar standards. The FCC Class B approval is deemed to be satisfactory evidence of compliance with Canada's ICES-003 of the Canadian Interference-Causing Equipment Regulations. All of these standards and limits are designed to provide reasonable protection against harmful interference in a residential installation. This equipment generates, uses and can radiate radio frequency energy and, if not installed and used in accordance with the instructions, may cause harmful interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation. If an INSCC-QP card does cause harmful interference to radio television, or other reception, which can be determined by turning the computer off and on, the user is encouraged to try to correct the interference by one or more of the following measures:

- Reorient the receiving antenna.
- Increase the separation between the computer and receiver (radio or TV).
- Connect the computer into an outlet on a circuit different from that to which the receiver is connected.

Shielded cables must be used with the INSCC-QP cards to insure compliance with emission limits. Changes or modifications to the INSCC-QP not expressly approved by InnoSys could void the customer's right to operate the equipment.

CE Declaration of Conformity

According to EN 45014

Manufacturer's Name and Address

InnoSys Incorporated
3095 Richmond Parkway, Suite 207
Richmond, CA 94806

Declares that the product:

Product Name: INSCC

Model Number: INSCC-QP

Conforms to the following Product Specifications:

EMC: EN 55022: 1994 Class B

EN 50082-1:1992

IEC 801-2:1984 - 4kV CD, 8 kV AD

IEC 801-3:1984 - 3 V/m

IEC 801-4:1988 - 1 kV Power Lines, .5 kV Signal Lines

following the provisions of the Electromagnetic Compatibility Directive.

It also meets the EN60950:1992 standard, including amendments 1, 2 and 3, relating to the Low Voltage Directive (ITE).

Richmond, CA, USA Mike Ridenhour

June, 1996 President

Voluntary Control Council for Interference by Information Technology Equipment

This equipment is in the 2nd Class category (information equipment to be used in a residential area or an adjacent area thereto) and conforms to the standards set by the Voluntary Control Council for Interference by Data Processing Equipment and Electronic Office Machines aimed at preventing radio interference in such residential area.

When used near a radio or TV receiver, it may become the cause of radio interference.

Read the instructions for correct handling.

InnoSys Incorporated

3095 Richmond Parkway, Suite 207

Richmond, CA 94806

(510) 222-7717 Voice

(510) 222-0323 FAX

info@innosys.com

Contents

OVERVIEW OF THE API.....	1
SUPPORTED PLATFORMS	1
IATE INSTALLATION REQUIREMENTS	2
APPLICATION REQUIREMENTS	2
SUMMARY OF IATE API FUNCTIONS	3
API LIBRARY REFERENCE	4
IATESTART	4
IATEOPEN.....	8
IATECLOSE	20
IATESTOP.....	22
IATEREAD.....	24
IATEWRITE.....	28
IATECONTROL COMMANDS.....	32
APISETAPIDEBUG	33
APISETAPILOGGING.....	34
APISETDEBUGOUT.....	35
APISETOPENDELAY	37
APISETTO.....	38
APISETMSG	40
APISETSEGMENT	42
APISETAUTOANS	43
APISETNOANS.....	45
APIGETTAPROT.....	46
APIGETTACCC.....	47
APIGETHOSTSTAT.....	48
APIGETTASTAT	50
APIGETTATHROTTLE	52
APIGETOBJECTCONFIG	53
APISENDACK.....	55
APIPRINTERSTAT.....	63
APINOTATIMEOUT.....	65
APIGETVERSION.....	67
APISETHEARTBEAT	68
APISTART1MIN	70
APIRESETLOCK	71
APIRESETLOCAL	75
APIFORWARDRESET	76
APIWHOAMI.....	77
PEER-TO-PEER MESSAGES	79
APIQUERYAPPLMSG	81
APIGETAPPLMSG	82
APISENDAPPLMSG	84
APIFORCESEPERATESOCKETS.....	87

APPENDIX A: ERROR CODES	88
ERROR -2002: SERVERUNREACHABLE / NoSERVERERROR	88
ERROR -2003: OUTOFBUFFERERROR	89
ERROR -2004: OBJECTUNDEFINED / NAMEISBAD	90
ERROR -2005: NAMEINUSE.....	91
ERROR -2007: DATAERROR	92
ERROR -2008: NOTSTARTEDERROR	93
ERROR -2009: BADVERSIONERROR.....	94
ERROR -2010: DIRECTIONVIOLATION.....	95
ERROR -2011: INTERCEPTERROR	97
ERROR -2101: APINoFREECHANNEL / TOOManYSESSIONS	98
ERROR -2102: APIBADCHANNEL / INVALIDREFNUM.....	99
ERROR -2103: APIOVERRUNERR.....	100
ERROR -2201: INTERNALLOGICERROR.....	102
ERROR -2205: HOSTUNREACHABLE	103
ERROR -2207: SESSIONNOTCONFIGURED.....	104
ERROR -2208: NOSOCKET.....	105
ERROR -2209: CANTCONNECTTOSERVER	106
ERROR -2210: UNEXPECTEDMSGTYPE.....	107
ERROR -2211: WRITEFAILED	108
ERROR -2212: READFAILED.....	109
ERROR -2214: OPENBLOCKED	110
ERROR -2215: SESSIONDISCONNECTED	111
ERROR -2216: NOTIMPLEMENTED	112
ERROR -2217: TOOMUCHDATAQUEUED	113
ERROR -2218: TOOMANYCONNECTIONS.....	114
ERROR -2404: INVALIDTASK	115
APPENDIX B: BACKGROUND INFORMATION ON THE GATEWAY	116
TERMINAL AND PRINTER DEVICE OBJECTS	116
DYNAMIC OBJECTS	117
APPENDIX C: DESCRIPTION OF HOST TRAFFIC	118
APPENDIX D: SHARING A TA	120
“INTERCEPT” MODE	120
“DIVERT” MODE	120
USAGE	121
MESSAGE FORWARDING	123
SAMPLE PROGRAM	124
APPENDIX E: THE IATE API FOR VISUAL BASIC.....	125
THE IATE API DLLS FOR VISUAL BASIC.....	125
SAMPLE PROGRAMS FOR VISUAL BASIC.....	125
USING THE IATE API IN VISUAL BASIC	129
IATE API FUNCTIONS IN VISUAL BASIC	130
“HELPER FUNCTIONS” IN THE SAMPLE APPLICATIONS FOR VISUAL BASIC	132
USER-INTERFACE FUNCTIONS IN THE SAMPLE APPLICATIONS FOR VISUAL BASIC	137
THE “TIMER OBJECT” IN THE SAMPLE APPLICATIONS FOR VISUAL BASIC	139

Overview of the API

The **IATE Application Programming Interface (API)** enables an applications program to communicate with an Airline Computer Reservation System (CRS), through an InnoSys IATE Gateway.

The API provides a set of program function in the C language, supplied as a dynamic link library (DLL) in Windows, or a static library on UNIX platforms.

The API communicates with the IATE Gateway via TCP/IP protocols. The Gateway communicates with the airline host via TCP/IP, ALC, or X.25. See the Gateway documentation for information about Gateway configuration and communications.

Supported Platforms

The API is supported on the following platforms:

- Windows 2000 (initial release or Service Pack 1).
- Windows NT4 (Service Pack 3 or later).
- Windows 98 (First or Second edition).
- Windows ME (Millennium).
- Windows 95.

Sun Solaris 2.4 and later (a.k.a. SunOS version 5.5.4 and later), including Solaris 7 and 8 (a.k.a. Solaris 2.7 and 2.8.)

Contact InnoSys if you require information about API availability for other UNIX or Linux platforms.

IATE Installation Requirements

The current API communicates with Version 2.x.x IATE gateways on Windows 2000, Windows NT4, or Solaris.

The API can also communicate with Version 3.3 or later of the IATE Gateway for Apple Macintosh, with IATEtcp version 2.2b8 or higher.

Application Requirements

The IATE supports single-threaded applications on both Windows and UNIX. The current API supports multithreaded applications on Windows only. At this time, the Windows version of the API is the only version that is guaranteed to be thread safe.

Since The UNIX version of the API is not guaranteed to be thread safe, it does not support multithreaded applications on UNIX. However, the UNIX API can be used with multiple processes.

For related information on multithreaded or multiprocess applications, please refer to the description of the APIForceSeparateSockets command in the **IateControl** section of this document.

Summary of IATE API Functions

These functions are described more fully in subsequent sections of this document.

Function: **IateStart**

Purpose: **IateStart** initializes the API. The application calls **IateStart** once, before calling any of the other API functions below. The application must also call **IateStop** once before terminating.

Function: **IateOpen**

Purpose: **IateOpen** opens a communication session through which the application communicates with a TA object on the airline host, via the IATE Gateway. After opening a session with **IateOpen**, the application can **IateRead** or **IateWrite** to communicate, or **IateControl** to control various options on the session. For each session opened with **IateOpen**, the application should later call **IateClose** to close the session.

Function: **IateRead**

Function: **IateWrite**

Purpose: The application uses **IateWrite** and **IateRead** to send and receive messages from the airline host, via the IATE Gateway. These functions communicate on a TA object which the application established through **IateOpen**.

Function: **IateControl**

Purpose: **IateControl** handles several special commands, with which the application can query and control various options related to the operation of the API, the Gateway, or a particular TA object. See the **IateControl** section of this document for information about the various commands available.

Function: **IateClose**

Purpose: The application uses this to release a TA object, when the application is no longer using it. (This function is the counterpart to **IateOpen**.)

Function: **IateStop**

Purpose: The application uses this to release its connection to the API, when the application is about to terminate, or when the application has no further use for the API. (This function is the counterpart to **IateStart**.)

API Library Reference

This section describes the API functions in detail.

lateStart

Summary:

```
long  
IateStart(  
    long install_handlers,  
    long dummy,  
    unsigned char *buff);
```

Purpose:

IateStart initializes the API, preparing it for subsequent API function calls.

Usage:

The application calls **IateStart** before **IateOpen**. If **IateStart** succeeds, it returns a *start code* value, which the application should pass in any subsequent calls to **IateOpen**.

Under normal conditions with a valid IATE software installation, **IateStart** is not expected to fail. If **IateStart** fails and returns an error code, the application cannot use the API. The application should not call **IateOpen** or any other API functions after **IateStart** returns an error code. Any such calls would also return errors, because the API has not been initialized.

Arguments:

`install_handlers`

This argument tells the API whether or not to use its own signal handlers:

- 1 tells the API to use its own signal handlers. This is the recommended value.
- 0 means the application intends to supply its own signal handlers. This is not recommended.

`dummy`

This argument is ignored.
(It is included for backwards compatibility with older applications that used previous versions of the IATE API.)

`buff`

This string argument optionally specifies the Gateway host name or IP address, and the network service or port on which the Gateway listens for API client connections. The API assigns these as default values for subsequent calls to **IateOpen**.

IateStart does not connect to the Gateway. The values specified here apply only to subsequent **IateOpen** calls that do not explicitly specify the Gateway host address and network service or port.

If the application does not specify the host name or IP address in either the **IateStart** call or an **IateOpen** call, the default is the local host on which the application is running.

If the application does not specify the network service name or port number in either the **IateStart** call or an **IateOpen** call, the default is “**ialcserver**”, which is normally associated with the IATE default network port number, 1413.

If a service name is specified, it must be defined in the system's network “**services**” file. Refer to **Appendix I** for information about the “**services**” file.

This argument takes one of these four formats:

" "

a blank string (not a NULL pointer)

"@Host\\Service\\"

This specifies the Gateway host address, and the network service name or port number for connection to the Gateway.

Substitute the Gateway host's name, or IP address, in place of "Host" above.

In place of "Service" above, substitute the network service name, or network port number, for connection to the Gateway.

"@Host\\"

This specifies the Gateway host name or IP address.

(The unspecified service name defaults to "**ialcserver**", which is normally associated with the IATE default network port number, 1413.)

"ServiceName\\"

This specifies the network service name or port number for connection to the Gateway.

(The unspecified Gateway host name defaults to the local host on which the application is running.)

Note the double backslashes between the fields in the arguments above. This is a C-language convention: Each successive pair of backslashes \\ equates to a single backslash in the final string. Programs using the IATE API for Visual BASIC would use single backslashes rather than pairs.

Returns:

< 0 Error.

> 0 Success.

The return value is a session reference number for use in subsequent **IateRead**, **IateWrite**, **IateControl**, or **IateClose** calls.

Example:

```
long start_code;  
start_code = IateStart (1, 0, "@gw2\\ialcserver\\");
```

lateOpen

Summary:

```
#include "U_API.h"

long
IateOpen(
    long start_code,
    long cmd,
    unsigned char *buff);
```

Purpose:

IateOpen establishes a link to a TA object at the IATE Gateway, for communication with the airline reservation host.

Usage:

The application calls **IateStart** before calling **IateOpen**. (See the **IateStart** information above.)

The value returned from a successful **IateOpen** call is known as a reference number (or *refnum*). The application passes the refnum to all subsequent **IateRead**, **IateWrite**, **IateControl**, or **IateClose** calls for this object.

Objects are defined in the gateway configuration file, and map each object name to an IA TA line address. Refer to the gateway installation manual for more details on object names.

Arguments:

start_code

This argument should be set to the start-code value that **IateStart** returned.

cmd

This argument should be set to one of the following four commands:

APILinkToName - to link to an object by its name or group-name.

APILinkToTa - to link to an object by specifying an IA & TA.

APILinkToDyCrt - to link to a "dynamic CRT" object.

APILinkToDyPrt - to link to a "dynamic printer" object.

buff

This argument contains a string, which specifies the TA object (or group of objects) on which the application requests a connection.

The format and contents of this string argument depend on the type of connection requested, as indicated by the value of **cmd**. Explanations and examples are given below.

The contents of **buff** specify the TA object to connect; and may also specify the Gateway host name or IP address, and the network service or port on which the Gateway listens for API client connections.

If the application does not specify the host name or IP address in the **IateOpen** call, it defaults to the value specified in the earlier call to **IateStart**, or to the local host on which the application is running.

If the application does not specify the network service name or port number in the **IateOpen** call, it defaults to the value specified in the earlier call to **IateStart**, or to "ialcserver" (which is normally associated with the IATE default network port number, 1413).

If a service name is specified, it must be defined in the system's network "services" file. Refer to **Appendix I** for information about the "services" file.

Note:

The contents of **buff** are not preserved by the **IateOpen** call.

For the **APILinkToName** command:

If **cmd** is **APILinkToName**, **buff** contains a string which can specify the following values:

a Gateway host name,
a TCP/IP Service name, and
a TA object name or group name.

Syntax:

The **buff** string argument takes one of these formats:

"@HostName\\ServiceName\\ObjectOrGroupName "

Substitute the Gateway host's name, or IP address, in place of "Host" above.

In place of "Service" above, substitute the network service name, or network port number, for connection to the Gateway.

Specify a TA Object or Group name in place of "ObjectOrGroupName".

"@\\ServiceName\\ObjectOrGroupName "

This syntax omits the host address. It defaults to the host name or IP address that the application previously specified in the **IateStart** call, or to the local host on which the application is running.

In place of "Service" above, substitute the network service name, or network port number, for connection to the Gateway.

Specify a TA Object or Group name in place of "ObjectOrGroupName".

"@HostName\\\\\\\\ObjectOrGroupName "

This syntax omits the network service/port. It defaults to the service name or port number that the application previously specified in the **IateStart** call, or to "ialcserver" (which is normally associated with the IATE default network port number, 1413).

"@\\\\\\\\\\\\\\\\ObjectOrGroupName "

This syntax omits the HostName and ServiceName. The API uses the names that the application previously specified in the **IateStart** call.

(Note the double backslashes between the fields in the arguments above. This is a C-language convention: Each successive pair of backslashes `\\` equates to a single backslash in the final string. Programs using the IATE API for Visual BASIC would use single backslashes rather than pairs.)

The TA **Object or Group Name** specifies which TA object the application wishes to use for communications with the airline host. This must match one of the object or group names listed in the Gateway's configuration.

If the application specifies a TA Object name, the **IateOpen** call requests a connection to that TA object. If the application specifies a Group name, this requests a connection to any object in the specified group. Refer to the IATE Gateway documentation for information about how to configure TA objects and groups.

For the **APILinkToTa** command:

The **APILinkToTa** command is generally not recommended. **APILinkToName** is usually appropriate.

If **cmd** is **APILinkToTa**, **buff** contains a string that specifies an IA and TA number (instead of an object or group name). The Host and Service names can also be specified in the same way as for the **APILinkToName** command (see above).

Syntax / Example:

```
"@\\HostName\\ServiceName\\040\020"
```

The IA and TA values are characters specified in ALC code using octal numeric values. In the example above, the IA value is 040 octal (equal to 20 hexadecimal or 32 decimal), and the TA value is 020 octal (equal to 10 hexadecimal or 16 decimal).

All C compilers allow octal character values, such as `'\040'` and `'\040'` as shown in the example above. Some compilers also support hexadecimal character values using a different syntax, but this may not work in some cases, so we recommend the octal format.

