HP TP Desktop Connector for ACMS

Client Application Programming Guide

January 2006

This manual describes how to write, develop, and debug desktop client programs that access HP ACMS applications from the desktop, using the *HP TP Desktop Connector* for ACMS client services.

Contents

3 Developing HP ACMS Applications

Part II Portable API Client Development

4 Developing Portable API Client Programs

5 Using Portable API Extensions for Microsoft Windows

6 Using Portable API Extensions for OSF/Motif

7 Forced Nonblocking Extension to the Portable API

A Sample Application Code

B Tools

Index

Examples

Figures

Tables

Preface

This guide describes how to use the *HP TP Desktop Connector* for ACMS software to:

- Incorporate TP Desktop Connector client services in a customer-written desktop client program based on sample code supplied in the kit. The desktop client program signs the user in to the HP ACMS system, selects ACMS tasks, and signs the user out of the ACMS system.

- Design, code, build, and debug a desktop client program that uses forms processing products and interacts with the end user on the desktop system and a HP ACMS system.

- Manage the desktop system running the desktop client program and the system running the TP Desktop Connector Gateway for ACMS software.

Intended Audience

This guide is intended for readers with diverse backgrounds:

For ACMS developers:

- Part I is for users who are knowledgeable about desktop presentation tools and ACMS programming.

- Part II describes development procedures for TP Desktop Connector applications.

Manual Structure

This manual has the following structure:

Part I	Components and Design
Chapter 1	Describes the components of the software and the basic processing that is performed.
Chapter 2	Explains the design essentials.
Chapter 3	Describes the development to be done on the OpenVMS system.

Part II	Portable API Client Development
Chapter 4	Describes the steps to code, debug, and build a desktop client program that uses TP Desktop Connector client services in a blocking environment.
Chapter 5	Describes the steps to code, debug, and build a desktop client program that uses TP Desktop Connector client services in a Windows event-driven nonblocking environment.
Chapter 6	Describes the stages to code, debug, and build a desktop client program that uses TP Desktop Connector client services in a Motif, event-driven, nonblocking environment on OpenVMS.
Chapter 7	Discusses guidelines for building TP Desktop Connector client programs using the forced nonblocking client services.

Appendixes	
Appendix A	Lists the locations of code that accompanies the TP Desktop Connector software.
Appendix B	Lists the general purpose tools, include files, and sample programs in the ACMSDI$EXAMPLES directory.

Related Documents

For information on developing ACMS applications with *HP TP Desktop Connector* for ACMS, refer to the following manuals:

- **HP TP Desktop Connector for ACMS** *Getting Started*

  Introduces you to the TP Desktop Connector software and provides information for building client applications that call ACMS tasks.

- **HP TP Desktop Connector for ACMS** *Client Services Reference Manual*

  Provides a list of client services with their format, parameters, and qualifiers.

- **HP TP Desktop Connector for ACMS** *Gateway Management for ACMS*

Describes system management tasks for the gateway.

- **HP ACMS for OpenVMS** *Getting Started*

  Provides a high-level discussion and examples of the activities to develop, install, and run a complete application.

If you are new to programming with ACMS software, HP recommends reading the following books before using the **HP TP Desktop Connector for ACMS** product:

- **HP ACMS for OpenVMS** *Writing Applications*

  Describes procedures to follow using the Application Development Utility (ADU).

- **HP ACMS for OpenVMS** *Writing Server Procedures*

  Describes how to write and debug procedures for ACMS applications. Also supplies reference information for application and system programming services.

For additional information on ACMS software and on installing the HP ACMS product, refer to the following manuals:

- **HP ACMS for OpenVMS** *Introduction*

  Describes basic concepts and terms concerning the ACMS environment.

- **HP ACMS for OpenVMS** *ADU Reference Manual*

  Describes the details of the syntax for the definitions you create and the commands you use to build the runtime components.

- **HP ACMS for OpenVMS** *Installation Guide*

  Provides installation requirements and steps to install *HP TP Desktop Connector* for ACMS software and verify the installation.

- *DECtp Implementation Toolkit*

  Provides valuable extensions to HP DECset tools to tailor them for developing an ACMS solution. Available through HP Services.

- *DECtp Design Toolkit*

  Provides valuable extensions to DECdesign to help design effective ACMS solutions. Available through HP Services.

For information on OpenVMS programming tools, refer to this document:

- *Using HP DECset*

  Describes the OpenVMS programming environment, provides helpful hints about conducting a software project, and shows a case study of HP DECset tools. Provided with the HP DECset documentation set.

  The ACMS documentation also describes how you can use the HP DECset tools to create an effective development environment.

Conventions

This guide uses the following conventions and symbols:

`User Input`	In examples, user input is differentiated in boldface type from system output.
$	The dollar sign indicates a generic command line prompt. This prompt may be different on your system.
Return	A key name in a box indicates that you press that key on the keyboard.
Ctrl/x	Press the Ctrl (control) key and hold it down while pressing the specified key (indicated here by x).
WORD	Uppercase text indicates OpenVMS data types, commands, keywords, logical names, and routines or services; C files and data structures; Microsoft Windows data structures; and HyperCard data types.
word	In format descriptions, a lowercase word indicates parameters, variables, and services or procedures.
italics	*Italic* type is used for emphasis and for parameters in text. Titles of manuals are also italicized.

[ ]	In format descriptions, square brackets surround a choice of options; select none, one, several, or all of the choices.
. . .	A vertical ellipsis in an example means that information not directly related to the example has been omitted.
TP Desktop	Refers to *HP TP Desktop Connector* for ACMS software.
CDD	Refers to the Oracle CDD product.
Rdb	Refers to the Oracle Rdb product.
Windows	When used alone, Windows indicates any supported member of the family of Microsoft Windows operating systems. Where necessary, specific Windows operating systems are mentioned. For a list of Microsoft Windows operating systems supported by the TP Desktop Connector product, see the product's *Software Product Description* (SPD).

Part I

Components and Design

Part I describes the components of the *HP TP Desktop Connector* for ACMS client services software and outlines basic design and development issues for HP ACMS solutions. It also discusses OpenVMS development procedures that are common across desktop platforms.

1

HP TP Desktop Connector Components and Processing

This chapter introduces the TP Desktop Connector components and processing phases.

1.1 TP Desktop Connector Components

The TP Desktop Connector software provides access to HP ACMS systems from desktop systems running their native operating system, such as Microsoft Windows. TP Desktop software implements a client/server computing model in which the following operations are typical:

- The desktop client program makes a **request** to the TP Desktop Connector software through an application programming interface (API).

- The TP Desktop Connector Gateway for ACMS (server) **responds** after processing the desktop client program request.

In the TP Desktop client/server model, you develop an application solution comprised of:

- One or more ACMS tasks
- Task group
- Application definitions
- Desktop client program

Figure 1–1 shows a simplified view of the TP Desktop components.

Figure 1–1 TP Desktop Components
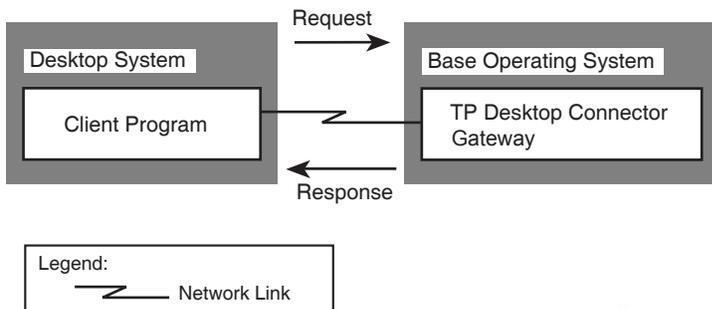
TAY-0414A-AF

You write desktop client programs to run on desktop systems using the **TP Desktop Connector client services**. Through the TP Desktop Connector client services, these desktop client programs send requests to and receive responses from the ACMS system. The TP Desktop Connector client services and the embedded network software on the desktop system allow the desktop system to transmit and receive TP Desktop Connector messages over a communications medium.

The desktop client program communicates with an ACMS system through the **TP Desktop Connector Gateway for ACMS** shown in Figure 1–1. The TP Desktop Connector client services call the TP Desktop Connector Gateway for ACMS to execute requests. As an ACMS agent, the TP Desktop Connector gateway communicates with other ACMS components for the desktop client program and sends the results of requests back through the mutually understood interface.

Access through TP Desktop software to an ACMS system requires a software configuration similar to that shown in Figure 1–2. Although your system need not be exactly the same as the one shown, your system must have at least:

- Desktop client program
- Network link
- TP Desktop Connector gateway system components
- ACMS TP System

Figure 1–2 TP Desktop Configuration
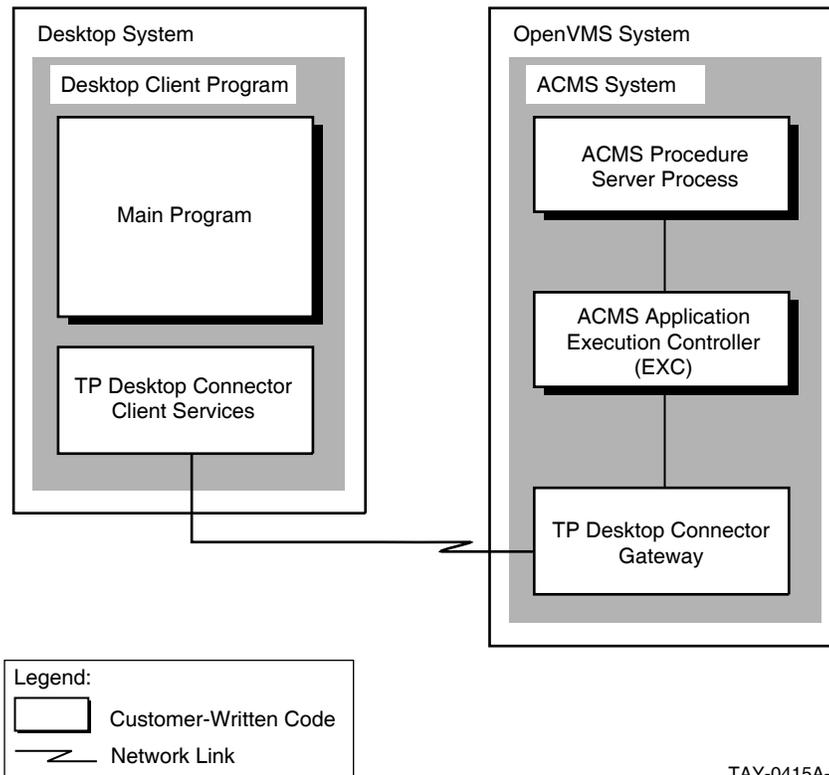
TAY-0415A-AI

1.1.1 Desktop Client Program Components

Desktop client programs interact with the user of the desktop system and access ACMS applications residing on the OpenVMS system. The TP Desktop components support access to ACMS software through the TP Desktop portable client services.

The TP Desktop portable client services support the following operating systems:

- Microsoft Windows
- Tru64 UNIX (See SPD for supported versions.)
- OpenVMS

The programming interface for the Microsoft Windows, OpenVMS, and Tru64 UNIX operating systems are all the same; both blocking and nonblocking client programs are supported. In addition, all supported operating systems can perform I/O tasks.

1.1.2 Network Components

The network software transmits messages between the desktop client program and the TP Desktop Connector gateway over the network link shown in Figure 1–2.

Although network software is required for TP Desktop software to work, the programmer does not need to understand networking to develop desktop client programs. The network software is largely transparent to the desktop client program. The TP Desktop client services shield the desktop client program from both the message protocol used by TP Desktop software and the networking services used to implement that protocol.

To communicate, both the TP Desktop Connector gateway and the desktop systems using TP Desktop software must have the appropriate network software installed. For example, OpenVMS can use either a DECnet or a TCP/IP transport. The user or system manager must install this software and ensure that a network connection links the desktop client program to the TP Desktop Connector gateway.

TP Desktop transparently handles all the network communications necessary to call ACMS tasks running on an OpenVMS system.

Table 1–1 lists the transports TP Desktop supports on OpenVMS.

Table 1–1 Transports for OpenVMS Systems

Clients	Transports
Windows	TCP/IP
OpenVMS	DECnet, TCP/IP
Tru64 UNIX	TCP/IP

Refer to **HP TP Desktop Connector for ACMS** *Gateway Management Guide* for detailed information on specifying a transport for communications.

1.1.3 TP Desktop Connector Gateway for ACMS System Components

The TP Desktop Connector Gateway for ACMS typically starts during system initialization. While ACMS software is running, the TP Desktop Connector gateway accepts TP Desktop sign-in, task start, and sign-out requests from the desktop client program. After the TP Desktop Connector Gateway for ACMS ensures that the desktop client program passes the authentication checks on that OpenVMS system, it processes requests from the program by dispatching them to the ACMS software. The OpenVMS system on which the TP Desktop Connector gateway runs consists of these major parts:

- ACMS system and customer-written ACMS applications

- TP Desktop Connector Gateway for ACMS

- Network software

In TP Desktop Connector systems, shown in Figure 1–3, the TP Desktop Connector gateway performs authentication and task invocation for desktop systems.