Optionally, the **buff** argument can also specify a port name, immediately following the TA value in the string. The port name resolves to a particular physical line. It's necessary to specify the port name only if the Gateway has the same IA and TA configured on multiple ports. The specified port name must match one that is specified in a Gateway configuration file's **PORT_NAME** directive.

Syntax / Example:

```
"@\HostName\ServiceName\040\020PortName"
```

For the **APILinkToDyCrt** or **APILinkToDyPrt** command:

The **APILinkToDyCrt** and **APILinkToDyPrt** commands request a connection to a “dynamic” terminal or printer object, respectively.

A “dynamic” object is one that the Gateway configuration specifies with the **TERMINAL_API** or **PRINTER_API** object-type.

Dynamic objects are similar to named “groups” of objects, in the sense that the application requests a connection to “any object” of the specified dynamic type.

If **cmd** is **APILinkToDyCrt** or **APILinkToDyPrt**, the application should specify the **buff** string using the same syntax as for **APILinkToName**:

```
"@HostName\ServiceName\ObjectName"
```

(The Host and Service names are optional under the same conditions as for **APILinkToName**. See the examples under **APILinkToName**, above.)

For Dynamic objects, the specified Object Name is just a placeholder. The API will not use it, but it must be specified anyway.

Returns:

If **IateOpen** succeeds in establishing a link to a TA object, it returns a nonnegative value.

If the link was not successful, **IateOpen** returns a negative error value. Refer to **Appendix A** for information about IATE API error values and their causes.

IateOpen does not preserve the string argument that the caller placed in **buff**. (If the caller will later re-use the string value that it placed in **buff**, the caller may need to save the string value in a separate buffer before calling **IateOpen**.)

On return, **buff** will contain a 2-byte value. Historically, this indicated the IA and TA associated with the connected object. Although this indication is still valid in some cases, it is generally not recommended to use these returned IA and TA values in new application code. If the application requires information about the configured IAs and TAs, the preferred way to obtain such information is to use **IateControl** with the **APIGetObjectConfig** command.

Note:

The API normally enforces certain restrictions on **IateOpen** calls, due to the nature of TCP socket connections.

It takes time to close down a TCP/IP socket gracefully. Rapidly opening and closing sockets may be inefficient on the system and network, and may cause problems on some systems.

The API enforces a minimum time between consecutive **IateOpen** calls. The interval required between successive **IateOpen** calls can be adjusted using **IateControl** with the **APISetOpenDelay** command.

We strongly recommend that the open delay be set to no less than 10 seconds.

See also:

IateControl function, **APISetOpenDelay** command.

Appendix B: Background Information about the Gateway.

Example:

This example assumes two gateways, one running on a remote machine and one running on the local machine along with the application program.

The application runs on system “gw1”. The gateway on that system has the following objects defined:

IA	TA	type	object	group
01	01	TERMINAL	term11	**
01	02	TERMINAL_API	term12	**
01	03	TERMINAL_API	term13	**
01	04	TERMINAL	term14	group1
01	05	TERMINAL	term15	group1

The second gateway is running on a remote machine named “gw2” with the following objects defined:

02	01	TERMINAL	**	group2
02	02	TERMINAL_API	term22	group2
02	03	TERMINAL_API	term23	**
02	20	TERMINAL	term220	**
02	05	TERMINAL	term25	*

This program shows the ability of the API to connect to multiple gateways and different types of objects. We'll assume that no other IATE applications are running, so the configured objects are all available.

```

/* ----- links.c ----- */

#include <stdio.h>
#include "U_API.h"
#include "U_APItyp.h"
#include "U_APIpros.pro"

/*
   Initialize a list of TA object connection specifiers for IateOpen.
*/

#define NUM_OBJS 4
char *connection_specifiers[NUM_OBJS] =
{
    "gw2\\ialcserver1\\term23", /* an object on a remote gateway */
    "term14", /* an object on a local gateway */
    "gw1\\ialcserver\\group1", /* a request for an object in the group "group1" */
    "gw1\\ialcserver\\dummy" /* a dynamic link */
};

unsigned char
    buff[MAX_BUFF_SZ], /* message buffer */
    ctrl[CTRL_BLK_SZ]; /* control block:C1,C2,EOMx,CCC_OK,MORE */
/* (See the IateRead documentation) */

int userBreak(void); /* program termination function
                     (defined below) */

main()
{
    struct u_link_response
        config; /* object configuration information */

    long
        start_code, /* start-code returned from IateStart */
        result_IateRead, /* return value from IateRead */
        result_IateWrite, /* return value from IateWrite */

        refnums[NUM_OBJS]; /* reference number for each connection */

    short
        open_delay; /* parameter for APISetOpenDelay command */

    int
        conn_index = 0, /* index to list of connections */
        num_conns = 0; /* number of connections */

    /*
       Initialize API -
       IateStart returns "start code"
       which must be passed to IateOpen.
    */

    start_code = IateStart(1, 0, (unsigned char *)"");
    if (start_code < 0)
        exit(1);
    else
        printf ("IateStart OK\n");
}

```

```

/*
    Set the minimum time required between IateOpen calls.
    Do not set this time to less than 10 seconds.
*/

open_delay = 10;                /* (seconds) */
IateControl(
    0,
    APISetOpenDelay,
    (unsigned char *) &open_delay);

/*
    Link to gateway running on remote machine "gw2".
    (Host and service names are required here.)
*/

refnums[num_conns] =
    IateOpen(
        start_code,
        APILinkToName,
        (unsigned char *) connection_specifiers[num_conns]);
        /* "@gw2\\ialcserver1\\term23" */

num_conns++;

/*
    Link to gateway running on the local machine "gw1".
    (Host and service names are optional here.)
*/

refnums[num_conns] =
    IateOpen(
        start_code,
        APILinkToName,
        (unsigned char *) connection_specifiers[num_conns]);
        /* "term14" */

num_conns++;

/*
    Link to remote gateway on
    IA: hexadecimal 02 = octal \002,
    TA: hexadecimal 20 = octal \040.

    When using IateOpen with APILinkToTa,
    the IA and TA are the ALC values converted to octal format,
    for proper embedding in the object_name string.
*/

refnums[num_conns] =
    IateOpen(
        start_code,
        APILinkToTa,
        (unsigned char *)connection_specifiers[num_conns]);
        /* "@gw2\\ialcserver1\\002\040" */

num_conns++;

```

```

/*
  Link to first free TA in group 1 on the local machine "gw1".
  This should result in a link to term15.
*/

refnums[num_conns] =
  IateOpen(
    start_code,
    APILinkToName,
    (unsigned char *) connection_specifiers[num_conns]);
    /* "@gw1\\ialcserver\\group1" */

num_conns++;

/*
  Dynamic link to a Dynamic CRT object on the local machine:
  This results in a link to "term22", since this is the first
  dynamic CRT. The "dummy" object-name must be specified here
  as a placeholder, although the API does not use it.
*/

refnums[num_conns] =
  IateOpen(
    start_code,
    APILinkToDyCrt,
    (unsigned char *) connection_specifiers[num_conns]);
    /* "@gw1\\ialcserver\\dummy" */

num_conns++;

/*
  For each TA object that was successfully linked,
  send a message to the host.
*/

for (conn_index = 0; conn_index < num_conns; conn_index++)
{
  if (refnums[conn_index] < 0)
    printf (
      "IateOpen failed (error %d) for: %s\n",
      refnums[conn_index],
      connection_specifiers[conn_index]);
  else
  {
    /* Get the object's configuration information. */

    IateControl(
      refnums[conn_index],
      APIGetObjectConfig,
      (unsigned char *) &config);

    /* Display the object's IA and TA. */

    printf(
      "Successful link to object IA TA: %s\n",
      config.iata_str);

    /* Send a message to the host. */

    strcpy (buff, "I");
    result_IateWrite =
      IateWrite(
        refnums[conn_index],
        strlen(buff),
        buff);
  }
}

```

```

    if (result_IateWrite < 0)
        printf(
            "IateWrite failed (error %d)\n",
            result_IateWrite);
    }
}

/*
Cycle through the sessions and read the host responses.
See the IateRead documentation for details.

Note: This is not a complete example.
Additional code, not shown in this example, should provide a way to
exit this loop. For example, in a console program, a signal handler
could catch a Ctrl-C keystroke, call IateClose for the open sessions,
and call IateStop before terminating this process.
*/

while (!userBreak())          /* loop until user requests termination */
{
    for (conn_index = 0; conn_index < num_conns; conn_index++)
    {
        if (refnums[conn_index] > 0)
        {
            result_IateRead =
                IateRead(
                    refnums[conn_index],
                    MAX_BUFF_SZ,
                    buff,ctrl);

            if (result_IateRead == 0)
                continue;          /* no message received */
            else
            {
                if (result_IateRead < 0)
                {
                    printf(
                        "IateRead failed (error %d)\n",
                        result_IateRead);

                    refnums[conn_index] = -1;
                }
                else
                {
                    /*
                     * Null-terminate and display the received string
                     */

                    buff[result_IateRead - 2] = '\0';

                    printf(
                        "Reply from host (refnum %d): %s\n",
                        refnums[conn_index],
                        buff);
                }
            }
        }
    }
}
}

```



```

int
userBreak(void)
{
    /*
    This function should return a nonzero value if the
    user has requested termination of the program.
    After this function returns nonzero,
    the caller should terminate the program gracefully.

    The body of this function is not shown here.
    The means of detecting user input depends on the
    platform (Windows or UNIX), the type of application,
    and choice of implementation.

    For example, in a console program, this function could
    work with a signal handler to detect a Ctrl-C keystroke.
    After the user presses Ctrl-C to terminate the program,
    this function would return nonzero, and the caller
    would proceed to terminate the program.
    */

    /*
    ... Insert code here, to return nonzero
    if the user has requested program termination ...
    */

    return 0;
}
/* ----- */

```

lateClose

Summary:

```
long  
IateClose(long refnum);
```

Purpose:

IateClose terminates a TA object link that was opened with **IateOpen**.

Usage:

IATE applications should use **IateClose** to close any open connections -- before the application terminates, or whenever the application has finished using an open session.

A TA object accepts normal connections from only one application at a time. (There is an exception to this. A second application can connect to an object through the "Shared TA" mechanism, but that is a special type of connection.) After an application uses **IateOpen** to connect, other applications cannot establish normal connections to the object, until the first application calls **IateClose** to disconnect.

If the application fails to call **IateClose**, and leaves any TA object connections open when it terminates, the open connections may not be properly closed. (This may happen if the application crashes, or if the application was not properly written.) In that case, the connection may remain 'stuck' open indefinitely, unless the Gateway is configured to disconnect it after an inactivity timeout.

Arguments:

refnum

The reference number that **IateOpen** returned.

Returns:

Zero on success, or a negative error code. Refer to **Appendix A** for information about IATE API error values and their causes.

Example:

```
long refnum;  
...  
refnum = IateOpen(...);  
...  
IateClose(refnum);
```

IateStop

Summary:

For Windows applications:

```
long  
IateStop(long startcode);
```

For Solaris or other UNIX:

```
long  
IateStop(void);
```

Purpose:

IateStop terminates the application's use of the IATE API.

Usage:

The application should call **IateStop** before terminating.
(**IateStop** is the counterpart to **IateStart**,
which the application called while starting up.)

Argument:

For the Windows version only:

startcode

This is the start code value that **IateStart** returned.
Applications running on Windows pass this value to **IateStop**.
Applications running on UNIX do not pass this value to **IateStop**.

Returns:

Zero on success, or a negative error code.

Refer to **Appendix A** for information about IATE API error values and their causes.

Example:

```
long startcode;
...
startcode = IateStart(...);
...

/* For Windows only: */
IateStop(startcode);

/* For UNIX only: */
IateStop();
```

IateRead

Summary:

```
long
IateRead(
    long refnum,
    long nchars,
    unsigned char *buff,
    unsigned char *ctrlblock);
```

Purpose:

IateRead reads message data that the Gateway has received from the airline host.

Usage:

If message data is already available when the application calls **IateRead**, it will return immediately.

If no message is immediately available, **IateRead** will wait until a message or segment arrives, or until a timeout expires. The application can specify the timeout by calling **IateControl** with the **APISetTO** command. If the wait time expires with no message data received, the call to **IateRead** will complete with a return value of 0.

IateRead can read complete messages, or it can read individual message segments. This depends on whether the application has selected message or segment reading mode. (See **IateControl**, **APISetMsg** and **APISetSegment**.)

Arguments:

refnum

The reference number that **IateOpen** returned for this session.

nchars

The size of the buffer to receive data.

Suggested value: `MAX_BUFFER_SZ`.

buff

This is the buffer to receive data. If **IateRead** reads a message, this buffer will contain the text of the message.

The message text in this buffer will not include the command characters, the EOM, and the CCC indicator. They will be returned in **ctrlblock**.

The received data is in ASCII. (The Gateway handles character translation from ALC to ASCII.)

ctrlblock

An additional buffer of size `CTRL_BLK_SZ`, in which **IateRead** will store the following information:

`ctrlblock[CTRL_C1]`

The C1 character in ASCII.

`ctrlblock[CTRL_C2]`

The C2 character in ASCII.

`ctrlblock[CTRL_EOM]`

The EOM character in ASCII.

ctrlblock[CTRL_CCC_OK]

- 1 if the CCC (message checksum) was valid,
- 0 if the CCC was invalid.

(If the CCC was invalid, the application should discard the message.)

ctrlblock[CTRL_MORE]

- 1 if this message is not complete:
a subsequent call to **IateRead**
will return the next part of the message.
- 0 if this call to **IateRead** returned a
complete message.

Returns:

If **IateRead** returns any message data in **buff**, the function's return value indicates the length of the message, plus 2. The additional 2 characters, C1 & C2, are positioning characters in the message envelope, as described in **Appendix C: Description of Host Traffic**.

IateRead returns zero if the timeout expired with no data available.

Negative return values indicate errors. Refer to **Appendix A** for information about IATE API error values and their causes.

Examples:

(See examples in this document and the sample programs which come with the IATE API package.)

```
unsigned char
    buff[MAX_BUFF_SZ],          /* 2015 character message buffer */
    ctrl[CTRL_BLK_SZ];        /* control block:
                               C1, C2, EOMx, CCC_OK, MORE */
long
    refnum,                    /* successful IateOpen return value */
    nchars,                    /* nchars = len of message plus 2 */
    ret;                       /* IateRead return value */

...
IateRead (refnum, MAX_BUFF_SZ, buff, ctrl);
```

lateWrite

Purpose:

lateWrite sends a message to the airline host.

Syntax:

```
long  
lateWrite(  
    long refnum,  
    long nchars,  
    unsigned char *buff)
```

Description:

refnum

The reference number that **lateOpen** returned for this session.

nchars

The size of the buffer containing data to send.

buff

This is the buffer containing data to send.

The data is in ASCII. (The Gateway handles character translation from ASCII to ALC.)

If the last character of the message data is an EOM character, the API sends the message to the host with that EOM. If no EOM is supplied, the API sends the message with EOMC (End Of Message – Complete).

Returns:

IateWrite returns a negative value if an error occurs. Refer to **Appendix A** for information about IATE API error values and their causes.

Two errors in particular, **-2210 (DirectionViolation)** and **-2103 (API OverrunErr)** can occur as a result of application design issues. For details, refer to the descriptions of these errors in **Appendix A**.

Example:

(See examples in this document and the sample programs which come with the IATE API package.)

```
unsigned char
    buff[MAX_BUFF_SZ];
long
    refnum,          /* successful IateOpen return value */
    ret;            /* IateWrite return value */

...
strcpy (buff, "I");    /* send an Ignre message to the host: */
ret = IateWrite (refnum, strlen(buff), buff);
```

lateControl

Summary:

```
long  
IateControl(  
    long refnum,  
    long cmd,  
    unsigned char *buff);
```

Purpose:

The **IateControl** function performs a number of different functions (specified by a command-code parameter) -- such as setting various parameters for API and Gateway operation, obtaining configuration information from the Gateway, managing message acknowledgment, and indicating printer status.

Arguments:

refnum

The reference number for the TA object connection on which to perform a control command. (This can be zero for certain commands which do not require a TA object connection.)

cmd

A command code specifying the operation to perform. See below.

buff

Buffer for input or output data, used with some commands.

Returns:

On success, **IateControl** returns zero, or a command-specific return value.
(Refer to the description of each command below.)

On failure, **IateControl** returns a negative error code.
Refer to **Appendix A** for information about IATE API error values and their causes.

lateControl Commands

The command code is the second of the three argument to **lateControl**. The first **lateControl** parameter is a TA object connection reference number (if required). The third argument is an input or output parameter which depends on the particular command.

Some **lateControl** commands operate on an established TA object connection. Such commands require the connection reference number as the first argument to **lateControl**. Some other commands do not operate on a particular object connection; for those commands the first argument should be zero.

APISetApiDebug

Purpose:

This **IateControl** command enables or disables API diagnostic output to the default or custom output destination, as described below.

Arguments:

The first argument to **IateControl** should be zero for this command, because this command does not operate on a specific TA object.

The second argument to **IateControl** is the **APISetApiDebug** command.

The third argument to **IateControl** is a bit-mask value specifying the diagnostic output levels to enable. See **Appendix G** for a description of each diagnostic output level. If the third argument's value is zero, this turns off all API diagnostic output.