The TP Desktop Connector gateway handles user sign-in and is responsible for invoking tasks. The EXC calls the TP Desktop Connector gateway to handle exchange steps, passing the same type of workspaces and other data to the TP Desktop Connector gateway that it passes to the Command Process (CP).

The TP Desktop Connector gateway fulfills these roles:

- As a **gatekeeper**, the TP Desktop Connector gateway ensures that users of desktop client programs are authorized to access the ACMS system. The users of desktop client programs must pass through the full range of checks that are applied to a user on a VT terminal logging in to the OpenVMS system, and signing in to the ACMS system.

- As a **dispatcher**, the TP Desktop Connector gateway routes requests between the desktop system and the ACMS system.

For efficiency, the TP Desktop Connector gateway runs as a multithreaded process. It can handle multiple desktops and multiple requests simultaneously. The capacity is controlled by factors such as the maximum network links available on the OpenVMS system, and the licensing scheme. Like all ACMS processes, the TP Desktop Connector gateway is started before any user activity is requested. This prestarting takes care of all overhead activities before a user request is received. The TP Desktop Connector gateway design ensures effective use of resources, with optimal throughput and response time.

Figure 1–3 TP Desktop-Oriented Components

TAY-0287A-AD
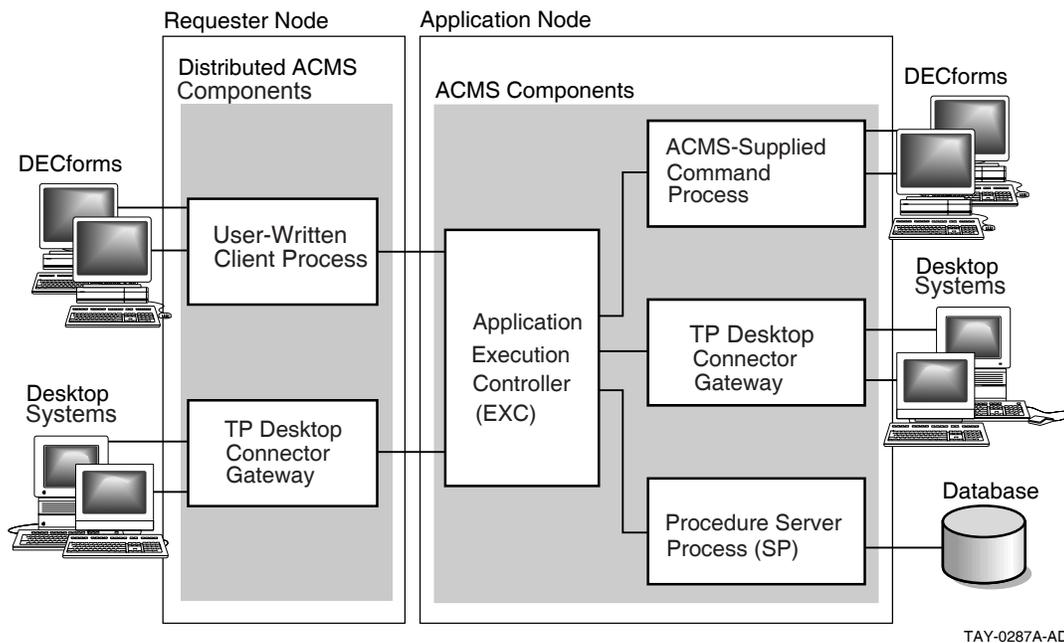
Because the TP Desktop Connector gateway complements the Command
Process (ACMS CP), using TP Desktop software has no impact on the rest
of the ACMS system. As shown in Figure 1–3, the TP Desktop Connector
gateway and the CP can run on the same system. The ACMS application,
task group, and task definitions, and the server procedures that support a VT
terminal can be used unchanged from a desktop system having TP Desktop
software.

VT terminals and desktop systems can use a single runtime instance of
an ACMS application simultaneously. The desktop systems use their own
presentation procedures, while the VT terminals use HP DECforms form I/O
or TDMS request I/O. The HP DECforms and TDMS runtime libraries are not
used on the desktop system.

The implementation of the TP Desktop Connector gateway as a complement
to the CP ensures that the benefits of the ACMS runtime system are retained.
This includes the efficient processing provided by the ACMS runtime system
and the capabilities that ACMS software supplies in application control,
availability, resource management, event-tracking, and task-level security. TP
Desktop Connector software allows you to take full advantage of the unique
benefits of both the desktop system and the ACMS software environment.

1.2 Desktop Client Program Processing

This section describes TP Desktop client services, program action routines, and general processing phases.

1.2.1 TP Desktop Client Services

The TP Desktop client services are callable routines that allow the desktop client program to do the following:

- Sign a user in to an ACMS system

- Run ACMS tasks

- Sign the user out of the ACMS system

These routines are shown as TP Desktop client services in Figure 1–2.

The desktop client program can run in a blocking, a nonblocking, or a forced nonblocking environment:

- In the **blocking** environment, the desktop client program calls a TP Desktop client service and is blocked awaiting a message from the TP Desktop Connector gateway.

- With the **nonblocking** versions of the portable client services, the desktop client program calls a TP Desktop client service, specifying a completion routine. The program regains control without having to wait for a message from the TP Desktop Connector gateway. When an TP Desktop message is received from the TP Desktop Connector gateway, the completion routine is called.

- The **forced nonblocking** environment allows presentation tools like Visual Basic to issue nonblocking calls even though these tools do not support pointer types to handle arguments passed by reference.

The nonblocking services are designed for multitasking event-driven systems like Windows or OSF/Motif.

1.2.2 Desktop Client Program Action Routines

TP Desktop software supports predefined action routines for checking versions of software on the desktop system. The desktop client program can provide a routine to check application-defined version information supplied from the system running the TP Desktop Connector gateway to ensure that software versions are compatible. See Section 4.3 for information on writing version-checking routines.

1.2.3 Phases of Desktop Client Program Processing

A desktop client program passes through several processing phases:

- Initialization phase

  During the **initialization phase**, a desktop client program creates necessary data structures and performs activities such as setting up the user interface and establishing a message-polling mechanism.

- Sign-in phase

  During the **sign-in phase**, a desktop client program establishes one or more sessions with the same TP Desktop Connector gateway or with different TP Desktop Connector gateways.

  The TP Desktop Connector gateway performs these functions:

  1. Authenticates the user for user name, password, time of day, and other OpenVMS login characteristics.

  2. Authenticates the user for ACMS sign-in.

  3. Assigns a unique session identifier that allows the desktop client program and the user to distinguish among multiple connections to one or many ACMS systems by one desktop user.

  A desktop client program can have multiple active sign-ins, however, a user can have only one task active for each sign-in.

- Task-selection phase

  During the **task-selection phase**, a desktop client program requests an TP Desktop client service to start a task in an ACMS application. The service sends a request to the TP Desktop Connector gateway to select the task.

- Task-processing phase

  During the **task-processing phase**, the ACMS system does the following:

  1. Checks the task access control list (ACL).

  2. Begins processing the task.

  3. Dispatches processing steps to procedure server processes.

  4. Dispatches exchange steps to the TP Desktop Connector gateway for transmission to the desktop client program.

  5. Determines task flow based on control information returned on each step.

6. Completes the task and sends a response to the desktop client program.

- Sign-out phase

  During the **sign-out phase**, a desktop client program must sign the user out of the ACMS system. On the desktop system, the desktop client program calls an TP Desktop client service, which does the following:

  1. Sends a request to the TP Desktop Connector gateway to sign the user out of the ACMS system.

  2. Disconnects the network logical link for that session.

Figure 1–4 illustrates the processing phases of a client program.

Figure 1–4 Desktop Client Program Processing Phases
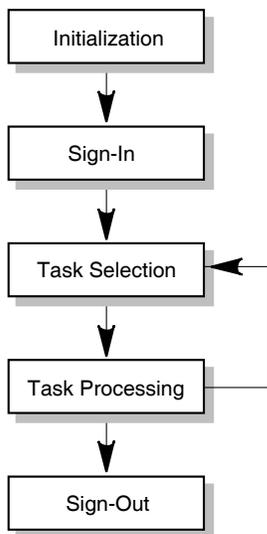
MR-5203-AD

Because of the authentication required during the sign-in phase, it is typically more efficient to have users remain connected across multiple task-selection and task-processing phases. The task-selection and task-processing phases repeat until the desktop client program explicitly requests to end the session.

1.3 Programming with TP Desktop Software

The TP Desktop software supports the following approaches for developing the desktop client program for an ACMS application:

- Develop the desktop client program within the desktop environment using desktop programming languages and tools.

  Frequently, this best ensures that the desktop client program takes full advantage of the capabilities unique to specific desktops.

- Develop the desktop client program within the OpenVMS environment, using languages and tools portable between OpenVMS and desktop systems. Move the source code to the desktop system to compile, link, and debug the program using desktop tools.

You can take advantage of expertise on one platform rather than maintain expertise across multiple platforms. For example, a desktop client program, written for the Windows system using portable tools, can also be run on the OpenVMS system. The programming interface for the TP Desktop client services for the OpenVMS system is the same as that for Windows. If the programming language and presentation tool are portable between OpenVMS and Windows systems, less in-depth expertise for the Windows system is required to create effective desktop client programs. With this approach, you can also take advantage of testing source control software available on the OpenVMS system.

In developing portable applications, you must still take into account certain differences between operating systems even if you are using portable tools.

Chapter 2 introduces the key design issues to consider in using TP Desktop software effectively. Later chapters describe how to build a desktop client program using each of the approaches outlined above, and explain the details of the TP Desktop client services and presentation code.

1.4 System Management with TP Desktop Software

TP Desktop software provides the ACMSDI$GET_SUBMITTER_INFO system management service for OpenVMS and a sample program demonstrating its use. This service can be used in utility programs running on the server to retrieve information about users currently connected to an ACMS system through TP Desktop software. The information returned by this service is necessary when the system manager needs to cancel a desktop client program session.

For documentation on the service and the location of a sample program that uses this service, see the **HP TP Desktop Connector for ACMS** *Client Services Reference Manual*.

See the **HP TP Desktop Connector for ACMS** *Gateway Management Guide* for information on managing TP Desktop systems and using the TP Desktop system management services. Also, the **HP TP Desktop Connector for ACMS** *Installation Guide* describes those aspects of system management related to installing TP Desktop.

1.5 TP Desktop Sample Application

Sample desktop client programs provided with TP Desktop software are based on the vehicle rental application called AVERTZ. The TP Desktop version of the sample includes similar tasks and procedures as the ACMS version of AVERTZ. This sample application illustrates the use of TP Desktop software to create multiplatform solutions supported by a common ACMS application.

This manual discusses design and development issues in the context of this sample. The sample desktop client programs illustrate many of the most common design considerations and provide a base of code from which you can build your solution. Most of the code shown as examples is taken from the AVERTZ sample application.

The sample desktop client programs are a key resource for understanding how to design and implement TP Desktop solutions. The software distribution kit contains sources and executables for each supported desktop system. After installation of the TP Desktop kit, you can find the source code on the OpenVMS system on which the kit was installed. Appendix A lists the directories containing the sample code.

The installation documentation describes how to set up and run the sample application on the host and the desktop systems. You can examine in detail how the various issues in building a complex, multiple-environment system are addressed in at least one solution. The samples are not intended as production applications. However, they help illustrate how to use TP Desktop software.

2
Designing TP Desktop Connector Solutions

This chapter outlines the major considerations in designing a TP Desktop Connector solution. The focus is on those areas that relate specifically to using the TP Desktop client services.

For a discussion of HP ACMS design in general, refer to the documents in the ACMS documentation set listed in the Preface section titled "Related Documents".

2.1 TP Desktop Connector for ACMS Design Questions

In designing solutions that use TP Desktop Connector software, you face significant design issues such as those listed in Table 2–1.

Table 2–1 Design Issues

Category	Issues
Processing	Structuring the desktop client program code Handling errors Implementing flow control
User interface design	Choosing presentation software Using graphics and other display tools effectively Handling multiple sign-ins
Data design	Data conversion Local caching of data Data compression Ensuring data integrity Handling validation

Some key issues, such as the choice of presentation software, span these categories. These issues are discussed in the following sections.

2.1.1 Understanding Application Requirements

Most important in ensuring an effective design is understanding what you are trying to build. The requirements of the application in such areas as performance, expense, usability, availability, data integrity, manageability, and maintainability may force you into difficult tradeoffs. The requirements you are addressing can interact or conflict. For example, you may want to increase message buffer size to improve network performance, but this forces you to consider adding memory to the machines, which increases cost.

A solution that meets requirements in one area can fall short of requirements in another. For example, an application that caches much information locally can have extremely good performance on tasks with local validation. But it can be costly in management overhead and network traffic to keep the validation data updated. Or an application that captures transactions and invokes only NO I/O tasks for periodic transmission of the transactions to a HP ACMS system may have exceptional availability. But it can restrict the kinds of validation available on the desktop system and, therefore, decrease the overall usability of the system.

Having explicit requirements for your application increases your ability to create an effective design. If you must trade off one requirement over another, you have better information with which to make your trade-offs.

2.1.2 Choosing Presentation Software

The presentation tool you choose can have a major impact on other design issues. For example, the presentation capabilities available in COBOL can be adequate for many character-cell applications. A portable tool like COBOL provides development benefits that may outweigh limitations in the application's user interface. On the other hand, different screen management tools on the market provide significant enhancements to the user interface, though at the cost of additional expertise required of the development staff.

HP recommends that you prototype at least part of your solution and then begin major development. Prototyping helps ensure that you understand any design issues unique to that tool before you begin implementation. It also helps to uncover any issues that might arise in using that tool in TP Desktop presentation code.

2.1.3 Strategy for Implementing the Solution

You can take either of the following approaches to implementing your solution:

- Design, build, and test the ACMS parts of the solution before building the presentation code.

- Start with the presentation code. After you build an effective user interface, implement the ACMS parts of the solution.

TP Desktop software supports both of these approaches equally well. Whichever approach you choose, ensure that the sequence and the context of messages represented by the task definitions are understood and handled by your presentation code.

2.2 Processing Design

This section explores the major considerations in designing the processing for solutions using TP Desktop Connector software. Part II describes how you structure desktop client programs for each platform.

2.2.1 NO I/O and I/O Tasks

From the point of view of the desktop client program, ACMS tasks fall into two categories:

- **NO I/O** (processing only) **tasks**

  This category of task has the NO TERMINAL USER I/O clause in the task definition. It is written to include only processing steps. The task itself does not specify input or output to the user; rather, the interaction with the user occurs outside the task. Any information required within the task or to be returned by the task is passed as task arguments. That is, workspaces declared for the task are passed as parameters on the call to the task and on the return from the task.

- **I/O tasks**

  This category of task has either the FORM I/O clause or the REQUEST I/O clause in the task definition. For applications that support VT terminals, most task definitions include both exchange steps and processing steps. In these tasks, interaction with the user is typically performed in the exchange steps of the task, using HP DECforms software, TDMS, or other presentation tools.

2.2.1.1 NO I/O Task and TP Desktop Connector Interaction

Example 2–1 shows a task definition and annotations describing how the task statements are handled.

Example 2–1   NO I/O Task Actions for TP Desktop Connector Users

```
REPLACE TASK employee_lookup
USE WORKSPACES
  employee_number,
  employee_record;
TASK ARGUMENTS ARE
  employee_number WITH ACCESS READ,    1
  employee_record WITH ACCESS MODIFY;

BLOCK WORK WITH NO TERMINAL I/O    2
 PROCESSING
 get_employee:
        PROCESSING WORK
           IS CALL PROCEDURE get_employee    3
              IN SERVER employee_server
              USING  employee_number,
                     employee_record;
END BLOCK WORK;
END DEFINITION;     4
```

The annotations in Example 2–1 indicate the following:

1 The task arguments are workspaces passed to and received from the desktop client program.

2 The task is declared to have no terminal I/O.

  NO I/O tasks have no exchange steps, with the NO TERMINAL USER I/O clause specified for the block.

3 The task calls server procedure get_employee in the ACMS application.

4 The task returns workspaces with possible changes to the desktop client program.

TP Desktop Connector software handles each NO I/O task by accepting workspaces and other arguments from the desktop client program at the start of the task, and sending workspaces and other arguments to the TP Desktop Connector client services at the end of the task.

2.2.1.2 I/O Task and TP Desktop Connector Interaction

TP Desktop Connector supports tasks that have exchange steps as well as tasks that have only processing steps. TP Desktop Connector software handles each exchange step by sending task workspaces and other arguments between the TP Desktop Connector client services and ACMS by means of the TP Desktop Connector Gateway for ACMS. Example 2–2 shows a task definition and annotations describing how the task statements are handled.

Example 2–2 I/O Task Actions for TP Desktop Connector Users

```
REPLACE TASK employee_lookup
USE WORKSPACES
  employee_number,
  employee_record;
BLOCK WORK WITH FORM I/O   1
 IS
 get_employee_number:
       EXCHANGE WORK IS
        RECEIVE FORM RECORD get_employee_key  2
          IN FORM employee_form
          RECEIVING employee_number;
 get_employee:
       PROCESSING WORK
         IS CALL PROCEDURE get_employee  3
            IN SERVER employee_server
            USING  employee_number,
                   employee_record;
  display_employee:
       EXCHANGE WORK IS
        SEND FORM RECORD display_employee_record  4
         IN FORM employee_form
         SENDING employee_record;
END BLOCK WORK;
END DEFINITION;                       5
```

The annotations in Example 2–2 indicate the following:

1   The task is declared with the FORM I/O clause.

   This indicates that HP DECforms syntax is used for the exchange steps.

2 A request is received from the user.

For TP Desktop Connector users, the task receives workspaces and workspace counts from the desktop client program through the TP Desktop Connector client services.

For nondesktop users, the HP DECforms request FORMS$RECEIVE is called by ACMS.

3 The task calls procedure get_employee in the ACMS application.

4 A response is sent to the user.

For TP Desktop Connector users, the task sends workspaces and status information to the desktop client program through the TP Desktop Connector client services.

For nondesktop users, the HP DECforms request FORMS$SEND is called by ACMS.

5 The task ends.

The format of the interaction between the desktop client program and the ACMS task depends on the following factors:

- The desktop system in use

- Whether the desktop client program uses the blocking or nonblocking form of the services and presentation code

2.2.1.3 Choosing Between I/O and NO I/O Tasks

TP Desktop Connector software allows you to call I/O tasks (FORM I/O and REQUEST I/O) and NO I/O tasks (NO TERMINAL I/O) from a desktop client program. Choosing between I/O and NO I/O tasks is a choice of where to place control of the application flow. With I/O tasks, the task controls the application flow. With NO I/O tasks, the client program controls the application flow.

If the application supports only desktop systems or you are optimizing the ACMS application for desktop access, consider using only NO I/O tasks.

NO I/O tasks fit better with event-driven desktop systems, such as Microsoft Windows. With these systems, the desktop client program rather than the ACMS application controls the major processing flow.

NO I/O tasks also allow the greatest freedom in handling the logic in the desktop client program. However, task invocation can be more expensive than invoking processing steps within a task. Each task invocation requires workspace management by the ACMS Application Execution Controller (EXC), which is considerably more expensive in processor time than the workspace management done for exchange steps. Also, the ACMS system requires

processor time to set up control structures when the task starts and to clear the structures when the task ends.

The additional central processor usage for task invocation compared to processing step invocation is generally small. However, the extra overhead for task invocation can be a significant factor in a system with high throughput requirements.

2.2.2 Application Design Approaches

Two approaches to a ACMS solution are the following:

- Common applications

  The solution is common to both desktop systems and VT terminals.

- Tailored applications

  The solution involves multiple applications in which the tasks are tailored to either the desktop system or the VT terminal.

2.2.2.1 Common ACMS Applications

TP Desktop Connector software allows you to structure a common ACMS application to be accessed from both desktop systems and VT terminals. The task definitions, workspace definitions, task group definition, and application definition can be the same.

If users at both VT terminals and desktop systems in your enterprise need to run the same tasks, it is best to have a common ACMS application. This minimizes your development and maintenance work. If you use a presentation tool that is portable between OpenVMS and the desktop system, possibly no additional code is required to support the presentation software on both platforms.

To use NO I/O tasks in an environment that has both desktop systems and VT devices, you can take advantage of the task-call-task feature of ACMS. Divide common processing steps into standalone NO I/O tasks. The desktop client program calls these NO I/O tasks directly. The VT-terminal users request I/O tasks that include exchange steps. These I/O tasks then use the task-call-task feature of ACMS to invoke the common NO I/O tasks as part of their processing steps. Any task error recovery within the common NO I/O tasks is shared by both presentation device types.

Maintenance increases slightly in this environment, because flow control is duplicated in the tasks called by the VT-terminal users and within the desktop client program.

2.2.2.2 Tailored ACMS Applications

If your TP Desktop Connector solution does not include VT terminals, you can tailor the ACMS application and design the tasks somewhat differently from a common application. Use one of the following alternatives:

- Specify the NO TERMINAL USER I/O clause in the task definition, so that the ACMS tasks are NO I/O rather than I/O tasks.

- If you use FORM I/O or REQUEST I/O tasks, place more flow control in the desktop presentation code.

  ACMS applications that use HP DECforms or TDMS forms for VT-presentation code can include a significant amount of flow control logic. Because you can write TP Desktop Connector presentation code in third-generation languages, you can include similar or more powerful flow control logic in the desktop client program, just as you include in HP DECforms or TDMS forms.

  However, the ACMS task definition is effective for representing the structure of a task clearly. For improved maintainability, represent the flow of a task in its definition rather than controlling the flow through its step procedures or its exchange steps.

- Have the desktop system focus on data capture, intermittently transmitting transactions to the ACMS system for processing as queued tasks.

These design approaches can also be applied to solutions that include both VT terminals and desktop systems. For example, to decrease network traffic, use a queued task model for the desktop systems and retain an interactive model for your VT terminals.

TP Desktop Connector software does not preclude optimizing the ACMS application for desktop systems and retaining a separate but comparable application for VT terminals. You can optimize the ACMS application by building separate desktop tasks within the same application definition or by building alternatives into the task definitions. However, carefully consider the additional maintenance cost required for such a solution.

2.2.2.3 Conversion of a Task: I/O to NO I/O

If you have I/O tasks and want to incorporate a desktop design in the application that does not depend on terminal interaction, you can convert the I/O tasks to NO I/O tasks. Converting an I/O task to a NO I/O task involves the following steps:

1. Delete the exchange steps.

2. Separate the processing steps into individual tasks.

3. Modify the action clause for each processing step to eliminate references to exchange steps.

4. Add a TASK ARGUMENTS clause for any workspaces used by a NO I/O task.

5. Add the NO I/O tasks to the task group definition.

Task conversion can be done without affecting anything else in the application.

2.2.3 Preventing Loss of Work with Local Data Capture

In some applications, desktop systems are most effective as relatively independent data capture resources. Users work with reference data stored locally on their personal computer, building up a queue of completed transactions awaiting their forwarding to a ACMS system. The desktop system signs in to the ACMS system and uploads the transactions from the desktop system to the ACMS system.

TP Desktop Connector software can effectively support the data capture model for desktop systems. Figure 2–1 shows how such a queue-based system can work.

The TP Desktop Connector software does not directly provide queuing facilities, for example, services for data capture. The desktop client program can create and manage a file of transactions to be transmitted to the ACMS system. It also can determine when to establish the connection, such as at regular intervals, or when the queue reaches a predefined threshold. The desktop client program can then use the TP Desktop Connector client services to invoke tasks in a ACMS application that either process the queued items directly or run other tasks to move the items into ACMS queues on the ACMS system.

The TP Desktop Connector client services do not guarantee that an item queued on the desktop system is processed only once. If this guarantee is required, the application must supply a capability, such as having the receiver application check a queue item identifier for duplicates before storing a transaction.

Figure 2–1   Using a Queued Task with TP Desktop Connector
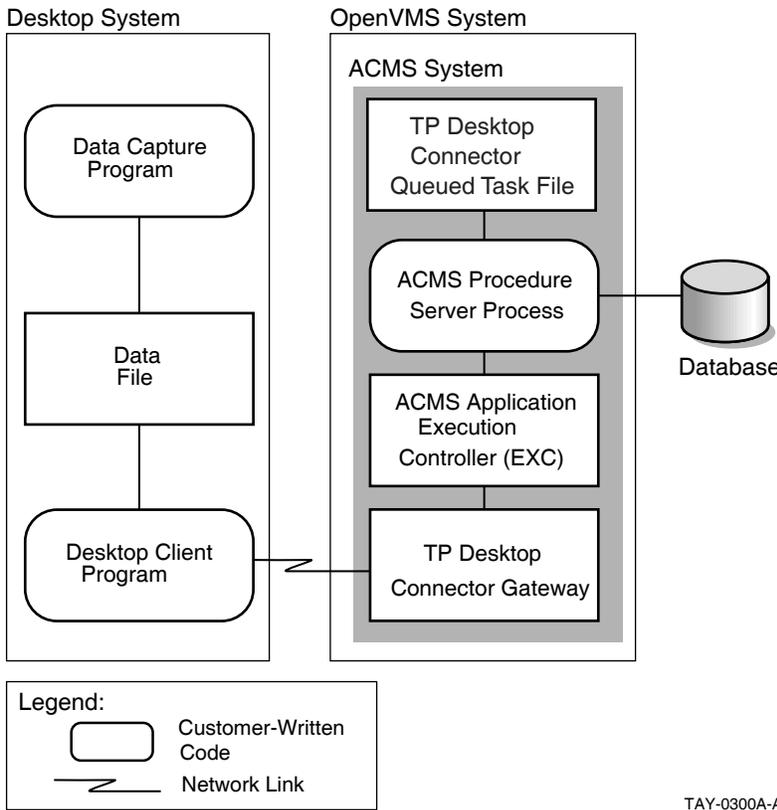
TAY-0300A-AD

2.2.4 Event-Driven Systems and the Nonblocking Environment

In event-driven systems like OSF/Motif and Windows, a program is structured
to include an event-dispatching mechanism (event dispatcher) and a collection
of procedures that this dispatching mechanism invokes as the result of an
event. Ideally, these procedures quickly return control to the event dispatcher
so that other events are dispatched (or processed) without delay. If these
procedures fail to return control quickly, slow and undesirable behavior
in programs can result. This characteristic is especially true for Windows
programs. When a procedure hangs in any given Windows program, the
system cannot give control to another Windows program.