If the third argument's value is nonzero, the API will generate the enabled diagnostic output to a default or custom output destination. The default destination is the "standard output" which works for console (text-mode) programs. A non-console application (such as a program that uses a graphical user interface, or no on-screen user interface), or an application that requires a different destination for diagnostic output, can use the **APISetDebugOut** command to define a custom diagnostic output function.

See also:

APISetDebugOut
APISetApiLogging

Example:

```
short val = 0xff;  
IateControl (0, APISetApiDebug, (unsigned char *) &val);
```

APISetApiLogging

Purpose:

This **IateControl** command enables or disables API diagnostic output to a log file, as described below.

Arguments:

The first argument to **IateControl** should be zero for this command, because this command does not operate on a specific TA object.

The second argument to **IateControl** is the **APISetApiDebug** command.

The third argument to **IateControl** is a bit-mask value specifying the diagnostic output levels to enable. See **Appendix G** for a description of each diagnostic output level. If the third argument's value is zero, this turns off all API diagnostic output.

If the third argument's value is nonzero, the API will generate the enabled diagnostic output to a file named "**iatelog.log**".

See also:

APISetApiDebug

Example:

```
short val = 0xff;  
IateControl (0, APISetApiLogging, (unsigned char *) &val);
```


APISetDebugOut

Purpose:

This command specifies an alternate function for displaying or processing any API diagnostic output enabled by **APISetApiDebug**. See the sample program below and “**testterm.c**” (supplied with the API distribution) for examples of an alternate output function.

Arguments:

The first argument to **IateControl** is zero, and the third argument specifies the diagnostic output function.

The application supplies the output function. It is a formatted output function which works like **printf**. It takes a format string as its first argument, followed by a variable argument list containing any additional output parameters. The example below shows a typical implementation.

Example:

```
void testDebugOut (char *format_str, ...);
IateControl (0, APISetDebugOut, (unsigned char *) testDebugOut);

...

#define MAX_DBG_MSG_LENGTH 255

void
testDebugOut (char *format_str, ...)
{
    char line[MAX_DBG_MSG_LENGTH + 1];

    /*
     * Format the output string and parameters,
     * in the same manner as printf()
     */

    va_list marker;
    va_start (marker, format_str);
    _vsnprintf (line, MAX_DBG_MSG_LENGTH, format_str, marker);
    va_end (marker);
}
```

```
/*  
    Output the string  
*/  
  
printf (line);    /* (or use a custom output function) */  
}
```

APISetOpenDelay

Purpose:

This command sets the delay, in seconds, required between successive **IateOpen** calls. Do not set this delay to less than 10 seconds.

It takes time to close down a TCP/IP socket gracefully. Rapidly opening and closing sockets may be inefficient on the system and network, and may cause problems on some systems.

The API enforces a minimum time between consecutive **IateOpen** calls. The interval required between successive **IateOpen** calls can be adjusted using **IateControl** with the **APISetOpenDelay** command.

Arguments:

The first argument to **IateControl** is zero.

The second argument is the **APISetOpenDelay** command.

The third argument specifies the delay time.

The delay time should not be set to less than 10 seconds.

Example:

```
short val = 10;  
IateControl (0, APISetOpenDelay, (unsigned char *) &val);
```

APISetTO

Purpose:

This command sets or disables a timeout for **IateRead**.

When the application calls **IateRead**, the call will return after message data becomes available from the host, or the read timeout expires (whichever occurs first).

The read timeout defaults to 1 second. The application can use **APISetTO** to control the timeout, as described below.

If the blocking interval expires with no message received, the call to **IateRead** returns zero.

Arguments:

The first argument to **IateControl** specifies the TA object connection to which this command applies.

If the first argument is zero, this command does not immediately affect any currently open session. However, this command sets default timeout values for any future sessions created through subsequent calls to **IateOpen**.

The second argument to **IateControl** is the **APISetTO** command.

The third argument to **IateControl** specifies the timeout period, using a **timeval** structure. In that structure, the **tv_sec** field can specify the number of seconds, and the **tv_usec** field can specify an additional number of microseconds. Three different modes can be specified:

- If **tv_sec** and/or **tv_usec** are not zero, subsequent calls to **IateRead** will return after the specified timeout, or after data arrives (whichever comes first).
- If both the **tv_sec** and **tv_usec** fields are zero, subsequent calls to **IateRead** will return immediately, regardless of whether or not any data has been received.
- If the third argument is NULL, the **APISetTO** command disables the read timeout. In this mode, calls to **IateRead** will not return until message data becomes available from the host.

Caution:

A blocking interval of one second or less may cause some inaccuracy in the amount of time that **IateRead** waits for data. **IateRead** may return with no data before the blocking interval has expired.

See Also:

APIResetLock

Example:

```
#include <time.h>

struct timeval to;

/*
   The following example sets a blocking interval of 1 second.
   (Note that IateRead might return with no data
   before 1 second has elapsed.)
*/

to.tv_sec = 1;
to.tv_usec = 0;
IateControl (refnum, APISetTO, (unsigned char *) &to);

/*
   The following example sets a blocking interval of 100 microseconds.
   IateRead will complete when a message is available
   or after 100 microseconds have expired, whichever comes first.)
*/

to.tv_sec = 0;
to.tv_usec = 100;
IateControl (refnum, APISetTO, (unsigned char *) &to);
```

APISetMsg

Purpose:

This command puts a TA object connection into “Message Mode”. In Message Mode, **IateRead** returns after receiving a complete data message of one or more segments, up to and including a segment that ends with the EOMC indicator (End of Message, Complete), or EOMU (End of Message, Unsolicited).

If the application does not issue this command, the TA object connection will remain in “Segment Mode” by default. In Segment Mode, **IateRead** returns as soon as a received data segment becomes available, without waiting for completion of a multi-segment message.

It is generally advisable to use Segment Mode rather than Message Mode, for reasons such as those explained below.

Caution:

It is not appropriate to use Message Mode (and Auto-Answer mode) if the application must acknowledge the individual segments of an incoming message.

For example: If the application serves a printer TA object to generate tickets or other critical output, the application may be required to acknowledge each segment after it is printed, as a form of end-to-end assurance. (This requirement applies to SABRE “protected mode” printer connections.)

In such cases, the Message Mode should not be enabled, and the Gateway's Auto-Answer mode should not be enabled. (See **APISetAutoAns.**)

Another possible reason to avoid Message Mode is that the accumulation of an entire message may sometimes exceed the API's buffer capacity.

Arguments:

The first argument to **IateControl** specifies the TA object connection to which this command applies. The second argument is the **APISetMsg** command. The third argument is ignored and should be zero.

See Also:

APISetSegment

Example:

```
IateControl (refnum, APISetMsg, (unsigned char *) 0);
```

APISetSegment

Purpose:

This command puts a TA object connection into “Segment Mode”. In Segment Mode, **IateRead** returns as soon as a received data segment becomes available, without waiting for completion of a multi-segment message.

This is the default mode; **APISetMsg** selects the alternative mode. It is generally advisable to leave the connection in Segment Mode. (For more information, see **APISetMsg** and **APISetAutoAns**.)

Arguments:

The first argument to **IateControl** specifies the TA object connection to which this command applies. The second argument is the **APISetSegment** command. The third argument is ignored and should be zero.

See Also:

APISetMsg

Example:

```
IateControl (refnum, APISetSegment, (unsigned char *) 0);
```


APISetAutoAns

Purpose:

This command turns on the Gateway's "Auto-Answer" mode, for automated acknowledgments. This instructs the Gateway to send segment acknowledgments to the host automatically, immediately after the gateway receives each segment.

The Auto-Answer mode is required when operating in Message Mode (after an **APISetMsg** call). However, Message Mode and Auto-Answer are generally not advised, for reasons such as those explained below.

Caution:

It is not appropriate to use Message Mode and Auto-Answer mode if the application must acknowledge the individual segments of an incoming message.

For example: If the application serves a printer TA object to generate tickets or other critical output, the application may be required to acknowledge each segment after it is printed, as a form of end-to-end assurance. (This requirement applies to SABRE "protected mode" printer connections.)

In such cases, the Message Mode should not be enabled, and the Gateway's Auto-Answer mode should not be enabled.

The **APISetAutoAns** command affects only the specified TA object connection on which the application issues the command.

The Gateway also has an AUTO_ANSWER configuration item. Specified in a Gateway configuration file, AUTO_ANSWER determines whether the auto-answer mode is turned on or off by default, for all of the connections defined in that configuration file. "AUTO_ANSWER 1" enables Auto-Answer. "AUTO_ANSWER 0" disables it, and that is the default setting. (Refer to **IATE Gateway documentation** for more information.)

If the application does not issue **APISetAutoAns**, and if the AUTO_ANSWER option is not specified in Gateway configuration (or if it is set to 0), the Auto-Answer mode will be disabled.

Arguments:

The first argument to **IateControl** specifies the TA object connection to which this command applies. The second argument is the **APISetAutoAns** command. The third argument points to a character which should contain a nonzero value to turn the option on.

See Also:

APISetNoAns

Example:

```
unsigned char x = 1;  
IateControl (refnum, APISetAutoAns, &x);
```

APISetNoAns

Purpose:

This command turns off the Gateway's "Auto-Answer" mode. This prevents the Gateway from sending segment acknowledgments to the host automatically.

If the Auto-Answer mode is turned off, the application must use **IateControl** with the **APISendAck** command to acknowledge each received segment. The Gateway sends a segment acknowledgment to the host after the application uses **APISendAck**.

This command is appropriate for any TA object requiring end-to-end segment-delivery assurance, such as a ticket printer TA.

This command affects only the specified TA object connection. The Gateway also has an AUTO_ANSWER configuration item. That option, specified in a Gateway configuration file, determines whether the auto-answer mode is turned on or off by default, for all the connections defined in that configuration file. (See the IATE Gateway documentation for information on Gateway configuration.)

Arguments:

The first argument to **IateControl** specifies the TA object connection to which this command applies. The second argument is the **APISetNoAns** command. The third argument is ignored and should be zero.

See Also:

APISetAutoAns

Example:

```
IateControl (refnum, APISetNoAns, (unsigned char *) 0);
```

APIGetTaProt

Purpose:

This command finds out whether or not the specified object is associated with a SABRE protected printer TA. This function applies to SABRE connections only.

Arguments:

The first argument to **IateControl** specifies the TA object connection to which this command applies. The second argument is the **APIGetTaProt** command. The third argument is ignored and should be zero.

Example:

```
ret = IateControl (refnum, APIGetTaProt, (unsigned char *) 0);
if (ret >= 0)
    printf(
        "TA %s in protected mode",
        ret ? "is" : "is not");
```

APIGetTaCCC

Purpose:

This command finds out whether or not the segment checksum (CCC) validation succeeded on the most recent message or segment received from the host.

The CCC validation status also available in the control-buffer returned with each message or segment from **IateRead**. If the application checks that status from **IateRead**, it may not be necessary to use **APIGetTaCCC**. Refer to the discussion of **IateRead** for details.

Arguments:

The first argument to **IateControl** specifies the TA object connection to which this command applies. The second argument is the **APIGetTaCCC** command. The third argument is ignored and should be zero.

Returns:

IateControl returns a value of zero or greater to indicate that the checksum validation succeeded, or a negative value to indicate that the validation failed.

If the validation succeeded, the segment contains valid data.

If the validation failed, the segment is corrupt, and the application should not use it.

Example:

```
ret = IateControl (refnum, APIGetTaCCC, (unsigned char *) 0);
if (ret >= 0)
    printf ("CCC was %s on last message ", ret ? "GOOD" : "BAD");
```

APIGetHostStat

Purpose:

This command retrieves host connection status information. The meaning of the returned status information depends on the type of host connection, as noted below.

Arguments:

The first argument to **IateControl** specifies the TA object connection to which this command applies. The second argument is the **APIGetHostStat** command. The third argument specifies a buffer in which this call returns a host connection status value.

Returns:

For ALC connections,

the returned status information is as follows:

The status value indicates the current state of the IA polling state and the host line's modem control signals. (The required modem control signals include DCD, DCR, and/or CTS, depending on Gateway configuration.)

If the host connection's required modem signals are up and the IA is being polled, the **IateControl** return value is 1, and the status value returned in the buffer is 701 hexadecimal.

If the host connection's required modem signals are not all up, or the IA is not currently being polled, the **IateControl** return value is 0, and the status value returned in the buffer has a value other than 0x701.

The host status bit mask values, which provide detailed status information, are described in file “**U_API.h**” (see **Appendix J**).

For TCP or X.25 host connections,

the status value returned in the buffer has the following meanings:

- 0 (zero):
The host connection is not available.
(The application should not send data while the connection is unavailable.)
- 701 hexadecimal:
The host connection is available and operational
(insofar as the Gateway is able to verify).
The application is permitted to send data messages.
- If the returned status buffer contains any other value:
The host connection may be, or may not be available and operational.

The application is permitted to send data messages, even though the host connection may not be operational. If the Gateway's connection to the host has been lost, messages cannot immediately reach the host, but the Gateway will attempt to re-establish the connection and then send the messages.

To verify message delivery, the application should check for any expected responses from the host. If the application receives no response from the host (within a reasonable time period after sending a message), this may indicate that the host is currently not available for communications.

If an error occurs in the course of obtaining the host connection status information, **IateControl** returns a negative error code. Refer to **Appendix A** for information about IATE API error values and their causes.

Example:

```
long status;  
  
ret = IateControl (refnum, APIGetHostStat, (unsigned char *) &status);  
if (ret >= 0)  
    printf ("APIGetHostStat returned %d, status code %04X", ret, status);
```

APIGetTaStat

Purpose:

This command finds out whether or not the API has received any host messages or segments which the application can retrieve immediately using **IateRead**.

In general it is more efficient to simply use **IateRead** and process any data received, without using **APIGetTaStat**.

The behavior of this command depends in part on whether the channel is in Segment Mode or Message Mode. (See **APISetSegment** and **APISetMsg**.)

Arguments:

The first argument to **IateControl** specifies the TA object connection to which this command applies. . The second argument is the **APIGetTaStat** command. The third argument is ignored and should be zero.

Returns:

IateControl returns a value of zero or greater to indicate the number of segments that the API has received, which the application can retrieve immediately using **IateRead**.

Even if **IateControl** returns a positive value to indicate an available message, a subsequent call to **IateRead** may return zero, indicating no data received.

This behavior is caused by message queuing logic in the API.

In general it is more efficient to simply use **IateRead** and process any data received, without using **APIGetTaStat**.

If an error occurs, **IateControl** returns a negative error code.

Refer to **Appendix A** for information about IATE API error values and their causes.

See Also:

APISetSegment

APISetMsg

Example:

```
ret = IateControl (refnum, APIGetTaStat, (unsigned char *) 0);  
if (ret >= 0)  
    printf ("There are %d messages waiting to be read", ret);
```

APIGetTaThrottle

Purpose:

This command finds out whether or not sufficient time has elapsed between **IateWrite** calls, so that the application may issue the next **IateWrite** call on the specified connection.

After an application calls **IateWrite** on a given object connection, the API will not accept another **IateWrite** on the same object, until the API Throttle Interval time period has elapsed.

The Gateway configuration item `API_THROTTLE_INTERVAL` sets the throttle interval time. Its default value is one second.

Arguments:

The first argument to **IateControl** specifies the TA object connection to which this command applies.

Returns:

IateControl returns a value greater than zero if the throttle interval has elapsed, and the application can call **IateWrite** on the specified object. **IateControl** returns zero if the throttle interval has not yet elapsed since the application's last call to **IateWrite** on this object.

If an error occurs, **IateControl** returns a negative error code.

Refer to **Appendix A** for information about IATE API error values and their causes.

Example:

```
ret = IateControl (refnum, APIGetTaThrottle, (unsigned char *) 0);
if (ret >= 0)
    printf(
        "IateWrite %s be issued on connection %d at this time",
        ret ? "may" : "may not",
        refnum);
```

APIGetObjectConfig

Purpose:

This command retrieves Gateway configuration information for the specified TA object. This information is available only after the application has successfully connected to the object by using **IateOpen**.

Arguments:

The first argument to **IateControl** specifies the TA object connection to which this command applies. The second argument is the **APIGetObjectConfig** command. The third argument is a buffer in which this command returns the configuration information (**struct u_link_response**).

Example:

```
#include "U_APItypes.h"

struct u_link_response config;
IateControl (refnum, APIGetObjectConfig, (unsigned char *) &config);
```

Following is a listing of **struct u_link_response**:

```
struct u_link_response
{
    char
        iata_str[10];        /* IA and TA (ALC values) */

    unsigned short
        protocol,           /* host type, defined in U_CMNhos.h */
        throttle_limit,    /* time between IateWrites,
                           set in gateway configuration file */
        port;               /* port number (for ALC), defaults to 0 */
        board;              /* board number (for ALC), defaults to 0 */

    char
        server_ver[10];     /* server version */

    unsigned char
```

```

    asc_eompb,          /* EOM characters defined in serverde.h */
    asc_eomc,
    asc_eomu,
    asc_eomi;

unsigned char
    alc_ia,            /* IA */
    alc_ta;           /* TA */

short
    object_type;      /* object type:
                       1: TERMINAL
                       2: PRINTER
                       3: TERMINAL_API
                       4: PRINTER_API */

short
    gate_type;        /* gateway type:
                       1: Windows or UNIX gateway
                       2: Mac gateway */

char
    object_name[MAX_CLIENT_NAME+2];
                       /* name of object */

unsigned char
    answer_back_rules,
    expect_aid,        /* AID character for CPARS */
    default_aid,       /* starting inbound AID for CPARS */
    fill[1];
};

```

APISendAck

Purpose:

This command instructs the Gateway to send a message/segment acknowledgment to the host, if necessary.