By providing nonblocking services, TP Desktop Connector software allows a desktop client program to release control to the event dispatcher (in Microsoft Windows the message dispatching loop) without waiting for the ACMS system to respond. This nonblocking characteristic is particularly important when the network delays a response or the ACMS system performs extensive processing.

Because the release of control to the event dispatcher is so critical in event-driven systems (for getting user input, and so on), TP Desktop Connector supports nonblocking presentation procedures as well as the basic nonblocking client services.

For a description of forced nonblocking, see Chapter 7.

2.2.5 Nonblocking Design Considerations

Consider the following when designing your desktop client program for a nonblocking program and runtime environment:

- Return control to Windows

  In event-driven environments, structure desktop client programs to give up control to the event dispatcher as soon as all of the work is done and that it can be completed without a significant delay.

  - For service requests

    When requesting a nonblocking version of a TP Desktop Connector client service, promptly return control to Windows. The program does not wait for the server response to the request.

  - For presentation procedures

    When a presentation procedure request is received from the desktop gateway, return control to Windows promptly. The program does not wait to send a response to the gateway. To use Windows features such as dialog boxes for exchange step processing, the program must first return control to Windows to handle the user interaction with the dialog box.

- Establish a mechanism to poll for incoming TP Desktop Connector messages

  The desktop client program must supply and activate a procedure to periodically check for messages received from the gateway.

The AVERTZ sample applications for Motif and Microsoft Windows use a procedure to activate timer events so that pending TP Desktop Connector Gateway for ACMS messages can be processed. To process a timer event, the timer-event handler calls the TP Desktop Connector service acmsdi_dispatch_ message to poll for and dispatch messages from the gateway.

2.2.6 TP Desktop Connector Gateway for ACMS Availability

Similar to distributed ACMS configurations for VT-terminal users, a desktop
system can connect a desktop client program to multiple submitter nodes that
in turn invoke applications on application nodes. This can be achieved by
using multiple copies of the desktop client program or a single copy initiating
multiple ACMS sign-ins.

As discussed in Chapter 1, if the application node fails in a distributed
configuration, the user can automatically be routed to another ACMS system
on the next task selection. Having a submitter node responsible for sign-in
distinct from a ACMS system that actually processes the application requests
can provide a high degree of availability without any extra work in the desktop
client program. For example, in Figure 2–2, if the application Node A fails,
the TP Desktop Connector Gateway for ACMS can automatically reroute
subsequent requests to Node B.

Figure 2–2 Application Node Failover Configuration

TP Desktop Connector Application Environment

```
                  ┌─────────────────────┐
                  │                     │
                  │   Desktop System    │
                  │                     │
                  └──────────┬──────────┘
                             │
                  ┌──────────┴──────────┐
                  │     TP Desktop      │
        ┌─────────┤      Connector      ├─────────┐
        │         │    Gateway Node     │         │
        │         └─────────────────────┘         │
        │                                         │
┌───────┴────────┐                     ┌──────────┴───────┐
│                │                     │                  │
│                │                     │                  │
│     Node A     │                     │     Node B       │
│                │                     │                  │
└────────────────┘                     └──────────────────┘
```
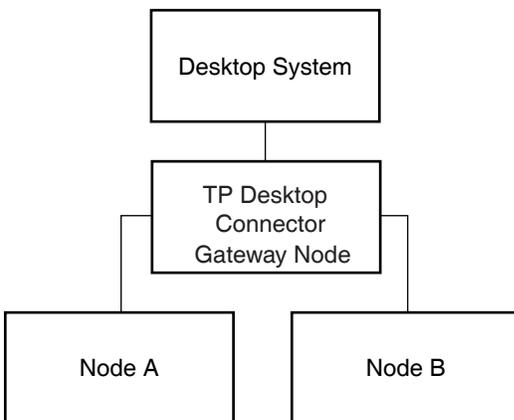
TAY-0288A-AD

If the submitter node to which desktop users are currently attached fails, the
TP Desktop Connector client services do not automatically route signed-in
users to a new submitter node. This is true even if the user is connected to one
of several nodes in a VMScluster system. Because authentication of the user
is performed on the submitter node, a failure of the submitter node invalidates
the authentication of the user.

A desktop client program can provide failover for submitter nodes. Because a desktop client program can support multiple active sign-ins (though not multiple active tasks within a single sign-in), the user can have sign-ins active on multiple submitter nodes. The desktop client program explicitly signs the user in to several submitter nodes. If the primary submitter node fails, the desktop client program can sense the failure and switch to another sign-in. The user is already signed in to an alternative system and is ready to continue operations more rapidly. In this case, as shown in Figure 2–3, the desktop client program can be written to hide the submitter node failure from the user.

Figure 2–3 Submitter Node Failover Configuration

TP Desktop Connector Application Environment

TAY-1000A-AD
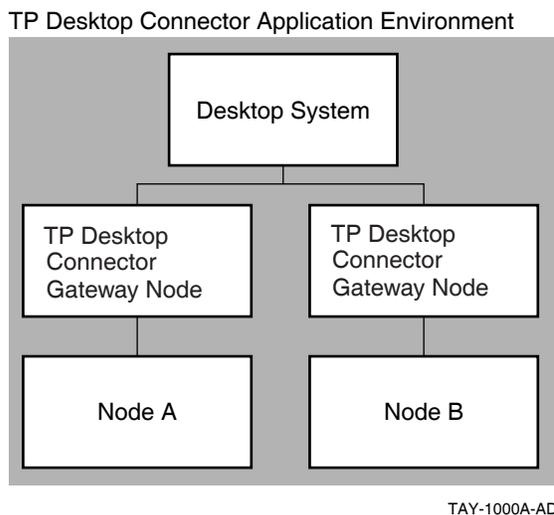
2.2.7 Error Handling

As with any ACMS solution, some application errors are handled directly by the ACMS software. For example, fatal errors that are not handled by customer-written server procedures can cause the ACMS system to cancel a task. In general, however, the TP Desktop Connector software returns any error it receives to the desktop client program. Therefore, the desktop client program can appropriately handle virtually all error conditions.

With TP Desktop Connector sign-in, task call, and sign-out, the desktop client program can trap the status returned by these operations and take appropriate action. The ACMSDI return status values identify conditions to which the desktop client program should respond. These conditions include sign-in

failures, task processing errors, TP Desktop Connector Gateway for ACMS failures, and other error conditions. For example, ACMSDI_NORMAL is the return status indicating successful completion of TP Desktop Connector client services. If the ACMSDI_INTERNAL error is returned, the desktop client program should end any active tasks, sign the user out of the ACMS system, and instruct the user to exit from the application.

Error handling for exchange steps in I/O tasks is straightforward. When a ACMS task is canceled, the TP Desktop Connector Gateway for ACMS reports a specific error where possible. If the gateway cannot convert a ACMS error to a specific TP Desktop Connector status, it returns ACMSDI_TASK_FAILED to the desktop client program.

If the desktop client program encounters an error during its own processing of an exchange step, that error can be handled within the desktop client program. The desktop client program can then return status information in a workspace to be evaluated in the task definition for appropriate action. The desktop client program can also return any valid OpenVMS error code (including HP DECforms, TDMS, and application-defined values) to the ACMS system to instruct ACMS to continue or cancel the task.

2.2.8 TP Desktop Connector Gateway for ACMS Error Checking

The TP Desktop Connector Gateway for ACMS responds to recoverable thread-level errors by dropping the thread in a controlled way, causing that thread to be cleaned up while insulating other threads from any ill effects. The gateway treats some errors (such as access violations) as fatal to the gateway, thereby dropping all active threads.

These checks detect errors from the following sources:

- Network corruption

  If a network router corrupted packets, the gateway fails while trying to interpret the ill-formed messages, dropping all threads.

---------------------------- **Note** ----------------------------

TP Desktop provides the option of cyclic redundancy checking (CRC), which can detect this type of network corruption. See the **HP TP Desktop Connector for ACMS** *Gateway Management Guide* for information on how to activate this feature.

- TP Desktop client program errors

  Client program errors can sometimes destroy or overwrite data that the TP Desktop client services send to the gateway. Typically, an ID value is destroyed. When the gateway detects the bad ID, TP Desktop disconnects that client, rather than causing the gateway to fail.

2.3 User Interface Design

This section discusses the considerations in designing an effective user interface for your desktop client program. Some aspects of the interface, such as application flow, are discussed in Section 2.2. This section focuses on those questions that are most directly related to what the user sees on the display device.

Following these and other published user interface design principles can maximize user productivity:

- Consistency across applications

  As with any application, strive for a consistent user interface across applications, so that users do not have to learn new rules to do different kinds of tasks.

- Appropriate operating metaphors

  Provide a model for your interface that is familiar to your users. For example, just as using a map can help users efficiently locate an office, using an existing form as a model for a screen display can help users learn the application.

- Multiple sign-ins

  TP Desktop Connector software allows a desktop client program to establish multiple sign-ins to one or more ACMS systems. Having multiple sign-ins increases application availability, as discussed in Section 2.2.6. However, each sign-in increases network resource usage and consumes licenses. If the application has a large user base, the desktop client program might need to restrict the number of sign-ins allowed for each desktop system.

  A desktop client program that supports multiple sign-ins can also increase the user's ability to work effectively with complex functions. For example, complementary tasks can be available in different ACMS systems or in different applications on the same ACMS system. Each sign-in can be assigned its own window by the desktop client program. The desktop client program can run a task in each of these windows, allowing the user to cut

and paste information from one window to another, or compare information in the two windows.

Having the desktop code interact with NO I/O tasks rather than FORM I/O and REQUEST I/O tasks increases the usefulness of multiple active sign-ins. The greatest amount of control is retained on the desktop system. However, even with FORM I/O and REQUEST I/O tasks, having multiple active sign-ins can significantly increase the ability of the user to pull together many different tasks to perform a complex activity.

- Effective graphics use

  The presentation capabilities of the desktop environment are one of the great benefits of the desktop. Presentation tools that work within the desktop environment can provide significant benefits in the following areas:

  - Increasing clarity of information presented to the user

  - Structuring the presentation to increase the usability of the application

  - Tailoring the presentation to suit individual user needs

  Take advantage of presentation capabilities. However, because overuse of video attributes, fonts, and other features can be confusing, ensure that the presentation software contributes to the effectiveness of the application.

- Restricting user functions

  In event-driven environments, it is important for the user interface to restrict access to certain functions, such as by dimming icons. The design of the user interface must prevent the user from invoking tasks that are invalid under certain conditions. For example, the user interface needs to prevent a task from being invoked using a sign-in that is currently processing another task, and to prohibit signing out of the ACMS system when a task is being processed.

2.4 Data Design

The ability to have presentation code native to desktop systems communicate with application code native to OpenVMS systems raises many new questions. The following sections discuss how the software handles differences in data types between the various environments.

2.4.1 Data Conversion

Because TP Desktop Connector software does not manipulate workspace contents, it does not constrain what data types are used on the desktop system or in the ACMS application. However, TP Desktop Connector software does not automatically handle data conversion between the OpenVMS and desktop environments. In many cases, you can build your desktop client program such that data types are compatible between the desktop system and the ACMS system. If you have incompatible data between the systems, the application code must perform any conversion required.

The most commonly used data types are readily mapped between OpenVMS and desktop systems. Part II contains sections that list the data types that must be converted for each platform.

Typically, data conversion is done most effectively in the desktop client program. Doing so takes advantage of the processing power of the desktop system, and ensures the portability of the ACMS application.

2.4.2 Data Alignment with RISC Architecture Clients and OpenVMS Servers

The RISC architecture requires that data references be naturally aligned. That is, short words (2 bytes) must be on an even byte boundary. Long words (4 bytes) must be accessed on a boundary evenly divisible by 4.