Arguments:

The first argument to **IateControl** specifies the TA object connection to which this command applies. The second argument is the **APISendAck** command. The third argument is a character value which specifies the acknowledgment type, which may be either **NORMAL_ANS** or **UNABLE_TO_ACCEPT**.

The IATE header file **iate_pub.h** defines **NORMAL_ANS** and **UNABLE_TO_ACCEPT**, among several other acknowledgment values. The application should only use one of these two values with this command.

```
#define NORMAL_ANS      '0'      /* positive acknowledgment */
#define UNABLE_TO_ACCEPT '4'      /* negative acknowledgment */
```

An application handling traffic for a printer or ticket-imaging object uses **APISendAck** to acknowledge each segment.

Example:

Following is sample code for receiving and acknowledging messages while in Segment Mode. This sample program links to a printer object, receives printer data, acknowledges it, and writes it to a file.

```

/* ----- sendack.c ----- */

#include <fcntl.h>
#include <winsock.h>
#include <stdio.h>

#include "U_API.h"
#include "U_APItyp.h"
#include "U_APIpros.pro"

/* The following functions, defined in this example,
 * will also be used by other sample programs in this document.
 */
void Setup (void);
void Connect (void);
void Disconnect (void);
int userBreak (void);

/* Parameter values for APISendAck
 */
#define NORMAL_ANS '0'
#define UNABLE_TO_ACCEPT '4'

/* Parameter values for APIPrinterStat
 */
#define PAVAIL "1"
#define PUNAVAIL "0"

/* Connection specifier for IateOpen, to
 * connect to a printer object named "printer02"
 * on gateway host system "gw2"
 */
unsigned char
connection_specifier[] = "@gw2\\ialcserver\\printer02";

/* IATE session reference number
 */
long refnum;

/* Buffers for messages received via IateRead
 */
unsigned char
buff[MAX_BUFF_SZ], /* message buffer */
ctrl[CTRL_BLK_SZ]; /* control block: C1, C2, EOMx, CCC_OK, MORE */
/* (See the IateRead documentation) */

main()
{
    long ret,
        nchars;

    int fd;

```

```
unsigned char ack;

fd = open("printer.log", O_WRONLY | O_CREAT);

if (fd < 0)
{
    printf("Can't open printer.log file for writing");

    exit(1);
}

/*
 * The Setup() function sets the API diagnostic level of 0x2ff,
 * and sets the minimum time required between successive
 * IateOpen calls to 10 seconds.
 *
 * The Connect() function calls IateStart,
 * sets the IateRead blocking interval to 10 seconds,
 * requests a connection to the printer object,
 * and tells the Gateway that the printer is available.
 */

Setup();
Connect();
```

```

/*
 * Read messages from the host, acknowledge them,
 * and write them to a file.
 */

while (!userBreak())
{
    /*
     * Read a message from the host.
     */

    nchars = IateRead(refnum, MAX_BUFF_SZ, buff, ctrl);

    /*
     * IateRead returns the number of characters in the
     * host message, plus 2 control characters.
     * Refer to the IateRead documentation.
     */

    if (nchars < 0)
        ack = UNABLE_TO_ACCEPT;
    else
    {
        /*
         * A valid message segment checksum indicator
         * is equal to '1' (0x31). If the checksum indicator
         * value is '0' (0x30), it indicates an invalid segment.
         */

        if (ctrl[CTRL_CCC_OK] == 0x31)
        {
            ret = write(fd, buff, nchars-2);

            ack =
                (ret == (nchars-2))
                ? NORMAL_ANS
                : UNABLE_TO_ACCEPT;
        }
    }

    /*
     * The printer needs to acknowledge each received segment,
     * by sending an APISendAck, whether the segment is
     * valid or not.
     */

    IateControl(refnum, APISendAck, &ack);

    if (ack == NORMAL_ANS)
        printf ("Host message written to log file.\n");
}

```



```

    /*
     * Disconnect from the Gateway and API
     */

    Disconnect();
}

/*
 * The Setup() function sets the API diagnostic level of 0x2ff,
 * and sets the minimum time required between successive
 * IateOpen calls to 10 seconds.
 */

void
Setup ()
{
    long x;

    short val;

    /*
     * Set the API Debugging verbosity level.
     * (Refer to Appendix G.)
     */

    x = 0x2ff;
    IateControl(
        (long) 0,
        APISetApiDebug,
        (unsigned char *) &x);

    /*
     * Set the API Open Delay time.
     */

    val = 10; /* seconds */
    IateControl(
        (long) 0,
        APISetOpenDelay,
        (unsigned char *) &val);
}

```

```

/*
 * The Connect() function calls IateStart,
 * sets the IateRead blocking interval to 10 seconds,
 * requests a connection to the printer object,
 * and tells the Gateway that the printer is available.
 */

void
Connect ()
{
    struct timeval to;

    /*
     * Initialize the API.
     */

    startcode = IateStart(1, 0, (unsigned char *) "");

    if (startcode < 0)
    {
        printf(
            "IateStart failed (error %d)\n",
            startcode);

        exit(2);
    }

    /*
     * Open a connection to the printer object.
     */

    refnum =
        IateOpen(
            startcode,
            APILinkToName,
            connection_specifier);

    if (refnum < 0)
    {
        printf("\nIateOpen failed (error %d)\n", refnum);
        IateStop(startcode);

        exit(3);
    }
}

```

```

/*
 * Set the IateRead blocking interval to 10 seconds.
 */

to.tv_usec = 0;
to.tv_sec  = 10;

IateControl(
    refnum,
    APISetTO,
    (unsigned char *) &to);

/*
 * Tell the Gateway that the printer is available.
 */

IateControl(
    refnum,
    APIPrinterStat,
    (unsigned char *) PAVAIL);
}

/*
 * The Disconnect() function disconnects the
 * application from the Gateway and API.
 */

void
Disconnect (void)
{
    /*
     * Set printer status to 'unavailable'
     */

    IateControl(
        refnum,
        APIPrinterStat,
        (unsigned char *) PUNAVAIL);

    /*
     * Close the printer object connection
     */

    IateClose(refnum);

    /*
     * Terminate this application's use of the API
     */

    IateStop();
}

```

```

/*
 * The userBreak() function should return a nonzero value
 * if the user has requested termination of the program.
 * After this function returns nonzero,
 * the caller should terminate the program gracefully.
 */

int
userBreak(void)
{
    /*
     * The body of this function is not shown here.
     * The means of detecting user input depends on the
     * platform (Windows or UNIX), the type of application,
     * and choice of implementation.
     *
     * For example, in a console program, this function could
     * work with a signal handler to detect a Ctrl-C keystroke.
     * After the user presses Ctrl-C to terminate the program,
     * this function would return nonzero, and the caller
     * would proceed to terminate the program.
     */

    /*
     * ... Insert code here, to return nonzero
     *    if the user has requested program termination ...
     */

    return 0;
}

/* ----- */

```

APIPrinterStat

Purpose:

The **APIPrinterStat** command sends printer status information to the Gateway.

Any application that opens a printer TA object, and processes print data from the host, should use this command. A “printer TA object” is one specified with type PRINTER or PRINTER_API in the IATE Gateway's configuration, corresponding to a printer TA as defined in the airline host system.

The application informs the Gateway as to whether or not the destination output device is ready to process data messages from the host.

The destination device may be a printer, or some other device being used in place of a printer -- wherever the application sends the 'printed' data.

If the output destination device is known to be always available -- or if the application has no way to obtain its current status -- the application may choose to issue **APIPrinterStat** with PAVAIL just once, to declare the object permanently available.

Arguments:

The first argument to **IateControl** is the reference number of the TA object connection on which to set the printer status.

The second argument to **IateControl** is the **APIPrinterStat** command.

The third argument to **IateControl** specifies the status value: nonzero to indicate that the printer destination is available for messages, or zero to indicate that it is not available. See the example below.

Example:

```
/* Status values for APIPrinterStat:
 */
#define PAVAIL    "1"      /* printer available */
#define PUNAVAIL "0"      /* printer unavailable */

/* Tell the Gateway that the printer is available or not:
 */
if (printerAvail())      /* <-- a function defined by the application */
{
    IateControl (refnum, APIPrinterStat, (unsigned char *) PAVAIL);
}
else
{
    IateControl (refnum, APIPrinterStat, (unsigned char *) PUNAVAIL);
}
```

APIInoTaTimeout

Purpose:

The **APIInoTaTimeout** command disables the Gateway's “TA timeout” for the specified object. This command works only if the Gateway has not been configured to expect “heartbeat” keep-alive messages from the application. (By default the Gateway does not expect heartbeats.)

The Gateway's TA_TIMEOUT configuration item specifies the TA Timeout in minutes. This timeout can be specified in the Gateway configuration file.

If the TA Timeout elapses with no messages sent from the application, the Gateway may disconnect the application. If the application issues the **APIInoTaTimeout** command, this can prevent such disconnection.

The purpose of the Heartbeat option is to detect a “crashed” application's failure to disconnect from the Gateway. Contrast this to the TA Timeout, the purpose of which is to protect against idle applications keeping TAs occupied. See **APISetHeartbeat** for more information about heartbeats.

When expecting heartbeats, the Gateway does not use the TA Timeout, because the 60-second heartbeat timeout overrides it.

Arguments:

The first argument to **IateControl** is the reference number of the TA object connection on which to disable the TA Timeout.

Example:

```
IateControl (refnum, APIInoTaTimeout, 0);
```

See Also:

APISetHeartbeat

Gateway configuration items:
TA_TIMEOUT
HEARTBEAT_REQUIRED

APIGetVersion

Purpose:

The **APIGetVersion** command retrieves a text string containing the the API version number.

Arguments:

The first argument to **IateControl** should be zero for this command, because this command does not operate on a specific TA object.

The second argument to **IateControl** is the **APIGetVersion** command.

The third argument to **IateControl** specifies a buffer to receive the version string. The buffer should be at least 9 bytes in length.

Returns:

Current versions of the API return a version string in the form "2.XX.YY", where XX and YY are numeric values which will depend on the version installed.

The initial value 2 indicates that the running API is a member of the second generation of IATE API releases. The XX value indicates the major incremental release level, and the YY value indicates the minor incremental release level.

Example:

```
char buff[9];
IateControl (0, APIGetVersion, buff);
printf ("API version %s\n", buff);
```

APISetHeartbeat

Purpose:

The **APISetHeartbeat** command tells the API whether or not to allow the application to trigger “heartbeat” (keep-alive) messages to the IATE Gateway.

If the API uses this command to enable heartbeats, then the client application must use the **APIStart1min** command to send heartbeats to the Gateway.

If the Gateway's HEARTBEAT_REQUIRED configuration option is turned on, the Gateway expects the client application to send periodic heartbeat messages. If the Gateway's HEARTBEAT_REQUIRED option is turned off, the Gateway will not expect heartbeat messages from the client, unless the client begins to send them.

When the Gateway expects heartbeats, the client application should send heartbeats and/or data messages at periodic intervals no longer than 50 seconds (leaving a 10-second margin under the 60-second time limit). If 60 seconds elapse with no message received from the client, the Gateway disconnects the client.

The purpose of Heartbeats is to protect against idle applications keeping TAs occupied. Contrast this to the TA Timeout, the purpose of which is to detect a “crashed” application's failure to disconnect from the Gateway. See **APISetTO** for information about the TA Timeout.

When expecting heartbeats, the Gateway does not use the TA Timeout configuration item, because the 60-second heartbeat timeout overrides it.

Arguments:

The first argument to **IateControl** is the reference number of the TA object connection on which to set the Heartbeats option.

The second argument is the **APISetHeartbeat** command.

The third argument is nonzero to enable heartbeats, or zero to disable them.

See Also:

APIStart1min

Gateway configuration item:

HEARTBEAT_REQUIRED

Examples:

To enable heartbeats:

```
short val = 1; /* 1 = enable */
IateControl (refnum, APISetHeartbeat, (unsigned char *) &val);
...
...
IateControl (refnum, APIStartlmin, 0);
/* (periodically at intervals less than 60 seconds) */
```

To disable heartbeats:

```
short val = 0; /* 0 = disable */
IateControl (refnum, APISetHeartbeat, (unsigned char *) &val);
```

APIStart1min

Purpose:

The **APIStart1min** command sends a heartbeat message to the Gateway.

If the application will use **APIStart1min**, the application must first use **APISetHeartbeat** to enable heartbeat transmission.

See also:

APISetHeartbeat

Example:

```
IateControl (refnum, APIStart1min, (unsigned char *)"");
```

APIResetLock

Purpose:

This command resets the API's write lock.

When an application calls **IateWrite**, the API 'locks' the session against further writes. The application cannot issue another **IateWrite** on the session until:

- (1) a call to **IateRead** returns a response message from the host, or
- (2) the application issues **APIResetLock**.

It takes time for the host to respond to a message. Therefore, it is recommended that the application wait for a time somewhat longer than the host's typical response time. After waiting a reasonable amount of time but obtaining no response, the application can reset the write lock if necessary.

Note for Windows platforms only:

APIResetLock flushes messages queued from the Gateway to the host, for the object session specified by the first argument. Alternatively, **APIResetLocal** can be used to reset the write lock without flushing queues.

Note for UNIX platforms only:

APIResetLock does not flush the Gateway-to-host message queues. However, **APIForwardReset** can be used to flush the message queues.

Arguments:

The first argument to **IateControl** is the reference number of the TA object connection on which to reset the write lock and flush message queues. The second argument is the **APIResetLock** command. The third argument is ignored and should be zero.

See also:

APIResetLocal
APIForwardReset
APISetTO

Examples:

```
long refnum;
IateControl (refnum, APIResetLocal, (unsigned char *) 0);
```

The following example demonstrates a typical use of APIResetLock in a simplified messaging application.

```
/* ----- resetlock.c ----- */

#include <stdio.h>
#include "U_API.h"
#include "U_APItypes.h"
#include "U_APIpros.pro"

/* The following routines are defined in the APISendAck example.
 * These routines may need to be modified to suit this example.
 */
void Setup(void);
void Connect(void);
void Disconnect(void);
int userBreak(void);

/* Connection specifier for IateOpen, to
 * connect to a terminal TA object named "term16"
 * on gateway host system "gw2"
 */
unsigned char
    connection_specifier[] = "@gw2\\ialcserver\\term16";

/* IATE session reference number
 */
long refnum;

/* Buffers for messages received via IateRead
 */
unsigned char
    buff[MAX_BUFF_SZ],          /* message buffer */
    ctrl[CTRL_BLK_SZ];        /* control block: C1, C2, EOMx, CCC_OK, MORE */
                                /* (See the IateRead documentation) */

main()
{
    struct timeval to;
    long          nchars;
    int           ret;
}
```

```

unsigned char  ack;

/*
 * The Setup() function sets the API diagnostic level of 0x2ff,
 * and sets the minimum time required between successive
 * IateOpen calls to 10 seconds.
 *
 * The Connect() function calls IateStart,
 * sets the IateRead blocking interval to 10 seconds,
 * requests a link to the printer object,
 * and tells the Gateway that the printer is available.
 */

Setup();
Connect();

/*
 * Set the IateRead blocking interval to 2 seconds (as an example).
 */

to.tv_usec = 0;
to.tv_sec = 2;
IateControl (refnum, (long) APISetTO, (unsigned char *) &to);

/*
 * Send messages to the host,
 * and read any responses from the host.
 */

while (!userBreak())
{
    /*
     * Send the host an empty message.
     * (This is only a simplified demonstration.
     * A useful application would send a non-empty message.)
     */

    ret = IateWrite (refnum, 0, (unsigned char *) "");
    if (ret < 0)
        break;

    /*
     * Wait for a response message, up to the amount of time
     * we specified through APISetTO above.
     *
     * If no response arrives within that amount of time,
     * then we must reset the API write-lock, in order to
     * allow subsequent calls to IateWrite.
     */

    nchars = IateRead (refnum, MAX_BUFF_SZ, buff, ctrl);
    if (nchars == 0)
    {

```

```
        printf(
            "No response received.  Resetting the write lock ...\n");
        IateControl (refnum, APIResetLock, (unsigned char*) 0);
    }
}

/*
 * Disconnect from the Gateway and API.
 */
Disconnect();
}

/* ----- */
```


APIResetLocal

Purpose:

APIResetLocal is similar to **APIResetLock**, but does not flush the Gateway-to-host message queues. For details, please refer to **APIResetLock** (above).

Arguments:

The first argument to **IateControl** is the reference number of the TA object connection on which to reset the write lock.
The second argument is the **APIResetLock** command.
The third argument is ignored and should be zero.

Example:

```
long refnum;  
IateControl (refnum, APIResetLocal, (unsigned char *) 0);
```

APIForwardReset

Purpose:

On Windows platforms, this command is equivalent to **APIResetLock**: it resets the API write lock, and flushes Gateway-to-host message queues.

On UNIX platforms, where **APIResetLock** does not flush the message queues, this command can be used to do so.

This call is implemented in Windows and Sun PCI gateways version 2.5 or later, Windows API version 2.4.11 or later, and Sun PCI API version 2.4.9 or later.

Note:

It is recommended that applications periodically check the host connection status by using **APIGetHostStat**. The application should stop sending messages if the host connection has been lost. (It is not sufficient to reset the lock and send additional messages without checking the host status.)

On Sun Sbus (not PCI) systems, **APIForwardReset** is ignored.

Arguments:

The first argument to **IateControl** is the reference number of the TA object connection on which to reset the write lock and flush message queues. The second argument is the **APIForwardReset** command. The third argument is ignored and should be zero.

Example:

```
long refnum;  
IateControl (refnum, APIForwardReset, (unsigned char*) 0);
```

APIWhoAmI

Purpose:

The **APIWhoAmI** command retrieves the IATE Gateway's information about the type of the connected airline-host.

The information retrieved consists of a host type-code (a number) and a host type-name (a string).

The host codes are defined in the header file **U_CMNhos.h**,
Host type names are defined in **cmdnames.c**.

Note:

Only the Macintosh Gateway supports the Uniscope, Codacom, and AC100 host types. The Windows and UNIX gateways do not support those host types.

Arguments:

The first argument to **IateControl** is the reference number of the TA object connection on which to obtain host type information.

The second argument to **IateControl** is the **APIWhoAmI** command.

The third argument to **IateControl** points to the string buffer in which **APIWhoAmI** will return the host type name. The string buffer should be at least 16 bytes long. Some additional space is recommended, to accomodate any future host names which may be slightly longer.

Returns:

APIWhoAmI returns the host type code (as the **IateControl** function's return value), and the host type name (in the string buffer given by the third argument).

Example:

```
long refnum;
char buff[32];
ret = IateControl (refnum, APIWhoAmI, buff);

printf("Host type \"%s\", #d", buff, ret);
```

Peer-to-Peer Messages

The following LateControl commands support peer-to-peer messaging:

APIQueryApplMsg
APIGetApplMsg
APISendApplMsg

Peer-to-peer messages may be sent between any two objects linked to a single gateway.

A sample application, **sendpeer.c**, has been provided as an example of peer-to-peer messaging. Two instances of **sendpeer.c**, connected to two different TA objects respectively, communicate through peer-to-peer messages.

To use the peer-to-peer sample, set up 2 objects on a gateway; called (for example) “Object1” and “Object2”.

Start two instances of the sample program, using each of the two objects to talk to the other one:

```
sendpeer -c@HostName\\ServiceName\\Object1 -p@HostName\\ServiceName\\Object2  
sendpeer -c@HostName\\ServiceName\\Object2 -p@HostName\\ServiceName\\Object1
```

The **-c** command-line option specifies the source object, which is to send a peer-to-peer message to the second object, specified by the **-p** option.

By default, the program takes message text from the keyboard. Enter a line of text at the keyboard and press Enter, and the program will attempt to send that text in a peer-to-peer message to the remote object.

If you wish to send a message from a file (instead of sending from the keyboard), specify the **-f** option with the file-name. For example:

```
sendpeer -c@Host\\Service\\Object1 -p@Host\\Service\\Object2 -fFileName1  
sendpeer -c@Host\\Service\\Object2 -p@Host\\Service\\Object1 -fFileName2
```

The API may break a long message into segments while transmitting it to the Gateway and the peer object. The receiving peer should send back acknowledgment messages which indicate whether or not the peer processed each segment successfully. The sending peer’s API processes these acknowledgments and continues to send message segments, until the entire message has been transmitted (or until an error occurs).

After transmitting the entire message to the peer (or after detecting an error), the API passes the final acknowledgment code back to the sending application. The acknowledgment code indicates success or failure.

Peer-to-peer messages begin with the **struct u_applmsghdr** message structure, defined in the **U_APItypes.h** header file.

The application sends and receives peer messages in a buffer which begins with that structure, followed by the message data. The structure contains a field (**datalen**) that indicates the length of the message data, and a command code (**cmmd**) which indicates the type of message.

These peer-to-peer command codes are defined in the **U_API.h** header file:

PTRdataMsgRsp	A complete data message, or the final part of a multi-part data message.
PTRcontMsgRsp	The first part, or a continuation, of a multi-part data message.
PTRrspDone	Positive acknowledgment of a data message received.
PTRrspOffline	Negative acknowledgment: Printer is off-line.
PTRprinterFail	Negative acknowledgment: Printer failed.
PTRnotAllowed	Negative acknowledgment: Print not allowed on this TA.
PTRbusy	Negative acknowledgment: Printer is busy.
PTRrspIOfail	Negative acknowledgment: I/O failed.
PTRforwardReset	A “forward reset” control-message: flushes message traffic in one direction.

Positive acknowledgment messages have a command code of **PTRdataMsgRsp**. The various command codes for negative acknowledgments are also listed above. Acknowledgment messages have a zero value in the length field (**datalen**). See **APISendApplMsg** for an example of a peer acknowledgment message.

Each of the following calls uses the message header (**struct applmsghdr**) defined in **U_APItypes.h**.

APIQueryApplMsg

Purpose:

This command checks to see if there is any peer-to-peer traffic available for the application to retrieve.

Arguments:

The first argument to **IateControl** is the reference number of the TA object connection on which to check for peer-to-peer messages. The second argument is the **APIQueryApplMsg** command. The third argument is ignored and should be zero.

Returns:

IateControl returns zero if there is no peer message ready.

If a message is available, **IateControl** returns a value greater than zero. The application can retrieve the message via **APIGetApplMsg**.

If an error occurs, **IateControl** returns a negative error code. Refer to **Appendix A** for information about IATE API error values and their causes.

Example:

```
ret = IateControl (refnum , APIQueryApplMsg, (unsigned char *) 0);
if (ret >= 0)
    printf ("There %s peer messages waiting",
           ret ? "ARE" : "ARE NOT");
```

APIGetApplMsg

Purpose:

The **APIGetApplMsg** command retrieves a peer-to-peer message.

The application supplies the buffer (**buff**) to receive the message.

The peer-to-peer message header (**struct u_applmsghdr**) comes first at the beginning of the buffer, with length equal to **APPLMSGHDR_SZ**.

The message data follows the header, starting at offset **APPLMSGHDR_SZ**.

Arguments:

The first argument to **IateControl** is the reference number of the TA object connection on which to receive a peer-to-peer message.

The second argument to **IateControl** is the **APIGetApplMsg** command.

The third argument to **IateControl** is the address of the buffer in which to receive a peer-to-peer message. The application initializes the data length field (**datalen**) of the message header in the buffer, to **MAX_BUFF_SZ** (2015 characters).

Returns:

IateControl returns the total length of the peer-to-peer message received, or zero if no message was received.

If a message was received, the returned length value includes the length of the peer-to-peer message header (**struct u_applmsghdr**). The length of the message data proper can be found in the header's length field (**datalen**).

A peer-to-peer message may arrive in multiple parts (or 'segments'). If the received message data is only the first of multiple segments, the command field (**cmmd**) contains **PTRcontMsgRsp**, and the continuation field (**more**) contains a nonzero value. Subsequent peer-to-peer message(s) contain the subsequent parts of the data, and the final message of the series has command code **PTRdataMsgRsp**.

Example:

```
char buff[MAX_BUFF_SZ];  
...  
struct applmsghdr *pmhdr = (struct applmsghdr *) buff;  
pmhdr->cmmd = 0;  
pmhdr->datalen = MAX_BUFF_SZ;  
ret = IateControl (refnum, APIGetApplMsg, buff);
```

APISendApplMsg

Purpose:

The **APISendApplMsg** command sends a peer-to-peer message.

The application supplies the buffer (**buff**) that contains the message.

The peer-to-peer message header (**struct u_applmsghdr**) comes first at the beginning of the buffer, with length equal to **APPLMSGHDR_SZ**.

The message data follows the header, starting at offset **APPLMSGHDR_SZ**.

The maximum length of a message that can be sent at one time is 2015 characters of data (**MAX_BUFF_SZ**), plus the peer-to-peer header (**APPLMSGHDR_SZ**).

Arguments:

The first argument to **IateControl** is the reference number of the TA object connection on which to send a peer-to-peer message.

The second argument to **IateControl** is the **APISendApplMsg** command.

The third argument to **IateControl** is the address of the buffer from which to send a peer-to-peer message. The application initializes the data length field (**datalen**) of the message header in the buffer, to indicate the length of the message, up to **MAX_BUFF_SZ** (2015 characters).

Examples:

The sample program **seedpeer.c**, supplied with the IATE API software distribution, demonstrates sending and receiving of peer-to-peer messages.

The following examples use the **ExtractFromNameString** function, which can be found in the **seedpeer.c** sample program.

ExtractFromNameString extracts the Gateway host name, TCP/IP service name, and TA object name from a null-terminated string buffer formatted as follows:

```
"@HostName\\ServiceName\\ObjectName"
```

In this example the application that sends a data message is using the

TA object “Object1”. The application that receives the data message is using the TA object “Object2”.

The application receiving the message must acknowledge it, by sending a peer-to-peer acknowledgment message. The API for the sending application receives and process the acknowledgment, without involving the sending application. (The sending application itself does not receive the peer acknowledgment, so it does not implement any logic to process peer acknowledgments.)

This is example code for the sending application:

```

struct applmsghdr
    *pmhdr = (struct u_applmsghdr *) buff;
                                /* this buffer contains the peer-to-peer
                                message header, followed by data */

long nchars = strlen (buff);

pmhdr->cmmd = PTRdataMsgRsp; /* command code for a data message */
pmhdr->datalen = nchars; /* length of message data (after header) */
pmhdr->more = 0; /* indicate that this message is complete */

ExtractFromNameString (
    pmhdr->ToHostName, /* host name extracted from 4th argument */
    pmhdr->ToServerName, /* server name extracted from 4th arg. */
    pmhdr->ToObjectName, /* object name extracted from 4th arg. */
    "@HostName1\\ServiceName1\\ObjectName1");

ExtractFromNameString (
    pmhdr->FromHostName, /* host name extracted from 4th argument */
    pmhdr->FromServerName, /* server name extracted from 4th arg. */
    pmhdr->FromObjectName, /* object name extracted from 4th arg. */
    "@HostName\\ServiceName\\Object2");
                                /* send to Object2 (as an example only) */

ret = IateControl (refnum, APISendApplMsg, buff);

```

This is example code for the application that receives and acknowledges the data message:

```

char buff[MAX_BUFF_SZ];

struct applmsghdr
    *pmhdr = (struct applmsghdr *) buff;
                                /* a buffer to receive a message
                                and to send an acknowledgment */

/* Receive a peer-to-peer data message: */

```

```

pmhdr->cmmd = 0;
pmhdr->datalen = MAX_BUFF_SZ;
ret = IateControl (refnum, APIGetApplMsg, buff);

/* Send a peer-to-peer acknowledgment message: */

ExtractFromNameString (
    pmhdr->ToHostName,      /* host name extracted from 4th argument */
    pmhdr->ToServerName,   /* server name extracted from 4th argument */
    pmhdr->ToObjectName,  /* object name extracted from 4th argument */
    "@HostName\\ServiceName\\Object1");
                        /* send to Object1 (as an example only) */

pmhdr->cmmd = PTRrspDone; /* msg has arrived and printed */
pmhdr->datalen = 0;      /* data length is 0 for acknowledgements */
pmhdr->more = 0;        /* this is a complete message */

ret = IateControl (refnum, APISendApplMsg, buff);

```

The foregoing example uses the **PTRrspDone** code for positive acknowledgment. Additional acknowledgment command codes are defined in the **U_API.h** header file, as described in the **Peer-to-Peer Messages** section above.

APIForceSeperateSockets

Purpose:

This command forces the API to open a separate socket for each connection to any TA object. This will guarantee a unique socket file-descriptor associated with each object connection and reference-number.

A Windows application that opens multiple TA connections in separate threads must use this command before the first call to **IateOpen**. If the application does not turn on this option, the IATE API for Windows cannot support multiple connections serving multiple application threads.

A UNIX application opens multiple TA connections in separate processes must use this command before the first call to **IateOpen**. If the application does not turn on this option, the IATE API cannot support multiple connections serving multiple UNIX application processes.

Arguments:

The first argument to **IateControl** should be zero for this command, because this command does not operate on a specific TA object.

The second argument to **IateControl** is the **APIForceSeperateSockets** command.

The third argument to **IateControl** is nonzero to turn on the separate sockets option, or zero to turn it off. (The option is off by default, so an application should have no need to turn it off explicitly. Simply leave it off if it's not needed, or turn it on as shown below.)

Example:

```
short x = 1; /* nonzero to turn on the separate sockets option: */  
IateControl (0, APIForceSeperateSockets, (unsigned char *) &x);
```

Appendix A: Error Codes

This appendix explains the error codes returned by IATE API functions.

Error -2002: ServerUnreachable / NoServerError

Returned By: `IateOpen`

Explanation:

This error indicates that the API cannot connect to the Gateway host system.

One possible cause is that the application may have specified an invalid Service Name in the name argument to **IateOpen**. The application or its configuration should be modified to issue **IateOpen** with a correct Service Name.

This error indicates that the requested connection has not been established. The application should not call **IateRead**, **IateWrite**, **IateControl**, or **IateClose**, on a session that has not been established. If the application does so, the results are undefined, although those functions might also return this error.

For example:

If the name argument to **IateOpen** was:

```
@HostName\\ServiceName\\TaObjectName
```

then error -2002 means the ServiceName was invalid.

See **Appendix I** for more information about the Service Name.

See Also: **Error -2204, HostUnreachable.**

Error -2003: OutOfBufferError

Returned By: **IateOpen**
IateRead
IateWrite
IateControl
IateClose

Explanation:

The API has run out of internal memory buffers. This should not happen under normal conditions, but may occur because of inappropriate usage of the API, or because of abnormal communication conditions between the API and the Gateway.

To track down the cause of this error, begin by verifying that the IATE Gateway appears to be functioning. The API Out-of-Buffers error is sometimes a side-effect of a failure at the Gateway.

If the Gateway has reported an error or stopped operating, the Gateway needs to be restarted; and if the trouble recurs, the problem with the Gateway needs to be diagnosed.

If the Gateway appears to be functioning, but the API still reports the Out-of-Buffers error, then investigate the application design. Investigate the possibility that the application has overloaded the API with continuous commands, or with message data sent or received. (For example, this could possibly happen in an application that sends terminal messages on a terminal TA and also receives printer traffic from a printer TA.)

Error -2004: ObjectUndefined / NamelsBad

Returned By: IateOpen

Explanation:

This error means that the **IateOpen** call failed because the requested TA Object is not configured at the Gateway.

For **IateOpen** with the **APILinkToName** command (or the **APILinkToDyCrt** or **APILinkToDyPrt** command), this error means that the specified Object Name does not match any object name or group name in the Gateway's configuration.

For example, suppose the name argument to **IateOpen** was:

```
@HostName\\ServiceName\\ObjectName
```

The IATE Gateway, running on the specified HostName, has a configuration file associated with the specified ServiceName, but the specified ObjectName (or group name) does not appear in that file's TA objects list.

Typically this indicates a spelling error in the ObjectName that the application requested. In order to connect to the object successfully, the application must specify a correct object name (or the name of an object-group) in the call to **IateOpen**.

In addition to the spelling, the upper or lower case of each letter should also match the Gateway's configuration.

Object names should not contain any blank spaces. Object names should begin with a letter or digit, and should contain only letters and digits, and possibly underscores. Other characters or punctuation generally should not be used.

This error can also occur for the **APILinkToTa** command, which is used by some legacy applications (not recommended for new ones). That command specifies the IA number and TA number (rather than the name) of the object to connect. If the Gateway has no object configured with that IA and TA, **IateOpen** will return this error code.

Error -2005: NameInUse

Returned By: **IateOpen**

Explanation:

For **IateOpen** with the **APILinkToName** command, this error indicates that the specified TA Object Name matches one of the Gateway's configured object names, but that object is already in use by an application.

Similarly, for the **APILinkToDyCrt** or **APILinkToDyPrt** command, this error indicates that any objects matching the request were already in use by an application.

A new connection to an object cannot be established while that application is using it. Each TA object admits only one application connection at a time. The object may become available again later, when that application relinquishes the object by calling **IateClose** (or when the Gateway disconnects from the application for time-out or other reasons).

This error can also occur for Intercept-mode connections (attempted with **IateOpen** and the **APIInterceptName** command), if an application is already intercepting the specified object. Each TA object admits only one intercepting application at a time.