When Alpha and I64 clients define a C structure, they create padding in the structure, if necessary, to ensure that each field complies with these requirements. (The padding is not visible to you.)

OpenVMS, however, does not impose such restrictions on its data objects and does not pad its structures. The problem arises when data, defined on one of these machines, is transmitted across the network to the other machine, and interpreted using the same C structure definitions.

In a TP Desktop application, this is a concern only when ACMS workspaces are being sent (in either direction) between the client program on a RISC machine such as Tru64 UNIX Alpha and the ACMS application on OpenVMS.

Consider the following structure:

```
struct wksp_type
{
  char  field_1;
  short   field_2;
  char  field_3;
} my_wksp;
```

On an Alpha or an I64 client, my_wksp is stored as:

1003		1002	1001	
field_2		pad	field_1	1000
			field_3	1004

TAY-0328-AD
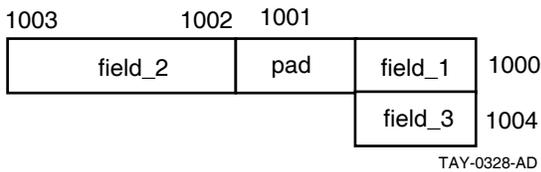
On an OpenVMS server, my_wksp is stored as:

1003	1002	1001	
field_3	field_2	field_1	1000

TAY-0329-AD
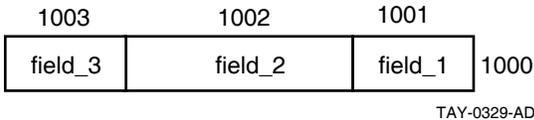
If my_wksp is allocated on the OpenVMS server and sent to an Alpha or an I64 client, the structure definition on the Alpha or an I64 machine is expecting padding that is not there. At run time, your application on the Alpha or I64 machine generates the following message:

```
Fixed up unaligned data access for pid nnnn (appl_name) at pc 0xnnnnnnn
```

One approach to dealing with this problem is to allocate new workspaces in the Alpha or I64 client program, and perform a byte copy of each field of the incoming OpenVMS server structure into the fields of the new client structure. For structures going from a RISC machine to OpenVMS, you also need to perform a byte copy of each field of the RISC structure to the outgoing data stream. If you are writing a portable application, this solution should not have an adverse effect when running on non-RISC platforms.

Example 6–9 and Example 6–10 show this approach from the AVERTZ Motif sample application.

2.4.3 Data Validation

TP Desktop software does not provide a mechanism for data validation on the desktop system. If you want to cache data on the desktop system for functions such as range checks and choice lists, the application must be responsible for handling the distribution of data to the desktop system. The application must also handle the updating of the local data to ensure that it is up to date.

In many cases, the validation on the desktop system can be restricted to fairly static and unchanging data. For example, a list of geographic codes can be built as an external table referenced by the desktop client program with minimal concern. Less static information, such as part numbers, employment codes, and site locations, might nonetheless be stable enough so that you can

distribute revised information easily. It is valuable to have as much of this validation on the desktop system as possible.

Windows desktop client programs can use the TP Desktop version-checking capability to ensure that the tables are current.

Caching of more dynamic information is also possible. However, it is difficult with currently available tools to ensure consistency between highly dynamic data on the desktop system and data in a ACMS system.

2.4.4 Data Integrity

Like ACMS software, TP Desktop software generally relies on the data management systems employed in the ACMS system to ensure the integrity of data. For example, the transaction semantics of database and distributed transaction software ensures that all activities within a transaction are atomic. TP Desktop relies on the locking capabilities of database software to ensure that all activities within a database transaction are serialized and that they can be rolled back, if necessary.

In most cases, complex update transactions imply displaying data to the users, allowing them to change it, and then applying those changes to the database. The cost of locking the data while the user looks at and modifies it is typically prohibitive. Therefore, other mechanisms are generally employed in the application code to allow the locks to be released during this user interaction. For example, to ensure that the software can detect changes made by another user while the lock is released, a copy or a sequence number of the original data is retained.

TP Desktop software does not alter the data integrity strategies normally used with ACMS systems. The same issues must be addressed, and the same solutions generally apply. The areas in which desktop solutions may be more restricted are in queued transactions and in local caching of data discussed in Sections 2.2.3 and 2.4.3. Tools that are available in homogeneous OpenVMS solutions, including distributed solutions, can provide a level of data integrity for OpenVMS solutions that is not available to solutions involving multiple platforms.

2.4.5 Workspace Design

As with distributed ACMS systems, the factor having the single greatest effect on application response time is typically the size of the messages being sent over the network. In an TP Desktop system, all message traffic during task processing is based on the workspaces defined for the task. To minimize the cost of network data transfer on each task invocation, exchange step, and task end, design the application workspaces with the following guidelines:

- Take advantage of unidirectional workspaces for client programs.

  TP Desktop lets you specify whether a workspace containing task arguments is sent from the client to the gateway (read-only), from the server to the client (write-only), or in both directions (modify). Using read or write access reduces the overall network cost of the task. Note that the read and write are from the perspective of the server. See Section 2.4.5.1 for more information about unidirectional workspaces.

- Take advantage of HP DECforms syntax in task definitions.

  Using HP DECforms syntax in the exchange steps allows different size workspaces in send and receive messages. Other syntax requires that workspaces be the same size in both directions.

- Keep the workspaces as small as possible.

  In tasks with great variations between data sent in one exchange step and data sent in another, create separate workspaces for each step rather than reusing a single workspace.

- Minimize the number of workspaces.

  A small incremental cost exists for each workspace transmitted either as a task argument in a NO I/O task or as data on an exchange step. For task arguments and exchange steps, use as few workspaces as possible, keeping each workspace and the total data sent as small as possible.

- Transmit a usable amount of data.

  In tasks that allow users to scroll through large amounts of data, return the data one page at a time. By caching each page on the desktop system as it is sent, you can ensure that the desktop client program does not have to return to the ACMS system to retrieve data already sent. With this approach, you ensure that users do not have to wait for numerous relations to be read from the database and sent to the desktop client program when the information they need is in the first relation.

- Compress task argument workspaces.

  TP Desktop supports compression on task argument workspaces. See Section 2.4.6 for information on compression.

- Take advantage of data definitions.

  If the data is drawn from multiple relations in the database, the workspace definitions may not map to database relations. However, you can base workspace definitions on the same field definitions that make up the database tables.

For TP Desktop performance, the most critical issue is to balance the workspace size and the number of workspaces. Minimize both the amount of data transferred in each message between the desktop system and the ACMS system and the number of messages being sent.

2.4.5.1 Unidirectional and Bidirectional Workspaces

TP Desktop provides a mechanism, called **unidirectional workspaces**, by which you can control how data is sent across the network. With unidirectional workspaces, only those workspaces that actually contain data need to be sent in either direction. Bidirectional workspaces send data in both directions. By specifying which workspaces need to go in one direction only, the application can significantly reduce unnecessary network traffic.

TP Desktop provides three types of workspaces, two of which are unidirectional:

- Read-only (undirectional)

  Read-only workspaces contain data that the client application needs to send to the ACMS task. The task needs to receive the data, but it does not need to update the workspace (or write into it). Therefore, you need to send read-only workspaces only in one direction: from the client application to the task.

  Read-only means that these workspaces are read-only from the server's perspective. This meaning is consistent with the ACMS task definition syntax.

- Write-only (unidirectional)

  Write-only workspaces contain data that the task needs to send to the client application. The client application does not send data in these workspaces to the ACMS task. The ACMS task writes values into these workspaces to send back to the client application. Therefore, you need to send write-only workspaces only in one direction: from the ACMS task to the client application.

  These workspaces are write-only from the server's perspective. This meaning is consistent with the ACMS task definition syntax.

- Modify (bidirectional)

  Modify workspaces contain data that the task needs to send to the ACMS task. The ACMS task:

  — Uses the data in these workspaces.

  — Makes updates to the workspaces.

  — Returns the updated workspaces to the client application.

Therefore, you must send modify workspaces in two directions: from the client application to the ACMS task, and from the ACMS task back to the client application.

Modify workspace is the default. If you do not specify the access type of the workspace to pass, TP Desktop sends all the workspaces that are passed to it in both directions.

You specify the access type of the workspace on the call to the acmsdi_call_task client service. See the *HP TP Desktop Connector for ACMS* *Client Services Reference Manual* for information on the acmsdi_call_task client service. For information on passing unidirectional workspaces with Visual Basic, see Chapter 7.

2.4.6 Data Compression

TP Desktop provides compression for task argument workspaces; workspaces on exchange steps are not compressed by any TP Desktop service. However, TP Desktop client services can accept compressed exchange-step workspaces and can decompress them before passing them to client applications. This section presents the call options required to compress task argument workspaces. Section 4.2.5 describes how to use compression with the Portable API.

The following rules apply to TP Desktop data compression:

- Data compression for task argument workspaces are initiated by the client application in the form of requests to the client services layer.

- Compression and decompression of exchange step workspaces are transparent to the client application.

- Data compression is activated for a call. Once activated, compression can be requested on a workspace-by-workspace basis.

- Although the client application can request compression for individual workspaces, the application never sees a compressed workspace, nor does the client application have access to the algorithm being used for compression.

- If compression is requested for a MODIFIABLE workspace, attempts are made to compress that workspace in both directions.

- The client services layer attempts to compress READ-ONLY and MODIFIABLE workspaces, if requested.

- The client services layer passes the application's compression requests to the back-end for MODIFIABLE and WRITE-ONLY workspaces, which causes ACMS to attempt to compress those workspaces.

- A compression attempt on a workspace can result in the total workspace size, including workspaces header data, being equal to or greater than the original, uncompressed, workspace size. If this is the case, the client services layer sends the workspace in its uncompressed form. The client application is not informed when this happens.

- If compressed workspaces are sent to an ACMS system, which does not support data compression, the gateway rejects the task call with a special error message.

2.5 Design Conclusions

The design issues discussed in this chapter are highly dependent on the design of the solution as a whole. For example, although applications should generally avoid using large workspaces, a given application can have excellent reasons for using them. By understanding all your own application requirements, you can decide on each of the issues raised in this chapter.

The design topics discussed in this chapter focus on desktop issues. Larger issues concerning the entire application environment are equally important. For example, if the data storage and access design of the ACMS system results in slow response time, that can have as immediate and visible an effect on the user interface as any decisions about presenting data to the user on the desktop system. Likewise, confusing flow control is a problem whether it is embedded in the task definition or in application-specific presentation code.

In creating a desktop solution, be sure to pay close attention to these larger issues, as well as the desktop-specific concerns. See the documentation for ACMS, Oracle CDD, Oracle Rdb, and other related products for information on these other areas of application design.

3

Developing HP ACMS Applications

This chapter presents guidelines for development work you do on the
OpenVMS system for use with desktop client programs.

3.1 Overall Development on the OpenVMS System

Some elements of your TP Desktop Connector solution must be developed
and run under OpenVMS software, regardless of where you develop your
presentation code. One of the key elements of the solution is the ACMS
application called by your desktop client programs. To create the ACMS
application, do the following on the OpenVMS system:

1. Create and populate the databases that your ACMS applications access.

2. Create definitions for the tasks in your ACMS application and workspaces
   for these tasks.

3. Write step procedures for the processing steps in your tasks.

4. Write task group and application definitions.

5. Build the task group and application database.

6. Link the procedure server image.

In developing these elements of your desktop solution, you perform exactly
the same steps as in developing ACMS solutions that do not use TP Desktop
Connector software.

Refer to the ACMS documentation listed in the Preface section titled "Related
Documents".

If your solution includes both VT terminals and desktop devices, create the
presentation code for the VT terminals on the OpenVMS system.

If you are not using VT devices and are using FORM I/O or REQUEST I/O
tasks, do not build the HP DECforms form libraries or TDMS request libraries
for the exchange steps in your task definitions. See Section 3.3.

3.2 Creating Data Definitions for the Desktop System

The maintenance cost of the ACMS solution can be reduced by having common data definitions in the ACMS application and in the desktop client program.

Note

The command procedure MAKE_RECORDS.COM found in the ACMSDI$EXAMPLES directory provides an example that automates creating common data definitions and converting data for the C and COBOL languages. You may need to edit the command procedure for the language you are using.

To ensure common data definitions, create include files for the languages being used as follows:

1. Define workspaces in CDD software (using the Operator utility CDO).

2. From the CDD definitions, create an include file in each third-generation language in which your desktop client program is written:

   a. Create a dummy OpenVMS program that includes or copies the CDD definitions from the dictionary.

   b. Compile the dummy program to extract the definitions and create a listing file with their language equivalents.

   c. Edit the listing file and extract each definition to create an include file.

3. Ensure that the include files are compatible with the desktop compiler you are using.

   For example, replace underscores in Microsoft COBOL with dashes, and convert the data type *int* in VAX C to *long int* for Microsoft C.