Error -2007: DataError

Returned By: **IateOpen**
IateWrite

Explanation:

This error indicates one of the following conditions. Upon receiving this error, it is the responsibility of the application programmer or tester to determine which of these cases applies:

1. Bad name argument to **IateOpen**:

The name argument that the application passed to **IateOpen** did not contain a TA Object specifier following the Host Name and Service Name.

In this case, the application should be corrected to supply a valid and complete name argument to **IateOpen**, in the required format:

```
@HostName\\ServiceName\\ObjectName
```

2. Bad message argument to **IateWrite**:

- a) The application passed the NULL value in place of the message buffer argument to **IateWrite**, or
- b) The application passed a negative value in place of the message-length argument to **IateWrite**.

In these cases, the application should be corrected to pass a valid message buffer, and nonnegative length value, to **IateWrite**.

Error -2008: NotStartedError

Returned By: IateOpen
IateRead
IateWrite
IateClose
IateStop

Explanation:

An application has made one the IATE API calls listed above, without first calling the **IateStart** initialization function.

The application should call **IateStart** before calling any of the other IATE API functions.

Error -2009: BadVersionError

Explanation:

The application is using a version of the IATE API that is not compatible with the connected Gateway. Check the version levels and verify correct IATE installation. Reinstall IATE if necessary.

Error -2010: DirectionViolation

Returned By: **IateWrite**

Explanation:

The application program issued **IateWrite** to send a new message, but the API did not send this message, because the API is awaiting the host response for the last message sent.

To avoid this error, the application should follow these rules:

1. After sending each message, normally the application will expect a response from the host, and should receive it through **IateRead**, before sending another message.

The **DirectionViolation** error enforces a “lock” condition to guard against sending another message before receiving a response, because that is usually inappropriate.

2. The application may decide to send a new message even though no response has been received for the last one.

(For example, the application might re-send a message after some time period, if the expected response did not arrive. Also, some interactive applications might allow their users to break the lock and send a new message at any time.)

In such cases, the application can use the **IateControl** command **APIResetLocal** or **APIForwardReset**. This will remove the message-lock condition, allowing one subsequent **IateWrite** to send the next message without the **DirectionViolation** error.

Every **IateWrite** call re-establishes the lock condition -- so that, once again, the API rejects any subsequent **IateWrite** call on this session (with the **DirectionViolation** error) -- until a response arrives, or until the application issues a reset command to break the lock.

(Continued on next page)

See Also: Error -2103, **APIOverrunErr**

The direction-violation error and the API overrun error occur for different reasons. They are not the same. The direction-violation error enforces alternating transmission and reception of messages (the “direction rule”), whereas the overrun error enforces a minimum time period between transmissions (the “throttling rule”).

The application must therefore respect the direction rule (to avoid the **DirectionViolation** error) as well as the throttling time (to avoid the **APIOverrunErr** error).

Error -2011: InterceptError

Returned By: **IateControl:**
APIintrWriteInput
APIintrWriteOutput

Explanation:

The application issued **IateControl**, with the **APIintrWriteInput** or **APIintrWriteOutput** command, to write data on a TA object intercept channel. The specified channel was invalid, or was not in intercept mode.

To avoid this error, the application must use the IATE API intercept functions correctly. First, obtain a session through **IateOpen** with the **APIinterceptName** mode argument. Then use **IateControl** with the **APIintrRouteInput** and/or **APIintrRouteOutput** command, specifying the INTRDIVERT or INTRBOTH mode. For more information, see **Appendix F: Sharing a TA**.

The application must complete those preparations in order to establish Intercept Mode operation on a TA object, before using **APIintrWriteInput** or **APIintrWriteOutput**.

Error -2101: APINoFreeChannel / TooManySessions

Returned By: **IateOpen**

Explanation:

An **IateOpen** call failed because the application already has reached the maximum number of open sessions that the API can support per application.

At this writing, current versions of the API support 253 sessions per application. If the application attempts to open more than 253 sessions, this error will result.

The application cannot open any more sessions until it closes one or more of the sessions that it has already opened.

See Also: **Error -2218, TooManyConnections.**

Error -2102: APIBadChannel / InvalidRefnum

Returned By: **IateClose**
IateControl
IateRead
IateWrite
IateOpen (see notes below)

Explanation:

This error indicates that the application specified an invalid Session Reference Number in a call to an IATE API function.

When the application calls **IateOpen**, the application must store the reference number (a.k.a. “refnum”) that **IateOpen** returns.

IateOpen establishes a TA connection “session” and returns the reference number to uniquely identify that session.

The application uses that reference number in all subsequent IATE API calls for that session.

IateRead, **IateWrite**, **IateClose**, and most **IateControl** calls require the application to specify the reference-number of an existing session. These calls will return this error if the specified reference-number does not match any active session.

IateOpen returns this error only if the caller specified a session reference number in the call. Applications usually do not specify a reference number in a call to **IateOpen**, since the most common usage of **IateOpen** is to obtain a new session and a new reference number. But the application may specify an existing reference number in order to 'reconnect' to an active session. If the specified reference number does not refer to any existing session, then the call returns this error.

Error -2103: APIOverrunErr

Returned By: `IateWrite`

Explanation:

This error indicates that the application, after issuing an **IateWrite** call, issued a second **IateWrite** call too soon, before the “API Throttle” time period had elapsed.

IATE Gateway configuration defines the API throttling time period. This is the minimum time period between the application's successive message transmissions through **IateWrite**, on each session.

After calling **IateWrite**, the application should not call **IateWrite** again on the same session, until the throttling time period has elapsed.

For more information on throttle-interval configuration at the Gateway, see the IATE Gateway documentation for the `API_THROTTLE_INTERVAL` configuration item.

At any time, if the application needs to find out whether or not **IateWrite** is disallowed due to throttling, one way to find out is to call **IateControl** with the **APIGetTaThrottle** command. The return value tells whether or not **IateWrite** is disallowed due to throttling. (See the discussion of **APIGetTaThrottle**, elsewhere in the API documentation.)

If the application wishes to override throttling on a session, it may do so by calling **IateControl** with the **APIInoThrottle** command. This will disable the throttling check, preventing the **APIOverrunErr** error from occurring on the specified session.

(Continued on next page)

See Also: Error -2010, **DirectionViolation**.

The direction-violation error and the API overrun error occur for different reasons. They are not the same. The direction-violation error enforces alternating transmission and reception of messages (the “direction rule”), whereas the overrun error enforces a minimum time period between transmissions (the “throttling rule”).

The application must therefore respect the direction rule (to avoid the **DirectionViolation** error) as well as the throttling time (to avoid the **APIOverrunErr** error).

See Also: Error -2214, **APIOpenBlocked**.

The Open-Blocked error and the API Overrun error are similar in that they both pertain to minimum intervals between certain API calls. But these errors apply to two different API functions, and should not be confused. The **OpenBlocked** error enforces a minimum interval between **IateOpen** calls, whereas **APIOverrunErr** enforces a minimum interval between **IateWrite** calls (on a particular open session).

Error -2201: InternalLogicError

Explanation:

This error indicates an unexpected problem within the API or gateway software. Please contact InnoSys if any IATE API function returns this error, or if this error appears in an API debugging log file.

Error -2205: HostUnreachable

Returned By: IateOpen

Explanation:

This error indicates that the IATE API cannot contact the Gateway host system, which the application specified in the name argument to the **IateOpen** call. Specifically, this error indicates that the specified name is not recognized by the local system's network host name resolution facilities.

To resolve this error:

- Verify that the application specified the gateway host system name with correct spelling, in the name argument to **IateOpen**.
- Verify that the application host's network name resolution facilities can recognize and resolve the specified gateway host name. For example, use the **ping** utility, e.g. "ping hostname", to verify name resolution and to check network connectivity to the specified host.

Depending on system configuration, network host name resolution may involve DNS servers, WINS servers, NIS servers, and/or a local hosts-database file. If necessary, consult your network administrator for assistance in verifying correct resolution of the gateway host name.

See Also: Error -2002, ServerUnreachable (a.k.a. NoServerError).

Error -2207: SessionNotConfigured

Returned By: IateOpen

Explanation:

During an **IateOpen** call, The IATE API requests session configuration information from the Gateway. This error indicates that the API did not receive the requested configuration data.

This error may indicate a version mismatch between the IATE API and Gateway. To resolve this error, verify that the connected IATE Gateway's software version level is current and compatible with the version of the IATE API that the application is using. Contact InnoSys if the error persists.

This error may also indicate network congestion. The API waits for a limited time to receive an expected configuration response message: If the wait time expires with no such response received, the API will return this error.

Error -2208: NoSocket

Returned By: `IateOpen`

Explanation:

During an **IateOpen** call, the IATE API attempts to contact the Gateway. As part of the connection procedure, the API requests the local system to allocate a connection endpoint, termed a “socket”. If the system's network facilities cannot allocate the socket, the API returns this error.

To resolve this error, verify that the system's TCP/IP networking facilities are correctly installed and operational. Also verify that the system's networking and memory resources are not overloaded. Contact your network administrator for assistance if necessary.

Error -2209: CantConnectToServer

Returned By: `IateOpen`

Explanation:

During an **IateOpen** call, the IATE API attempts to contact the Gateway. This error indicates that the API could not contact the Gateway, for one of these possible reasons:

- The API could not bind to a socket,
- The API could not reach the Gateway host system,
- The Gateway was not operational on that system,
- The Gateway was not configured to listen on the TCP port corresponding to the service name that the application specified in the **IateOpen** call, or
- A network-related or system-related problem prevented successful connection.

To resolve this error, verify the application host system's TCP/IP connectivity to the Gateway host system, and verify that the Gateway is operational on that system, and configured to listen on the TCP port corresponding to the service name that the application specified in the **IateOpen** call.

For example, if the application provided this name argument to **IateOpen**:

```
@GatewayHost\ServiceName\TaObjectName
```

then the IATE API attempts to connect to the gateway on the specified GatewayHost, using the TCP/IP port number corresponding to the specified ServiceName. On the Gateway host system, the Gateway must be running and configured to listen on the same port number.

If those requirements are satisfied but the error persists, check for other network-related or system-related problems that might prevent successful connection.

Error -2210: UnexpectedMsgType

Returned By: IateControl
IateRead
IateWrite

Explanation:

This error indicates that the IATE Gateway returned a message code that the IATE API cannot recognize.

This error may indicate a version mismatch between the IATE API and Gateway. To resolve this error, verify that the connected IATE Gateway's software version level is current and compatible with the version of the IATE API that the application is using. Contact InnoSys if the error persists.

Error -2211: WriteFailed

Returned By: **IateOpen**
IateControl
IateWrite

Explanation:

The **WriteFailed** error indicates that the API was unable to transmit information to the Gateway. The API's connection to the Gateway may have been lost. To recover from this error, the application may need to close and reopen the session.

(The **WriteFailed** error only indicates a transmission problem between the API and a Gateway; not between the Gateway and the airline host.)

If the **IateOpen** function returns the **WriteFailed** error, it indicates that the requested session could not be opened, because the API could not transmit a request to the Gateway to open the session. (The error can also be related to other internal messages involved in session startup, depending on the IATE software version level.)

If the **IateWrite** function returns the **WriteFailed** error, it indicates that the API could not transmit a data-message to the Gateway.

If the **IateControl** function returns the **WriteFailed** error, it indicates that the API could not transmit a control-message, or a peer-data message, to the Gateway. Details follow:

IateControl supports a variety of commands, many of which internally involve transmission of control-messages to the Gateway. (These internal control-messages are not visible to the application.) The **WriteFailed** error may occur with any such command, if the API could not send the necessary control-message to the Gateway. This indicates that the **IateControl** command that the application requested could not be completed.

IateControl also supports the **APISendApplMessage** command, which transmits a peer-to-peer data message from the requesting application to a separate application on another system. The **WriteFailed** error indicates that the API could not send the data message to the Gateway.

Error -2212: ReadFailed

Returned By: `IateRead`

Explanation:

The **IateRead** error indicates one of the following problems preventing reception of data from the Gateway:

- A socket read/receive function call failed,
- The read operation returned insufficient data from the Gateway,
- The connection to the Gateway terminated unexpectedly,
- The Gateway timed out the TA and closed the socket, or
- Gateway-to-API communications indicated a message length that exceeds the maximum length the API will accept.

For the first case above, the application can check `APIerrno` (in API version 2 and later), to obtain the error code returned by the failed system call.

Some of these error cases may be caused by network problems, resource problems at the application or Gateway systems, or Gateway software failure. Check the status of the Gateway system, network connectivity, and the application system.

The last case above (a message-length error) should not occur, but if it does occur, it may indicate a version mismatch between the IATE API and the Gateway. Verify that the connected IATE Gateway's software version level is current and compatible with the version of the IATE API that the application is using.

Error -2214: OpenBlocked

Explanation:

This error indicates that the application, after issuing an **IateOpen** call, issued a second **IateOpen** call too soon, before the “Open Delay” time period had elapsed.

The “Open Delay” is the minimum time period between the application's successive calls to **IateOpen**. The default Open Delay time is 70 seconds. After calling **IateOpen**, the application should not call **IateOpen** again until this time period has elapsed.

If the application wishes to change the Open Delay time, it may do so by calling **IateControl** with the **APIsetOpenDelay** command, specifying the desired Open Delay time, in seconds. The application can specify any delay time of no less than 10 seconds. For more information, refer to the section discussing the **IateControl APIsetOpenDelay** command.

See Also: Error -2103, APIOverrunErr.

The Open-Blocked error and the API Overrun error are similar in that they both pertain to minimum intervals between certain API calls. But these errors apply to two different API functions, and should not be confused. The **OpenBlocked** error enforces a minimum interval between **IateOpen** calls, whereas **APIOverrunErr** enforces a minimum interval between **IateWrite** calls (on a particular open session).

Error -2215: **SessionDisconnected**

Returned By: **IateRead**
IateWrite
IateControl

Explanation:

This error indicates that the Gateway has disconnected the session.

After the Gateway disconnects a session, the application's next call to **IateRead**, **IateWrite**, or **IateControl** will return this error. (Internally, the network socket connection between the Gateway and the API closes shortly afterward.)

The session that has been disconnected is the same one on which the application called the API function. The API cannot complete the requested read, write, or control operation, because the session is no longer available.

Note:

If a disconnection occurs while an **IateRead** call is in progress (awaiting data), that call may return zero (indicating no data), without returning the **SessionDisconnected** error.

If the application subsequently calls **IateRead**, **IateWrite**, or **IateControl**, on the same session, that subsequent call will return the **SessionDisconnected** error.

Error -2216: NotImplemented

Returned By: `IateControl`

Explanation:

The application issued an **IateControl** call containing a command code that is either invalid or not implemented by the running version of the API or Gateway.

The possible causes of this error are:

1. Version mismatch.
2. Incorrect IATE control code definitions.

To resolve this error, ensure the application is coded and tested to work with the installed version of the IATE API, and that the connected IATE Gateway is version-compatible with that release of API. Also ensure that the application was compiled using the correct set of header files supplied with that API release.

Error -2217: TooMuchDataQueued

Returned By: **IateControl:**
APIGetHostStat
APIGetTaStat

This error indicates that too much data is queued in the API. The API cannot complete the requested **APIGetHostStat** or **APIGetTaStat** control operation.

This can happen if the application issues too many **APIGetTaStat** or **APIGetHostStat** commands, without intervening **IateReads**, while the API's internal buffers fill with data incoming from the Gateway.

To resolve this error, the application must call **IateRead** in order to free some of the received data that the API is holding in its internal buffers.

A properly designed application avoids the **TooMuchDataQueued** error.

Background:

To check for received data on an open session, some applications simply post a blocking **IateRead** call to wait for data.

This can work for a single-threaded application using a single session, or for a multithreaded application using multiple sessions.

If the design requires that the application not block in **IateRead**, or if a single-threaded application uses multiple sessions, the application may use **APIGetTaStat** or **APIGetHostStat** to poll for data. When **APIGetTaStat** or **APIGetHostStat** indicates that incoming data has arrived on the session, the application should promptly call **IateRead** to retrieve the data.

By retrieving the received data from the API promptly, the application prevents overflow of the API's internal buffers. This prevents the **TooMuchDataQueued** error.

(It is important to satisfy this requirement on all open sessions. Even if the application retrieves data efficiently from a particular session, API buffers can still overflow if the application fails to retrieve data equally efficiently from other sessions. The well-designed application maintains efficient data flow on all of its open sessions.)

Error -2218: TooManyConnections

Explanation:

An API/Gateway connection attempt has failed because all the available number of TCP/IP connections with the Gateway have already been used.

(This differs from **APINoFreeChannel / TooManySessions** described below. **TooManyConnections** reflects a limit on the number of socket connections which can be opened, as opposed to the number of sessions with objects that may be established.)

The operating system defines a limitation on the number of files an application can have opened during a given process. This limit has been exceeded.

Error -2404: InvalidTask

Explanation:

This error code is obsolete. It was used in previous API releases for Windows 3.1x.

Appendix B: Background Information on the Gateway

This appendix describes the relationship between a terminal or printer object's network address and its “object name” or “group name”.

Terminal and Printer Device Objects

Gateway configuration associates an individual terminal or printer address with an “object name”. The address takes the form of an IA/TA or LNIA/TA pair. (IA = Interchange Address, LNIA = Line/IA, TA = Terminal Address.)