The include files are for use in the presentation code on the desktop systems. For example, if the desktop client program is written in C and COBOL, create the source files for both languages.

3.3 Treating Forms in Task and Task Group Definitions

Generally, task and task group definitions are not dependent on whether an application handles VT or desktop devices. However, if you are using FORM I/O or REQUEST I/O tasks, the treatment of forms in the task and task group definitions depends on whether the ACMS application runs without VT devices.

If you are using VT and desktop devices and either FORM I/O or REQUEST I/O tasks, refer in the task definitions to the real form name or names and in the task group definition to the real .FORM or .RLB file used by the ACMS application.

If you use the FORM I/O or REQUEST I/O attribute in your desktop-only task definitions (as opposed to using tasks with the NO TERMINAL USER I/O attribute), the task and task group definitions must still refer to actual files. This reference is necessary because the EXC opens these files when the ACMS application starts.

You can use names supplied by HP if you do not want to create your own:

- In the task definition, refer to ACMSDI_FORM as the form name.

- In the task group definition, refer to a form library file or a request library file:

  - SYS$LIBRARY:ACMSDI$FORM.FORM for FORM I/O tasks

  - SYS$LIBRARY:ACMSDI$RLB.RLB for REQUEST I/O tasks

The TP Desktop Connector software provides an empty but valid form library file and a request library file to which the task group definition can refer. These files are created when you install TP Desktop Connector software on an OpenVMS system.

Example 3–1 shows definitions of a task and a task group that refer to the file ACMSDI$FORM.FORM.

Example 3–1 Desktop-Only I/O Task and Task Group Definitions

```
REPLACE TASK sample_task
  WORKSPACES ARE sample_wks1, sample_wks2;
  BLOCK WORK WITH FORM I/O                    1
```

Example 3–1 (Cont.) Desktop-Only I/O Task and Task Group Definitions

```
   EXCHANGE
     TRANSCEIVE RECORD sample_rec1, sample_rec2 IN acmsdi_form
       SENDING sample_wks1                          2
       RECEIVING sample_wks2;
 .
 .
 .
  END BLOCK WORK;
END DEFINITION;

REPLACE GROUP sample_group
  USERNAME IS sample_exc
  FORM IS acmsdi_form
    IN "sys$library:acmsdi$form.form"    3
    WITH NAME acmsdi_form;
  SERVER IS
    sample_server: DCL PROCESS;
  END SERVER;
  TASK IS
    sample_task: TASK DEFINITION IS sample_task;
  END TASK;
END DEFINITION;
```

The following are the key points in Example 3–1 for a ACMS application without VT devices:

1 A task definition with FORM I/O signals the requirement for a HP DECforms form name.

2 The name ACMSDI_FORM refers to the HP DECforms form supplied with this product.

3 The task group definition shows the full specification for the HP DECforms form file supplied with this product.

3.4 Enabling Version Checking on OpenVMS Systems

If your desktop client program uses the portable client services, which support version checking, you can build a program to run on the OpenVMS system to pass version information to the desktop client program. See Section 4.3 for information on the processing support for version checking.

Your ACMSDI_GET_VERSION routine can do one of the following checks:

- Look at your ACMS application images to determine versions.

- Use a version number stored in the application database.

- Take other approaches to establish the version of the ACMS application software.

The routine returns a value in its version parameter, a 24-character string.

To do version checking, perform the following operations on the OpenVMS system:

- Link your ACMSDI_GET_VERSION routine into a shareable image (see Section 3.4.1).

- Define the system logical name ACMSDI_GET_VERSION on the TP Desktop Connector Gateway for ACMS system to point to the ACMSDI_GET_VERSION shareable image file (see Section 3.4.2).

  If the logical name is not defined or the action routine cannot be located, the TP Desktop Connector Gateway for ACMS logs a message in the SWL and continues without calling the routine.

3.4.1 Building the ACMSDI_GET_VERSION Shareable Image

To create a shareable image that the TP Desktop Connector Gateway for ACMS calls for checking versions, follow these steps:

1. Edit according to the documented interface the sample file named ACMSDI_GET_VERSION.C in the ACMSDI$EXAMPLES directory. *HP TP Desktop Connector for ACMS* Client Services Reference Manual describes the interface.

2. Compile the routine:

   ```
   $  CC ACMSDI_GET_VERSION.C
   ```

   The next few steps make the routine available to the gateway.

3. Create a linker options file to make the routine externally visible.

   The file, for example, ACMSDI_GET_VERSION.OPT, must contain at least the following line:

   ```
   UNIVERSAL=ACMSDI_GET_VERSION
   ```

4. Link the ACMSDI_GET_VERSION routine into a shareable image using the compiled routine and the options file:

   ```
   $  LINK /SHAREABLE=ACMSDI_GET_VERSION.EXE -
   _$   ACMSDI_GET_VERSION.OBJ, ACMSDI_GET_VERSION.OPT/OPTIONS
   ```

   This creates the file ACMSDI_GET_VERSION.EXE in the current default directory.

5. Place the newly created shareable image in a secure directory.

Because the gateway runs with elevated privileges, the ACMSDI_GET_
VERSION routine runs with the same elevated privileges when that
gateway calls it. Securing the image guards against a nonprivileged user
gaining unauthorized privileges.

Section 3.4.2 tells how to point the gateway to the image file.

3.4.2 Defining the Version-Checking Logical Name

The gateway uses a logical name to access the ACMSDI_GET_VERSION
routine as a shareable image (built as described in Section 3.4.1). Define
the system logical name ACMSDI_GET_VERSION to translate to the device,
directory, and file name of the image stored in the secure directory:

```
$  DEFINE /SYSTEM ACMSDI_GET_VERSION -
_$  SECURE_DEVICE:[SECURE_DIRECTORY]ACMSDI_GET_VERSION
```

The gateway tries to find and invoke the ACMSDI_GET_VERSION routine
only if a desktop client program signs in to the ACMS system and requests
version checking by specifying it as a submitter option. After the gateway
searches for the shareable image and routine the first time after its restart, the
same image and routine are used until the gateway restarts again.

3.5 Getting Desktop Submitter Information

To gather TP Desktop Connector runtime information, you can provide a
utility program to run on the same system as the gateway. ***HP TP Desktop
Connector for ACMS*** *Client Services Reference Manual* describes the system
management service to use.

A sample image (SHOW_DESKTOP_USERS.EXE), a source file (.C), a linker
options file (.OPT), and a build command procedure (.COM) are stored in the
ACMSDI$EXAMPLES directory to provide guidelines in coding and building
the utility.

3.5.1 Coding the Routine

The caller can request all current desktop submitters or can select desktop
submitters by user name, ACMS submitter identification, or desktop gateway
submitter identification. The following identifications apply to a submitter:

- Desktop gateway submitter identification

  The gateway uses this value internally.

- ACMS submitter identification

  This is the value shown in the output of the ACMS/SHOW commands.

- TP Desktop Connector submitter identification

This identification is supplied on the desktop system when the desktop client program signs the user in to the ACMS system. It never leaves the desktop and several desktop systems may have desktop client programs running with the same identification value. This identification is neither accepted by nor returned by the ACMSDI$GET_SUBMITTER_INFO service.

To filter which desktop submitters are reported, the ACMSDI$GET_SUBMITTER_INFO service applies all selection values provided on the call. If no selection values are provided, all submitters are reported.

To call the ACMSDI$GET_SUBMITTER_INFO routine, the software must perform these steps:

1. Initialize the user_context parameter to zero.

2. Enter a program loop that invokes ACMSDI$GET_SUBMITTER_INFO.

    To limit the number of submitters reported, specify one or more of the *target_submitter_ID*, *target_desktop_ID*, or *target_username* parameters. To have all submitters reported, specify none of the target parameters.

    Independent of the selection criteria, build the item list to specify what information you want reported about each submitter that matches the criteria. The service reuses the buffers you provide for the output information for each matching submitter. Have the program copy the reported information from the buffer to other program-managed storage.

3. Continue looping until ACMSDI$GET_SUBMITTER_INFO returns a status indicating that no more submitters match the selection criteria.

Because ACMSDI$GET_SUBMITTER_INFO takes a snapshot at the time of the first call, submitters can sign in to or out of the TP Desktop Connector system while the program loop calls the service repeatedly. The service always reports on a consistent set of submitters, the set that is signed in at the time of the first call. But the return status from the final call indicates whether the information reported from the snapshot is still valid as of the final call, or whether the set of submitters changed during the repeated calls.

3.5.2 Building the Shareable Image

The C-language prototype for the ACMSDI$GET_SUBMITTER_INFO routine and definitions for the item codes are included in the file ACMSDI.H in the ACMSDI$COMMON directory. Include this file in your C source file to obtain those definitions.

To link your program after you compile it, specify
ACMSDI$VMS:ACMSDI$CLIENT_OBJLIB.OLB as an input object library,
and SYS$SHARE:VAXCRTL.EXE as an input shareable image library. One
way to do this is to create a linker options file (for example, EXAMPLE.OPT)
containing these lines:

```
ACMSDI$VMS:ACMSDI$CLIENT_OBJLIB/LIB
SYS$SHARE:VAXCRTL/SHARE
```

Specify the linker options file on the link command:

```
$ LINK your-program.OBJ, EXAMPLE.OPT/OPT
```

Store the program in a protected directory.

After you install TP Desktop Connector software, the
ACMSDI$EXAMPLES directory contains an example program SHOW_
DESKTOP_USERS.EXE that uses the ACMSDI$GET_SUBMITTER_INFO
service. SHOW_DESKTOP_USERS.C and .OPT are the C-language source and
the linker options files. The command procedure BUILD_SHOW_DESKTOP_
USERS.COM includes the link command to build the image.

To modify the sample, copy the files to your own directory:

```
$ COPY ACMSDI$EXAMPLES:*SHOW_DESKTOP_USERS*.* *.*
```

Compile the program using the following command:

```
$ CC /INCLUDE=ACMSDI$COMMON: SHOW_DESKTOP_USERS.C
```

Link the resulting object into an image using this command:

```
$ @BUILD_SHOW_DESKTOP_USERS
```

This creates SHOW_DESKTOP_USERS.EXE in your directory. Run the image
using this command:

```
$ RUN SHOW_DESKTOP_USERS
 TP Desktop Connector          Desktop Node Local          Sign-in Time
Submitter     Submitter          Transport Username          Latest Message Time
  2790021    F8360;BD                63571 SMITH        8-FEB-2002 11:19:32.30
                                     DECnet             8-FEB-2002 11:19:32.30
```

3.6 Debugging TP Desktop Connector Solutions

Several approaches are available for you to debug your TP Desktop Connector solution. You can debug:

- Desktop client program only
- NO I/O tasks
- Tasks called from desktop client programs
- Procedure server code

3.6.1 Debugging the Desktop Client Program Only

During your software development, it is useful to run your desktop client program independently of the ACMS application. To be able to run the desktop client program independently, write stub routines that emulate the processing expected on a call to the ACMS application, and link your desktop client program with these routines rather than the TP Desktop Connector client services. After you debug the desktop client program, relink it with the TP Desktop Connector client services. Part II describes debugging for each type of desktop system.

3.6.2 Debugging NO I/O Tasks

With ACMS applications that use NO I/O tasks, you can debug both the tasks and the step procedures without using the desktop system or the gateway. Use the ACMS Task Debugger to completely simulate the behavior of both the desktop system and the gateway for the desktop system. This technique works only for tasks having BLOCK WORK WITH NO I/O declared.

Although the ACMS Task Debugger is not a source-level debugger, it works with the OpenVMS Debugger so the procedure server (written in any OpenVMS third-generation language) is accessed by means of the OpenVMS Debugger. When the procedure server returns to the ACMS task, the ACMS Task Debugger regains control and allows step, breakpoint, deposit, examine, and other basic Debugger functions on the ADU task definition code.

To emulate a submitter, make a command file containing deposit statements that write reasonable test case values into the workspaces used in the call to the procedure server. When the task is finally activated, arguments to the task are expressed in workspace data passed to the task. To emulate the submitting desktop system, load workspace arguments as the desktop client program does when it calls the task. The ACMS Task Debugger supports command files that are executed (@*filename* syntax) at the ACMSDBG prompt. All ACMS Task Debugger commands can be executed from a command file (including other command files).

After making the command file, debug by following these steps:

1. Start the procedure server with the START SERVER command (before the task is selected).

2. Set OpenVMS Debugger breakpoints in the procedure server.

   Set a breakpoint at *TASK-NAME\STEP-LABEL\$BEGIN>* for the task and processing step that calls the procedure server process.

3. Select the task and step to the breakpoint.

4. Execute the command file that sets up the workspace records for the test case.

5. Examine the workspaces to confirm that they are as needed.

6. Step through the procedure server.

   If you compiled and linked the procedure server with /DEBUG options, the procedure server can be debugged exactly as any third-generation code under OpenVMS. The ACMS Task Debugger steps into the procedure server under OpenVMS Debugger control.