The **IateOpen** API function call, given the **APILinkToName** option, requests a connection to a terminal or printer object specified by its unique "object name".

A collection of object names can be assigned a common "group name". The **IateOpen** API function call, given the **APILinkToName** option, can specify a “group name” to select any one of a named group of objects.

The **APILinkToTa** option requests the link using the IA and TA address of an object, rather than the object name. The **APILinkToTa** parameters can also include a port name, to specify a particular physical line if necessary.

Note:

APILinkToTa is supported only on TAs defined on ALC host connections, and on some, but not all X.25 connections. It is strongly recommended that the application use **APILinkToName**.

If an application connects to objects by Group names (rather than the individual object names), the application may need to discover the individual name of an object after connecting to it. For that purpose, use **APIGetObjectConfig**, which returns information that includes the name of the connected object.

The connection request may succeed if the object name is available, not already in use by another application. A connection request that specifies a group name will succeed if there is any object available in the specified group.

In addition to an object name, an IA TA, and a group name, each device address configured at the Gateway has a type. The types are **TERMINAL**, **PRINTER**, **TERMINAL_API** and **PRINTER_API**.

Dynamic Objects

Objects configured at the Gateway with type `TERMINAL_API` or `PRINTER_API` are called “dynamic objects”.

- To connect to a `TERMINAL_API` dynamic object, the application uses **IateOpen** with the **APILinkToDyCrt** option.
- To connect to a `PRINTER_API` dynamic object, the application uses **IateOpen** with the **APILinkToDyPrt** option.

The application does not specify the name or address of a particular dynamic object. Instead, the Gateway selects an available object of the specified type.

The **APILinkToDyCrt** or **APILinkToDyPrt** parameters can also include a port name, to specify a particular physical line if necessary.

Appendix C: Description of Host Traffic

This appendix contains a brief description of the format of ALC host messages. Most of this information applies specifically to ALC, not X.25 or TCP connections. However, the information about the message control characters (“C1” and “C2”) applies generally to all types of airline hosts.

The IATE API uses ASCII character codes. IATE user application programs use ASCII character codes in message communications through the API.

The IATE Gateway uses ALC character code set in communications with airline hosts that require it. The Gateway uses ASCII character codes in communications with the API. Therefore, when delivering messages from the API to the host, the Gateway translates them from ASCII to ALC; and, when delivering messages from the host to the API, the Gateway translates them from ALC to ASCII.

```
Application <-----> API <-----> Gateway <-----> Airline Host
           ASCII           ASCII           ALC
                                   (depending on host type)
```

A message from the airline host may consist of one or more message-segments. The airline host sends each ALC message segment to the Gateway in this format or a similar format:

<u>Addressing</u>	<u>Data</u>	<u>End-of-Message</u>	<u>Checksum</u>
IA TA	C1 C2 Message-Text	EOMc EOMi EOMu EOMpb	CCC

The IA (Interchange Address) and TA (Terminal Address) determine the station address.

The C1 and C2 characters (a.k.a. “Command 1” and “Command 2”) often specify positioning information, indicating where the text of the message should be displayed on a terminal. However, the exact meaning of C1 and C2 depend on the host type, the type of device (terminal or printer) configured on a TA, and the message type. For details, refer to the host system’s documentation.

There are four valid EOM character values: **EOMi**, **EOMc**, **EOMu**, and **EOMpb**. Each segment must contain one of these EOM characters (preceding the Checksum at the end). The final segment of a message typically uses the **EOMc** character.

The EOM character(s) contained in the final segment of a message are called final EOMs. The EOM character(s) contained in a multi-segment message's first segment, or in any intermediate segment, are called intermediate EOMs. Often EOMc and EOMu are used as final EOMs, while EOMi and EOMpb are used as intermediate EOMs; however, this differs on some systems.

The Checksum value (also called “**CCC**”) provides a confirmation code whereby the software receiving a message can validate it mathematically, to find out if the message data was corrupted during transmission. The user application need not be concerned with CCC code calculations. When the application sends a message, the IATE software generates checksums as needed. When the application receives a message, the **IateRead** function provides a CCC validation result flag (a simple Boolean value) which the application can test.

When an application uses **IateRead** API function to receive message data, the returned message-buffer contains the text of the message. The API removes the IA and TA values before returning the data to the application. **IateRead** also returns another buffer, called the “control” buffer, which contains the C1, C2, and EOM characters, and the CCC validation indicator.

Appendix D: Sharing a TA

This appendix describes the Shared TA mechanism of the IATE Gateway.

An application using the IATE API connects to the airline host through a TA Object configured at the Gateway. Message traffic passes through the Gateway on its way from the airline host to the application, or vice versa.

A second application can ask the Gateway to “intercept” or “divert” that message traffic. This is the Gateway's “Shared TA” mechanism. The Shared TA mechanism's two modes have the following characteristics:

“Intercept” Mode

- In the “intercept” mode, the first application retains its connection to the host, and the second application shares it:
- When the first application sends a message to the host, the Gateway sends a copy of the message to the second application. Both the host and the second application receive the message.
- When the host sends a message to the first application, the Gateway sends a copy of the message to the second application. Both the first and second applications receive the message.

“Divert” Mode

- In the “divert” mode, the Gateway diverts the messages that would normally pass between the first application and the host, re-routing them to/from the second application instead.
- When the first application sends a message to the host, the Gateway re-routes the message to the second application. The host does not receive it.
- When the host sends a message to the first application, the Gateway re-routes the message to the second application. The first application does not receive it.

The second application can also send messages to the first application or the host. The Gateway delivers such messages as if they had passed between the first application and the host. For details, see the **Message Forwarding** section below.

Usage

As explained above, the Shared TA mechanism involves two applications. The first application has opened a TA object by the normal means. The second application accesses it through the Shared TA mechanism, by using the following procedure:

1. The second application connects to the object using **IateOpen** with the **APIinterceptName** option:

```
refnum = IateOpen (StartCode, APIinterceptName, ObjectName);
```

2. The second application uses **IateControl** to select the “intercept” or “divert” mode, and the direction of message traffic to intercept or divert:

```
IateControl (StartCode, ShareCommand, ShareFlag);
```

The **ShareCommand** and **ShareFlag** arguments take the following values:

ShareCommand:

This argument selects the direction of message traffic that the second application will intercept or divert.

Use either one of the following values:

APIintrRouteOutput - for application-to-host message traffic, or
APIintrRouteInput - for host-to-application message traffic.

ShareFlag:

This selects the “intercept” or “divert” mode (discussed above).

Use either one of the following values:

INTRBOTH - to select the “intercept” mode, or
INTRDIVERT - to select the “divert” mode.

Examples:

To “intercept” messages that the first application sends to the host, use this `IateControl` call:

```
IateControl (StartCode, APIintrRouteOutput, INTRBOTH);
```

To “divert” messages that the first application attempts to send to the host, use this `IateControl` call:

```
IateControl (StartCode, APIintrRouteOutput, INTRDIVERT);
```

To “intercept” messages that the host sends to the first application, use this `IateControl` call:

```
IateControl (StartCode, APIintrRouteInput, INTRBOTH);
```

To “divert” messages that the host attempts to send to the first application, use this `IateControl` call:

```
IateControl (StartCode, APIintrRouteInput, INTRDIVERT);
```

3. The second application proceeds to receive and/or send messages on the intercepted channel, using **IateRead** and/or **IateWrite**.
4. To terminate the Shared TA mode, the second application again uses `IateControl`, with the `INTRNORMAL` flag:

```
IateControl (StartCode, APIintrRouteOutput, INTRNORMAL);
```

or

```
IateControl (StartCode, APIintrRouteInput, INTRNORMAL);
```


Message Forwarding

In addition to intercepting or diverting host data traffic, the second application can also send messages to the first application or the host. The Gateway delivers such messages as if they had passed between the first application and the host.

Message Forwarding requires “intercept” mode (INTRBOTH), not “divert” mode (INTRDIVERT). If the second application wishes to send a message to be forwarded on a connection that is currently in the “divert” mode, the application should switch to “intercept” mode before sending the message.

Note for Macintosh Applications:

The IATE API for Macintosh automatically resets to the normal mode (INTRNORMAL) after each message transaction. Macintosh API applications that use message forwarding must reset to “intercept” mode (INTRBOTH) before sending each message to be forwarded.

Examples of Message Forwarding:

The second application sends a message which the gateway will forward to the host (as if it came from the first application):

```
char buff[] = "SOME DATA TO THE HOST";
IateControl (refnum, APIintrWriteInput, buff);
```

The second application sends a message which the gateway will forward to the first application (as if it came from the host):

```
char buff[] = "SOME DATA TO THE SHARED TA";
IateControl (refnum, APIintrWriteOutput, buff);
```

Sample Program

The IATE software package includes an example of a sharing application, **testincp.c**. This application should be used in tandem with the sample terminal test application, **testterm.c**, as follows:

- Using **testterm**, open a connection to an available TA object.
- Run **testincp** using the same object name, selecting the the **divert** mode, as follows:

```
testterm -oan_object_name
testincp -oan_object_name -sINTRDIVERT
```

The Gateway will divert messages that would normally pass between the **testterm** application and the host. The **testincp** program will receive the diverted messages.

- Enter a message into **testterm**, and watch **testincp** receive the message. Also have the host send a message, and watch **testincp** receive that message as well.
- Stop **testincp**, and then restart it in the “intercept” mode, as follows:

```
testincp -oan_object_name -sINTRBOTH
```

- Messages can now be forwarded to the host by placing an “I” in front of **testincp** keyboard entries, or to the **testterm** application by placing an “O” in front of the keyboard entry.

Refer to the **testincp.c** source code file for further information about the sharing sample program.

To display a summary of usage information for either program, use the **-h** command-line option:

```
testincp -h
```

```
testterm -h
```

Appendix E: The IATE API for Visual Basic

This appendix describes the IATE API for Microsoft® Visual Basic.

The IATE API DLLs for Visual Basic

The IATE API for Visual Basic is named “**iate32b.dll**”. This DLL is for use with Visual Basic only. It is separate and independent from the C-language version of the DLL, “**iate32.dll**”.

Because the API for Visual Basic is provided through the traditional DLL mechanism (not a COM or .NET object), its usage requires explicit declarations of the API functions and their parameters. The Sample Programs discussed below include all of the required definitions. InnoSys recommends using the Sample Programs as a starting point for developing applications with the IATE API in Visual Basic.

Sample Programs for Visual Basic

The IATE Sample Programs for Visual Basic have been tested with Visual Basic version 6. At this time (2001), the Sample Programs have not been tested with other versions of Visual Basic, and are therefore not intended for use with any earlier version, or any later version such as Visual Basic 7 or the .NET framework.

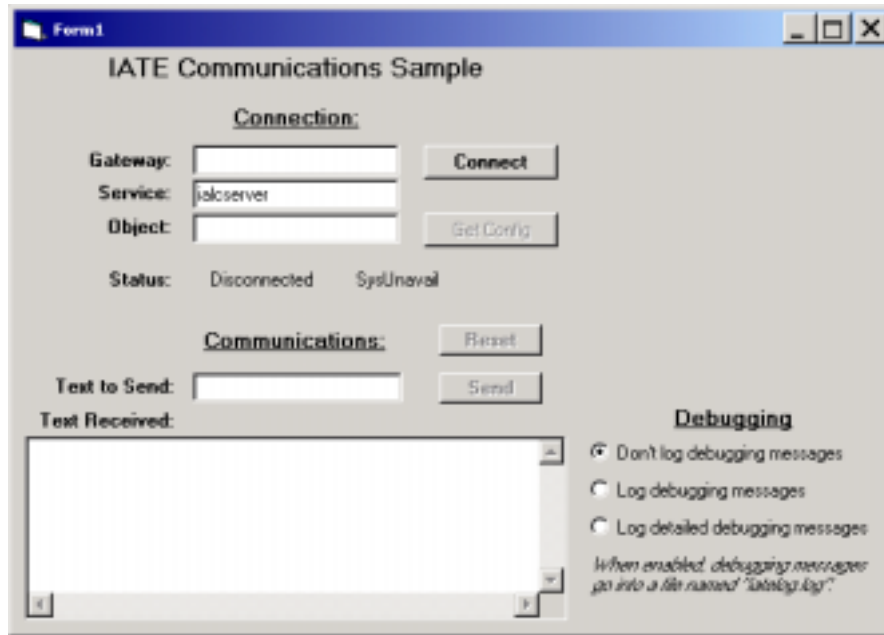
There are two different IATE Sample Programs for Visual Basic. The first is a simple ‘terminal’ application with which a user can connect to an airline host, enter commands, and view responses.

The second sample program is similar to the first, with additional features to demonstrate the IATE Intercept Mode. (Intercept Mode is discussed in **Appendix D: Sharing a TA**, on page 120). Aside from this difference in communication modes, the two sample programs are similar in purpose and structure.

Each sample program for Visual Basic uses a “Form” to present its user-interface. (A Form is the standard Visual Basic object to create a user-interface.) The sample programs’ Forms are illustrated below.

The Sample Programs' Forms

This is the Form (user interface) for the first sample program:



In the first three text fields on the form, the user enters the standard parameters required for an IATE connection: (1) the name of the IATE Gateway host, (2) a TCP/IP network service-name or port-number (such as “ialcserver” or 1413), and (3) a client/TA object name.

Next, the user presses the **Connect** button on the form, to connect to the gateway. When the user presses the **Connect** button, the sample program calls the **IateOpen** function to open the connection.

After connecting, the user can enter a command into the **Text to Send** field, and press the **Send** button to send the command to the airline host. The program uses **IateWrite** to send messages to the host through the Gateway, and **IateRead** to receive responses. Any responses received from the host will appear in the large text box in the bottom left corner of the form.

Additional controls on the form include: a write-lock Reset button, a configuration information retrieval button, and diagnostic logging control buttons. All of these controls perform their operations through the appropriate IATE API functions and supplementary Visual Basic code in the sample programs.

The form's code module contains the subroutines which operate all of those interactive objects (text fields and buttons) on the form. The form's code file is “**IATE_sample.frm**”.

The second sample program, which demonstrates the IATE Intercept Mode, uses a slightly different Form:

The screenshot shows a Windows-style window titled "Form1" with a blue title bar. The main content area is titled "IATE Sample for Intercept Mode". It is organized into three distinct sections. The "Connection:" section at the top contains three text input fields labeled "Gateway:", "Service:", and "Object:". The "Service:" field contains the text "ialcserver". To the right of these fields are two buttons: "Connect" and "Get Config". Below these fields, the status is displayed as "Status: Disconnected SysUnaval". The "Communications:" section in the middle features a "Text to Send:" text box, a "Text Received:" label, and a large scrollable text area. To the right of the text boxes are three buttons: "Reset", "Send to Host", and "Send to Client". The "Debugging" section on the right side contains three radio button options: "Don't log debugging messages", "Log debugging messages", and "Log detailed debugging messages". Below these options is a small italicized note: "When enabled, debugging messages go into a file named 'iatelog.log'".

This Form is nearly identical to the first sample's Form, except that the **Send** button is replaced with two separate buttons: **Send to Host** and **Send to Client**. The meaning of the **Connect** button is also different from the first sample.

In the first three text fields on the form, the user enters the standard parameters required for an IATE connection: (1) the name of the IATE Gateway host, (2) a TCP/IP network service-name or port-number (such as "ialcserver" or 1413), and (3) a client/TA object name.

Next, the user presses the **Connect** button on the form. When the user presses the **Connect** button, the sample program calls the **IateOpen** function to open the connection. To establish the Intercept Mode, the program calls the **IateControl** function with the **APlintrRouteOutput** and **APlintrRouteInput** commands, and specifies the **INTRBOTH** flag for Intercept Mode.

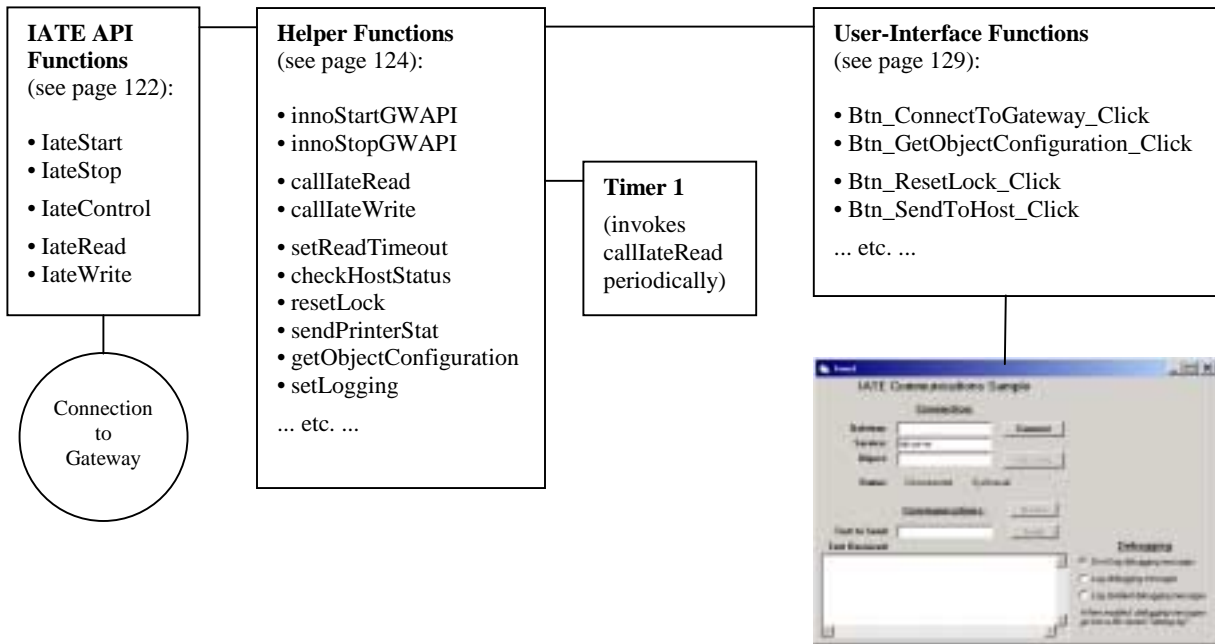
As in the first sample, the user can enter a command into the **Text to Send** field, and press the **Send** button to send the command to the airline host. In the Intercept Mode, the program sends the command "on behalf" of the intercepted client (as if that client had sent the command).