   If the procedure server was not compiled and built with /DEBUG options, the OpenVMS Debugger steps through it.

7. Confirm that the results are as expected by examining the data passed out of the procedure server call.

When the tasks and procedure servers are behaving as expected using this technique, you can then call the NO I/O tasks from your desktop client program, using the technique described in Section 3.6.3.

3.6.3 Debugging Tasks Called from Desktop Client Programs

To debug tasks, use the ACMS Task Debugger. The debugging steps are similar to those described for debugging tasks called from a user-written agent in *Writing Server Procedures*.

TP Desktop Connector allows you to bring up multiple copies of the gateway on a single CPU, and to route a client connection explicitly to one of these copies of the gateway. This facility, provided only for debugging purposes, allows multiple developers at the same time and on the same OpenVMS system to debug tasks called from TP Desktop Connector client programs.

Note

Using multiple gateways on a single CPU is not supported for production purposes.

To use the multiple gateway capability to debug tasks called from client programs, follow these steps:

1. Define the ACMS$DEBUG_AGENT_TASK logical name in one of the logical name tables for the user name under which you will be starting a copy of the gateway. The following example assumes that you are signed in to an ACMSDI_TEST account:

   ```
   $ DEFINE/GROUP ACMS$DEBUG_AGENT_TASK "Y"
   ```

   If your ACMS system and a gateway are currently running, you must stop and restart them. The ACMS$DEBUG_AGENT_TASK logical name is recognized only by agent processes in the group for which the logical name has been defined, and which are started after you have made this assignment.

2. Create or edit your gateway startup parameter file to define values for the SERVER_NAME and DECNET_OBJECT keywords. The names can be no longer than 5 characters.

   For example:

   - DECnet

     ```
     SERVER_NAME=SMITH
     DECNET_OBJECT=187
     ```

   - TCP/IP

     ```
     SERVER_NAME=SMITH
     TCPIP_PORT=1022
     ```

   - Both: DECnet and TCP/IP

     ```
     SERVER_NAME=SMITH
     DECNET_OBJECT=187
     TCPIP_PORT=1022
     ```

   See Section 3.6.3.8 for information on using transports other than DECnet.

3. Run the ACMSDI$STARTUP.COM command procedure to start the TP Desktop Connector Gateway for ACMS, specifying the gateway startup parameter file containing your keyword definitions. For example:

   ```
   $ SUBMIT/USER=ACMSDI_TEST/NOLOG SYS$STARTUP:ACMSDI$STARTUP -
   _$ /PARAM= SYS$STARTUP:SMITH.PRM
   ```

Assuming that the keyword declarations shown in the previous example are defined in a user-created SMITH.PRM file in SYS$STARTUP, this command starts a gateway with the process name ACMSDI$SRVSMITH connected to DECnet object 187. By specifying a user name, you associate a group logical name table with this gateway that is different from the logical name table associated with a production version of the gateway.

The ACMSDI_TEST account must have NETMBX, TMPMBX, SYSNAM, SYSPRV, SYSLCK, and CMKRNL privileges. The process issuing this command must have NETMBX and CMKRNL privileges.

This command can be incorporated into an ACMS task, using a DCL process server, so that users starting and stopping the server do not need to have system privileges. See Section 3.6.3.10.

4. Start an ACMS task debugger session, specifying the /AGENT_HANDLE qualifier. Set breakpoints, and then type ACCEPT to enable the task debugger session to receive task calls through your TP Desktop Connector gateway.

For example:

```
$ SET PROCESS/PRIV=SHARE
$ ACMS/DEBUG/AGENT_HANDLE=SMITH_TEST PERS_GROUP.TDB/WORKSPACE
ACMSDBG> SET BREAK PERS_UPDATE_TASK
ACMSDBG> START PERS_SERVER
DBG> SET BREAK PERS_GET_RECORD
DBG> GO
ACMSDBG> ACCEPT/CONTINUOUS
```

Using an agent handle like SMITH_TEST (rather than PERS_GROUP) enables multiple users to work on the same task group in different task debugger sessions, at the same time. It is not necessary to create multiple versions of the task group, only to specify a unique agent handle.

You can use a logical name to associate the agent handle with the application name used by a client program. See Section 3.6.3.2 for this information.

5. On the desktop client, set the environmental variable (or logical name, for OpenVMS clients) ACMSDI_DECNET_OBJECT_*node* to the object number you specified in starting the gateway. The *node* is the DECnet node name you will be specifying as the submitter-node parameter in ACMSDI_SIGN_ IN. For example, if you are going to connect to ACMTST (in this case, using DECnet from a OpenVMS client):

```
$ DEFINE ACMSDI_DECNET_OBJECT_ACMTST 187
```

Run tasks from your client program, specifying SMITH_TEST as the application name in the ACMSDI_CALL_TASK service. The following notice appears on the terminal at which you are running the task debugger process:

```
ACMSDBG>
Terminal is in SERVER SMITH_TEST
```

See Section 3.6.3.8 for information on using transports other than DECnet.

6. When you are finished with the debugging session, exit from the task debugger and stop your gateway process:

```
ACMSDBG> EXIT
$ @SYS$STARTUP:ACMSDI$SHUTDOWN SMITH
```

You can also deassign the ACMSDI$DEBUG_AGENT_TASK logical. Or you can start another task debugger session to debug other tasks in the same or another group. Note that you must stop your gateway in order to run another task debugger session, even if you want to bring up the same task group that you were just debugging.

These steps, and additional options you have in using the multiple server feature, are explained more fully in the sections that follow.

3.6.3.1 Defining the Gateway Process and Network Names

Multiple gateways on a single OpenVMS system are distinguished from each other both by their process names and by the network name or object number they are associated with. These values, as well as resource utilization and other aspects of the gateway, are determined at gateway startup.

The default name for the gateway is ACMSDI$SERVER. If you start additional gateways on the node, TP Desktop Connector assigns them process names of the form ACMSDI$SRV(extension). The (extension) is a user-supplied suffix of up to 5 characters. This suffix distinguishes one gateway from another, and forms an OpenVMS process name, which is limited to 15 characters.

The command file SYS$STARTUP:ACMSDI$STARTUP.COM has been extended to handle the starting of multiple gateways. The naming of the gateway and whether or not it is considered the primary gateway is determined by keywords that you can add to a user-created parameter file. The keywords are SERVER_NAME, DECNET_OBJECT, and TCPIP_PORT

When invoking ACMSDI$STARTUP.COM, the optional parameter file is used to define logical names from the keywords. The logical names are defined in the process logical name table, for the process which invokes ACMSDI$STARTUP.COM. If the keywords are not specified or are specified as null, the process logical names are not defined. The keywords that control

gateway communication are listed in Table 3–1 with their associated logical names.

Table 3–1   Gateway Communication Keywords

Parameter File	Logical Name	Default	Sample
SERVER_NAME	ACMSDI$SERVER_NAME	Uses decnet_object	DBG1
DECNET_OBJECT	ACMSDI$DECNET_OBJECT	87	"TASK=DBG1"
TCPIP_PORT	ACMSDI$TCPIP_PORT	1023	1022

For example, you can include the following declarations in a gateway startup parameter file:

```
SERVER_NAME=DBG1
DECNET_OBJECT="TASK=DBG1"
```

This startup file starts a gateway with process name ACMSDI$SRVDBG1, accessible for clients that specify DBG1 as the DECnet objectname with which to connect. The object number is assigned by DECnet. The name or number specified by the DECNET_OBJECT keyword is used to select the DECnet object that a gateway uses to listen for client connections when the DECnet transport is active. The DECNET_OBJECT can name a numbered DECnet object (DECNET_OBJECT = 187) or a named DECnet object (DECNET_OBJECT = "TASK=DBG1"). If additional gateways using DECnet are started, DECNET_OBJECT must be defined to be a unique object identifier. The object name is a suffix of up to 5 characters.

Similarly, the number specified by the TCPIP_PORT keyword specifies the TCP/IP port number used for client connections. For forced nonblocking calls, you also have the ability to specify your gateway node TCP/IP port number dynamically without having to define the environmental variable. See the *HP TP Desktop Connector for ACMS* Gateway Management Guide for more information on setting values for this keyword and specifying the port number dynamically.

The name specified by the SERVER_NAME keyword is used to distinguish between multiple gateways on the same node by providing a gateway naming extension. The value is a name suffix of up to five characters, which is appended to ACMSDI$SRV to create the process name; for example, if SERVER_NAME is specified as DBG1, the gateway process name is ACMSDI$SRVDBG1.

If SERVER_NAME is not specified, then the DECNET_OBJECT, or TCPIP_ PORT name is also used to distinguish between gateways. In this case, the DECNET_OBJECT name, for example, is used as a gateway name suffix (not including "TASK="). If neither SERVER_NAME nor DECNET_OBJECT is defined, then the default process name of ACMSDI$SERVER and DECnet object 87 is used.

Note

A single client process can start multiple sessions to the same gateway on one node or multiple sessions to individual gateways on different nodes. However, a single client process cannot start second session to an alternate gateway on a node to which it already has a gateway connection established. That is because the client services cannot distinguish the difference between different gateways on the same node and multiplex subsequent sessions over the one connection to the initial gateway. However, you can have multiple client programs active at the same time on a desktop system and connected to different gateways on a single node.

3.6.3.2 Defining Logical Names

Before starting the gateway, you must define the logical name, ACMS$DEBUG_ AGENT_TASK, in a logical name table associated with that process. For example, if you are in the same group as the user name under which the gateway will be started, you can use this command:

```
$ DEFINE/PROC ACMS$DEBUG_AGENT_TASK "TRUE"
```

The ACMS$DEBUG_AGENT_TASK logical name instructs ACMS to direct task selection requests to task debugger processes rather than to ACMS EXC processes. If this logical name has been set for a gateway or another ACMS agent, any task selection handled by that gateway or agent which specifies an application name for which there is no corresponding task debugger session fails with the "application not found" message.

This logical name affects only those gateways or agents for which it has been defined. You can, therefore, have both production and debug systems on the same OpenVMS node, as long as the ACMS$DEBUG_AGENT_TASK logical has not been defined for the ACMSDI$SERVER process.

The relationship between the application name specified by the client program and the task debugger process which will receive the task calls can be established in either of two ways:

- The application name specified in task call is the same as the string specified in the /AGENT_HANDLE qualifier for the ACMS/DEBUG command.

- The application name specified in task call is the same as a logical name in one of the gateway's logical name tables, whose equivalence name is the string specified in the /AGENT_HANDLE qualifier for ACMS/DEBUG.

For example, if the current process is in the same group as the name under which your gateway is started, you can use this command to set up a logical name for the application:

```
$ DEFINE/GROUP PERS_APPL SMITH_TEST
```

In this example, the client program calls tasks in an application named PERS_APPL. The task debugger session, however, is started with the agent handle set to SMITH_TEST, as shown in Figure 3–1.

Figure 3–1 Task Debugger Session
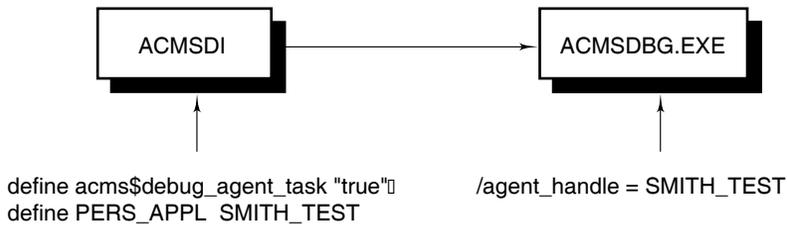

```
ACMSDI                                    ACMSDBG.EXE


define acms$debug_agent_task "true"       /agent_handle = SMITH_TEST
define PERS_APPL  SMITH_TEST

                                          TAY-0183-AD
```

With this approach, the application names in the client program need not change to identify unique task debugger sessions for different users.

The logical name relationship between the application name and the agent handle can be set up by TP Desktop Connector for your gateway process if you include the DEBUG_APPLICATION_NAME and DEBUG_AGENT_HANDLE keywords in your startup parameter file. For example:

```
DEBUG_APPLICATION_NAME = PERS_APPL
DEBUG_AGENT_HANDLE     = SMITH_TEST
```

The equated value of the DEBUG_APPLICATION_NAME keyword is used as the logical name. The equated value of the DEBUG_AGENT HANDLE keyword is used as the equivalence name. In addition, if you use these keywords, TP Desktop Connector also sets the ACMS$DEBUG_AGENT_ TASK logical to "TRUE". In this way, you do not need to have GROUP or other privileges to define these logical names.

3.6.3.3 Required Privileges

Several privileges are required for the user name under which a gateway is started, regardless of whether named or numbered DECnet objects are used. These privileges are:

NETMBX
TMPMBX
SYSNAM
SYSPRV
SYSLCK
CMKRNL

When starting the gateway, you can either invoke SYS$STARTUP:ACMSDI$STARTUP.COM directly, or you can use the SUBMIT command to start the gateway under a user name other than your own.