Any responses received from the host will appear in the large text box in the bottom left corner of the form. Because this program uses Intercept Mode, the host's responses will also reach the intercepted client (i.e., the response was "intercepted").

Intercept Mode is discussed further in **Appendix D: Sharing a TA**, on page 120.

Structure of the Sample Applications

The following diagram illustrates the general structure of the sample code. The functions listed in the diagram are described on the following pages.



Using the IATE API in Visual Basic

The IATE API for Visual Basic supports the same functions as the IATE API for the C language: **IateStart**, **IateOpen**, **IateRead**, **IateWrite**, and **IateControl**. The Sample Programs include the required declarations of those functions, along with definitions of some of their possible parameters (such as the various **IateControl** command codes). These declarations can be found in the file “**IATE_API_defs.bas**”.

To simplify development, it is recommended that the application use an additional set of “Helper Functions” in order to access the API functions mentioned above. The Helper Functions can encapsulate the typical usage of the API functions in the Visual Basic environment. This can be helpful during development and may also reduce code redundancy. The Sample Programs provide suggested Helper Functions in the file “**IATE_API_helper.bas**”.

Data Types

The IATE API for Visual Basic uses the following data types for function call parameters.

The following data types apply in parameters to the fundamental API functions (**IateStart**, **IateOpen**, **IateRead**, **IateWrite**, **IateClose**, and **IateStop**):

As Long - IATE API numeric parameters are generally declared **ByVal As Long**, corresponding to the **long** parameters in the API for C language.

As String - String parameters to several API functions are declared **ByVal As String**.

For example, the **IateOpen** function declaration has two Long parameters and a String parameter:

```
Declare Function IateOpen Lib "IATE32b.DLL" ( _  
    ByVal lStartCode As Long, _  
    ByVal lCmd As Long, _  
    ByVal sBuff As String _  
) As Long
```

As Any - The various **IateControl** commands use different data types in the third parameter to **IateControl**. These data types include: Long, String, or a data-structure type. Depending on the command-code, these may be either input to the function, or returned output. To accommodate all of these cases, the **IateControl** function’s third parameter is declared **ByRef As Any**.

Some additional data types (e.g., **As Integer** and **As Byte**) are used in parameters to the IATE API Helper Functions for Visual Basic, as described later in this appendix.

IATE API Functions in Visual Basic

The IATE API functions are declared in “**IATE_API_defs.bas**”. Also declared in the same file are some common parameter values, including the various command codes for **IateControl**.

Each API function’s parameters correspond directly to those of the C-language version of the function. For more information, please refer to the discussion of the C-language version of each function, in the **API Library Reference** section of this manual.

It is recommended that the application use the Helper Functions provided with the Sample Programs, instead of calling these API functions directly. Helper Functions are described in the next section.

```
' Function: IateStart()
' Purpose: The first API call required to begin using the API.
'
Declare Function IateStart Lib "IATE32b.DLL" ( _
    ByVal lInstallHandlers As Long, _
    ByVal lDummy As Long, _
    ByVal sBuff As String _
) As Long
```

```
' Function: IateOpen()
' Purpose: Open a connection to the IATE Gateway via the API.
'
Declare Function IateOpen Lib "IATE32b.DLL" ( _
    ByVal lStartCode As Long, _
    ByVal lCmd As Long, _
    ByVal sBuff As String _
) As Long
```

```
' Function: IateRead()
' Purpose: Read data from the host through the IATE Gateway.
'
Declare Function IateRead Lib "IATE32b.DLL" ( _
    ByVal lRefNum As Long, _
    ByVal MAX_BUFF_SIZE As Long, _
    ByVal sBuff As String, _
    ByVal innoCtrlBlock As String _
) As Long
```

```
' Function: IateWrite()
' Purpose: Write data to the host through the IATE Gateway.
'
Declare Function IateWrite Lib "IATE32b.DLL" ( _
    ByVal lRefNum As Long, _
    ByVal lCommandLength As Long, _
    ByVal sCommand As String _
) As Long
```

```
' Function: IateClose()
' Purpose: Close the connection with the IATE Gateway via the API.
'
Declare Function IateClose Lib "IATE32b.DLL" ( _
    ByVal lRefNum As Long _
) As Long
```

```
' Function: IateStop()
' Purpose: Terminate this program's usage of the IATE API.
'
Declare Function IateStop Lib "IATE32b.DLL" ( _
    ByVal sStartCode As Long _
) As Long
```

```
' Function: IateControl()
' Purpose: This function supports various API control commands.
'
Declare Function IateControl Lib "IATE32b.DLL" ( _
    ByVal lRefNum As Long, _
    ByVal lCmd As Long, _
    ByRef sBuff As Any _
) As Long
```

“Helper Functions” in the Sample Applications for Visual Basic

The IATE API Sample Programs for Visual Basic contain several Helper Functions. The Helper Function module code is provided in the file “**IATE_API_helper.bas**”.

The Helper Functions module operates as a “layer” between the IATE API functions and the rest of the application code. An application can use the Helper Functions to encapsulate typical usage of IATE API functions in Visual Basic.

Note: There are two different Sample Programs. The Helper Functions differ slightly between the main Sample Program, and the alternative sample for Intercept Mode. The following listing refers to the main sample, not the Intercept Mode sample.

```
' Procedure: innoStartGWAPI
'
' Purpose:      Open a connection to the IATE API, and then
'              open a connection to a TA object through the API and Gateway.
'
' Arguments:   GatewayName:      Name of IATE Gateway host.
'
'              ServiceName:      Name of TCP/IP Service
'                                (such as "ialcserver"),
'                                or TCP/IP port number.
'
'              ObjectName:       Name of TA object.
'
' Returns:     'True' to indicate successful connection, or
'              'False' to indicate failure to connect.
'
Public Function innoStartGWAPI( _
    ByVal gatewayName As String, _
    ByVal serviceName As String, _
    ByVal objectName As String _
) As Boolean
```

' Procedure: **innoStopGWAPI**
'
' Purpose: Close a connection with an IATE Gateway TA object,
' and then close our connection with the IATE API.
'
' Returns: 'True' if this function closed the connection, or
' 'False' if the connection was already closed.
'
Public Function **innoStopGWAPI()** As Boolean

' Function: **callIateRead**
'
' Purpose: Call the IateRead API function to retrieve
' any data that the API has received through the
' Gateway connection.
'
' Returns: The data received, as a string
' (or the empty string "" if no data
' has been received on this call).
'
' Usage: Call this frequently from a Timer event handler.
'
' Note: If IateRead were to sit and wait for data,
' the VB GUI would 'freeze' during the wait
' (in the current single-threaded implementation).
' To prevent IateRead from waiting, an earlier call
' to setReadTimeout has set the IATE read timeout
' to zero. IateRead will therefore return immediately
' (returning data, if any, that the IATE API has
' recently received from the Gateway).
'
' Since IateRead will return immediately, we'll have to
' call this function frequently to check for any
' additional data received. For that reason, a timer
' event handler will call this function periodically.
'
Public Function **callIateRead()** As String

' Function: **callIateWrite**
'
' Purpose: Call the IateWrite function to send a data message
' through the the IATE API and Gateway to the airline host.
'
' Argument: msg: The data message to send.
'
' Returns: The return code from the IateWrite API function.
' A value less than zero indicates an error.
'
Public Function **callIateWrite**(ByVal msg As String) As Integer

```
' Procedure: setReadTimeout
'
' Purpose:   Set a zero timeout for IateRead calls, so that
'           IateRead will not block while waiting for data.
'           To set that timeout, use the IateControl
'           API function with command APISetTO.
'
' Usage:    Call this once after opening a session.
'
' Note:     This will set a zero timeout, so that IateRead
'           will return immediately, not wait for data.
'           This helps to keep the VB GUI running smoothly, but
'           we shall have to call the callIateRead() procedure
'           repeatedly and frequently to check and retrieve any
'           data received. A Timer event should trigger those
'           periodic calls to callIateRead().
'
' See also: Timer1_Timer
'
Public Sub setReadTimeout()
```

```
' Procedure: checkHostStatus
'
' Purpose:   Check the status of the gateway/host connection,
'           using the IateControl API function with the
'           APIGetHostStat command.
'
' Returns:   'True' if the gateway/host connection
'           appears to be up and operational; or
'           'False' if there seems to be a possible
'           problem with that connection.
'
' Usage:    Call this frequently from a Timer event handler.
'
Public Function checkHostStatus()
```

```
' Procedure: resetLock
'
' Purpose:   Use the APIResetLocal command, asking the IATE API
'           to reset the Keyboard-Locked status on the current session.
'
Public Sub resetLock()
```

```
' Procedure: sendPrinterStat
'
' Purpose:   Send a Printer-Stat message to the Gateway.
'           This is a required formality, even though we have no
'           printer attached to this program. Because there is no
'           printer attached, this Printer-Stat message will
'           indicate that the printer is 'unavailable'.
'
' Usage:    Call this once after opening a session.
'
Public Sub sendPrinterStat()
```

```
' Procedure: getObjectConfiguration
'
' Purpose:   Retrieve information about the configuration
'           of the currently connected TA object.
'           To get that information, use the IateControl
'           API function with command APIGetObjectConfig.
'
' Returns:  A string containing a humanly readable (we hope)
'           representation of the object's configuration.
'
Public Function getObjectConfiguration()
```

```
' Procedure: setLogging
'
' Purpose:   Send a Set-Logging message to the Gateway.
'           This enables API debugging messages,
'           logged in a file named "iatelog.log".
'
' Usage:    Call this once after opening a session.
'
' Argument: level:      0 to turn off debugging, otherwise
'                   a level up to FFFF hexadecimal (65535 decimal)
'
' Note:     IATE API debugging facilities are limited in VB.
'           We can use APISetApiLogging (to log the debugging
'           messages in a file), but we cannot use APISetApiDebugging
'           (e.g. to output the messages to the screen).
'
Public Sub setLogging(level As Long)
```

' Procedure: **protocolName**
'
' Purpose: Given an IATE host protocol type number,
' return the name of the indicated host protocol.
'
' Called by: getObjectConfiguration()
'
Public Function **protocolName**(protocolNumber As Integer)

' Procedure: **objectTypeName**
'
' Purpose: Given an IATE object type number,
' return the name of the indicated object type.
'
' Called by: getObjectConfiguration()
'
Public Function **objectTypeName**(objectTypeNumber As Integer)

' Procedure: **gatewayTypeName**
'
' Purpose: Given an IATE gateway type number,
' return the name of the indicated gateway type.
'
' Called by: getObjectConfiguration()
'
Public Function **gatewayTypeName**(gatewayTypeNumber As Integer)

' Procedure: **lineTypeName**
'
' Purpose: Given an IATE host line type number,
' return the name of the indicated host line connection protocol.
'
' Called by: getObjectConfiguration()
'
Public Function **lineTypeName**(lineTypeNumber As Byte)

' Procedure: **innoShowError**
'
' Purpose: Set up an error message to report an IATE API error,
' specified by an error code returned from an IATE API function.
'
' Argument: innoErrNum: The error code returned from an IATE API function.
'
' Note: The error code numbers used here are defined in the
' IATE API C-language header file "u_apierr.h".
'
Public Function **innoShowError**(ByVal innoErrNum As Long) As String

User-Interface Functions in the Sample Applications for Visual Basic

Each sample program for Visual Basic uses a “Form” to present its user-interface, as discussed earlier (see “**The Sample Programs’ Forms**”, on page 126).

Following is an overview of the Visual Basic code attached to the form. This code performs the functions assigned to each text field and button object on the form, and also uses the Helper functions to communicate with the Gateway and host.

Note: There are two different Sample Programs. The Helper Functions differ slightly between the main Sample Program, and the alternative sample for Intercept Mode. The following listing refers to the main sample, not the Intercept Mode sample.

```
' Name:      Btn_ConnectToGateway_Click
'
' Purpose:   Mouse-click handler for the Connect/Disconnect button.
'           This subroutine calls other subroutines to Connect or Disconnect
'           from the airline host. This subroutine also updates the display
'           on the main form, to show the Connected or Disconnected status.
'
Public Sub Btn_ConnectToGateway_Click()
```

```
' Procedure: Btn_GetObjectConfiguration_Click()
'
' Purpose:   Mouse-click handler for the Get-Configuration button.
'           This subroutine calls getObjectConfiguration to obtain
'           information about the connected TA object's configuration,
'
Private Sub Btn_GetObjectConfiguration_Click()
```

```
' Procedure: Btn_ResetLock_Click()
'
' Purpose:   Mouse-click handler for the Reset-Lock button.
'           This subroutine calls resetLock to reset the
'           IATE API write-lock; and then turns off the
'           Keyboard-Locked indicator, and disables the
'           Reset-Lock button.
'
Private Sub Btn_ResetLock_Click()
```

```
' Name:      Btn_SendToHost_Click
'
' Purpose:   Mouse-click handler for the Send-Message button.
'
'           This subroutine uses callIateWrite to send a
'           message through the Gateway to the host.
'           The text of the message comes from the
'           "Text to Send" input text box.
'
'           This function also turns off the
'           Keyboard-Locked indicator, and
'           enables the Reset-Lock button.
'
Private Sub Btn_SendToHost_Click()
```

```
' Name:      Radio_Debugging0_Click
'
' Purpose:   Mouse-click handlers for the first of the
'           three API Debugging mode buttons:
'           This is the "Don't log debugging messages" button.
'
Private Sub Radio_Debugging0_Click()
```

```
' Name:      Radio_Debugging1_Click
'
' Purpose:   Mouse-click handler for the 2nd of the three
'           API Debugging mode buttons. This button, labelled
'           "Log debugging messages", enables a commonly used set of
'           API diagnostic messages for troubleshooting purposes.
'
Private Sub Radio_Debugging1_Click()
```

```
' Name:      Radio_Debugging2_Click
'
' Purpose:   Mouse-click handler for the third of the
'           API Debugging mode buttons. This button, labelled
'           "Log detailed debugging messages", enables all possible
'           API diagnostic messages for troubleshooting purposes.
'
Private Sub Radio_Debugging2_Click()
```

The “Timer Object” in the Sample Applications for Visual Basic

The sample code for Visual Basic uses a “Timer” object to check for messages from the host. The timer object periodically invokes its event handler subroutine, which uses **callIateRead** to retrieve any messages that the API has received from the Gateway and the airline host.

The reason for using a timer in this way is that the sample programs are single-threaded. When **callIateRead** calls **IateRead**, the rest of the sample program and the VB user interface are suspended, until after **IateRead** returns. If the **IateRead** call were to wait for data before returning, the entire program would appear to ‘freeze’.

To prevent **IateRead** from waiting, the sample code calls the **setReadTimeout** helper function during program startup. After **setReadTimeout** has set the read timeout to zero, the timer function can use **callIateRead** with minimal delay, so that the program will not ‘freeze’.

I

```
' Procedure:  Timer1_Timer
'
' Purpose:   Perform frequent periodic tasks:
'           - Retrieve any new data received
'             from the Host through the Gateway, and
'           - Check the status of the Gateway/Host connection.
'
' Usage:    A timer should trigger this procedure periodically,
'           on an interval of one second or less.
'           This allows us to call IateRead to check for data
'           almost continuously, without incurring any substantial
'           delay during the IateRead call. This rapid 'polling'
'           technique is necessary in our current single-threaded
'           implementation. We must avoid delays in IateRead,
'           to keep the VB GUI running smoothly.
'
'           A multi-threaded implementation could be more efficiently
'           driven by other "events" rather than a continuously polling
'           timer routine. But multi-threading would be more complex.
'           One way to do that would involve COM/ActiveX facilities
'           to achieve multi-threading. Or perhaps new threading features
'           in future versions of VB could be used. For basic IATE
'           communication purposes, it's adequate (and far simpler)
'           to use timer-driven polling in a single thread.
'
' See also:  setReadTimeout
'
Private Sub Timer1_Timer()
```