To invoke ACMSDI$STARTUP.COM directly, the process invoking the command procedure must have all six privileges in addition to the DETACH and OPER privileges.

To invoke ACMSDI$STARTUP.COM with the SUBMIT/USER command, the process issuing the SUBMIT command must have NETMBX and CMKRNL privileges. It does not have to have the other privileges listed in this section. However, the user name specified in the /USER qualifier must have all six privileges.

3.6.3.4 Installing the TP Desktop Connector Gateway for ACMS Images

If you have many users who are starting gateways, you can install the gateway images to reduce the image activation and memory utilization costs for running the gateway. Create a command file (for example, SYS$STARTUP:ACMSDI$INSTALL.COM), which you invoke from your system startup file, containing the following commands:

```
$ INSTALL ADD/OPEN/SHARE SYS$SYSTEM:ACMSDI$SERVER
$ INSTALL ADD/OPEN/SHARE SYS$SHARE:ACMSDI$SERVER_SHR
```

3.6.3.5 Troubleshooting Problems in Starting Multiple Gateways

If you receive an error in starting the gateway, check the Software Event Log for additional information on why the gateway could not be started. Conditions that can cause the gateway startup to fail include:

- Another gateway of the same name or with the same DECnet name or object number is already started.

- Insufficient system resources exist to start the gateway.

- User name quota values are insufficient to start the gateway.

- The process that is starting the gateway or the user name under which the gateway is being started do not have sufficient privileges.

3.6.3.6 Starting the TP Desktop Connector Gateway for ACMS from an ACMS Task

To minimize the privileges assigned to the individuals who may need to stop and start the gateway, create an ACMS task that users can invoke to start the gateway. For example:

```
REPLACE TASK START_SVR_TASK
BLOCK
  PROCESSING IS
    DCL COMMAND "SUBMIT/USER='P1 @SYS$STARTUP:ACMSDI$STARTUP.COM 'P2"
      IN START_SVR_SERVER;
END BLOCK;
END DEFINITION;
```

Define ACMSDI_START_SERVER in the task group as a DCL PROCESS server. In the application, assign to the gateway a user name that has NETMBX and CMKRNL privileges so that it can issue the SUBMIT/USER command; define the ACL for this task to allow access by any appropriate users. The users can then use the SELECT command from any ACMS menu to invoke this function:

```
Selection: SELECT start_svr_appl start_svr_task acmsdi_test test:smith.prm
```

In this example, acmsdi_test is the user name under which the gateway is started. SMITH.PRM is the name of the parameter file in the directory pointed to by the logical name TEST.

3.6.3.7 Running the Task Debugger Session

A process running the ACMS Task Debugger (ACMSDBG) is dedicated to debugging a single ACMS task group. Users use the ACMS/DEBUG to invoke the ACMSDBG.EXE image in their process. The procedure servers that the ACMS Task Debugger starts are run in subprocesses associated with the user process.

To debug a task, the task group definition must be built with the /DEBUG qualifier:

```
ADU> BUILD GROUP/DEBUG dbg_task_group
```

If the task group is not built with /DEBUG, you can start the task group in the task debugger, set breakpoints, and run tasks. However, you cannot examine or deposit workspace values.

Compile and link the step procedures with the /DEBUG qualifier. Otherwise, you cannot set breakpoints, or examine and deposit variables.

Before starting the task debugger, define any logical names you need for your procedure servers. These include any names you would normally include in the application definition SERVER LOGICALS clause, or that you would define in logical name tables accessible to the procedure servers.

To start the ACMS Task Debugger, use ACMS/DEBUG:

- Include the name of the .TDB task group database file containing the tasks to be run. Only one task group can be debugged at a time.

- Use the /AGENT_HANDLE qualifier to provide the ACMS Task Debugger with a unique handle to the agent that will be submitting the tasks.

- Use the /WORKSPACE qualifier to enable examining and depositing in workspaces.

For example:

```
$ ACMS/DEBUG/WORK/AGENT_HANDLE=SMITH_TEST PERS_GROUP
ACMSDBG> SET BREAK PERS_UPDATE_TASK
ACMSDBG> ACCEPT/CONTINUOUS
```

SMITH_TEST is the agent handle assigned for this task debugger process. PERS_GROUP.TDB is the name of the task definition file that you are debugging. SET BREAK establishes a breakpoint at the start of PERS_UPDATE_TASK. ACCEPT/CONTINUOUS prepares the task debugger process to receive task calls from the gateway. At this point, your process waits to receive either a task call, Ctrl/C, or Ctrl/Y.

Make sure that the gateway from which you will be accepting task calls is on the same OpenVMS system as your task debugger process. With the task debugger, you cannot use the application routing capability that is available when you are calling tasks in the normal ACMS runtime system.

The following restrictions apply when running a task debugger session with /AGENT_HANDLE specified:

- To run tasks in this debugger session, you must invoke them from a client program through the gateway, or from another ACMS agent. Do not select tasks from within the task debugger (that is, by using the SELECT command).

- If you want to end a task, do so from the client program rather than by using the CANCEL command in the task debugger.

- Make sure that you run only one task at a time in the task debugger session. If you need to test several tasks at one time (in multiple sign-ins) from the client program, set up a task debugger session for each of the sign-ins that you are going to create.

As you complete a task, you can start another task without leaving the task debugger. You can stop and start procedure servers within that task debugger session. However, once you exit from that task debugger session, you must also stop the gateway associated with that session. If you are going to do further debugging, restart your gateway, then restart your task debugger session. If you are not going to do any further debugging, also deassign the ACMS$DEBUG_AGENT_TASK and debug application logical name (if any) associated with your task debugger session.

3.6.3.8 Selecting a Gateway from a Portable API Client Program

To establish a connection between a client program and a gateway other than ACMSDI$SERVER (object 87), you must use the logical names (on OpenVMS) or the environmental variable (on Windows or Tru64 UNIX) ACMSDI_DECNET_OBJECT_*node* to define the gateway you want to invoke. The *node* in this case is the DECnet node name of the system to which you want to connect.

You can define one of two equivalence values for the ACMSDI_DECNET_OBJECT_node logical name or environmental variable:

- *object_number*

  Specifies a DECnet object number, in the range 128 to 255. For example:

  ```
  > SET ACMSDI_DECNET_OBJECT_ACMTST 187
  ```

- *"task-name"*

  Specifies the DECnet object name, up to 5 characters. For example:

  ```
  > SET ACMSDI_DECNET_OBJECT_ACMTST "SMITH"
  ```

Similarly, for transports other than DECnet, you define logical names or environmental variables that specify the port to which you want to connect. For TCP/IP, define ACMSDI_TCPIP_PORT_*host* to the same number that you assigned to the TCPIP_PORT keyword in the gateway startup parameter file. For example (for Tru64 UNIX):

```
# setenv ACMSDI_TCPIP_PORT_host 1023
```

You can start multiple sessions from a single client program. However, to run a task in more than one of those sessions at one time, you must have a separate task debugger session started for each of the tasks you want to run simultaneously.

3.6.3.9 Managing TP Desktop Connector Gateways Used for Debugging Purposes

All gateway capabilities, such as ACMSDI$GET_SUBMITTER_INFO, SHOW_DESKTOP_USERS, and Oracle Trace support, are available with multiple gateways.

To use ACMSDI$GET_SUBMITTER_INFO, you must define ACMSDI_DECNET_OBJECT_*node* on the OpenVMS node on which you are issuing the ACMSDI$GET_SUBMITTER_INFO service. For example:

```
$ DEFINE/GROUP ACMSDI_DECNET_OBJECT_0 187
```

Use 0 rather than the DECnet node name when defining this logical name. The definition must be in a logical name table visible to the gateway whose number or name you specify as the equivalence name. Only the information from that gateway is returned.

3.6.3.10 Stopping a TP Desktop Connector Gateway for ACMS

The command procedure SYS$STARTUP:ACMSDI$SHUTDOWN takes an optional parameter value to specify a particular gateway process to stop. Use the process name as the parameter value. For example:

```
$ @SYS$STARTUP:ACMSDI$SHUTDOWN DBG1
```

If you do not specify the parameter when invoking ACMSDI$SHUTDOWN.COM, ACMSDI$SERVER is stopped.

3.6.3.11 Restrictions on Using Multiple Gateways

The following restrictions apply when using the multiple gateway feature:

- Multiple gateways are supported for use in debugging only. For production systems, there must be only one gateway per OpenVMS system; that gateway must run under the default ACMSDI$SERVER user name, using the DECnet object 87 that HP has assigned for the gateway.

- TCP/IP has a guaranteed port number of decimal 1023.

Note

Check that the numbers you choose for DECnet object numbers and TCP/IP port numbers are not taken by another application on your system.

- The ACMSDI gateway must be running on the same node as the ACMS task debugger process that will be invoked by it on a task call.

- Only a single submitter can use an ACMSDI gateway started with ACMS$DEBUG_AGENT_TASK enabled. When the ACMSDI$SERVER image is started with ACMS$DEBUG_AGENT_TASK, it is identified as an agent to be used with an ACMS task debugger process. When the gateway submits a task in an application whose translated name matches a unique agent handle, that gateway become associated with that particular ACMSDBG.EXE; the gateway is not accessible to any other task debugger process.

- After the ACMSDBG debugging session has been terminated, the ACMSDI gateway associated with that session cannot be used for other users. Nor can it be used for another task debugger session started by the same user. The ACMSDI gateway associated with the terminated ACMSDBG debugging session must be stopped; a gateway using the same name can then be restarted for another debugging session by the same or a different user.

- An ACMSDBG debugging session is restricted to a single task group. If the tasks called by a client program are normally in one application (.ADB file), but that application contains several task groups (.TDB file), then the task debugger session and TP Desktop Connector gateway must be stopped and restarted for each of the task groups.

3.6.4 Debugging Procedure Server Code

An online server debugging technique provided on the ACMS system lets you place procedure servers running on line (that is, outside the ACMS Task Debugger context and in the ACMS production context) into the OpenVMS Debugger. However, this technique has no ACMS task-level debugging. The OpenVMS Debugger is invoked when the procedure server actually starts. See the ACMS documentation for information on online server debugging.

Part II

Portable API Client Development

Part II describes procedures to develop desktop client programs using the TP Desktop Connector portable API. General design and development issues are discussed in Part I. Refer to **HP TP Desktop Connector for ACMS** *Client Services Reference Manual* for reference information about each desktop system.

4

Developing Portable API Client Programs

This chapter provides guidelines for developing desktop client programs for the portable application programming interface (API). Examples in this chapter illustrate the use of the blocking form of the TP Desktop Connector client services.

For information on configuring network transports for any of the client platforms, refer to the **HP TP Desktop Connector for ACMS** *Gateway Management Guide*.

4.1 Guideline Summary

The following sections summarize the guidelines for developing desktop programs using the portable API client services.

4.1.1 Managing Code on the Desktop Client System

To effectively manage the relationship between elements of the desktop client program and the HP ACMS application for I/O tasks, create and use a library of application-specific presentation procedures that equate one-to-one to exchange steps. (This is the approach used in the sample application in Section 4.4.)

4.1.2 Structuring Exchange Steps in the Presentation Code

If you are using I/O tasks in your solution, your desktop client program includes the following components:

- Main program

  Contains procedures for signing the user in to the ACMS system, handling menus and task selections, and signing the user out of the ACMS system.

- Presentation procedures

  Contain presentation code for handling interactions with a user to display and gather data during exchange step execution in an I/O task.

To handle the terminal I/O in an exchange step, a ACMS application typically calls HP DECforms or TDMS services. However, when using the TP Desktop Connector software, those calls are made to the desktop client program. Linked into the desktop client program, customer-written routines provide the logic to perform the exchange step processing on the desktop system.

TP Desktop software defines the interface for presentation procedures, but the application developer writes the actual code for the presentation procedures. Customer-written **presentation procedures** are called by TP Desktop Connector client services to handle ACMS exchange steps.

The desktop client program can use any appropriate forms-processing or presentation tool to interact with the user. Because procedures to handle task selection can also gather data from the user for input to the task, the distinction between the desktop client program and presentation procedures is somewhat arbitrary. However, it is a significant distinction.

Presentation procedures run on behalf of exchange steps in the task on the ACMS application node. The main part of the desktop client program requests a task that the TP Desktop Connector Gateway for ACMS handles. When the task executes an exchange step, the TP Desktop Connector Gateway for ACMS calls the desktop client program using the same network link. The TP Desktop Connector client services on the desktop system call the presentation procedure in the desktop client program. The appropriate customer-written presentation procedure in the desktop client program executes.

If the TP Desktop Connector software cannot invoke the presentation procedure, it returns a fatal status to the desktop gateway. The fatal status is returned to EXC, which decides whether to cancel the task based on the task definition.

Figure 4–1 shows the execution flow to process presentation procedures.

Figure 4–1 Processing of Presentation Procedures
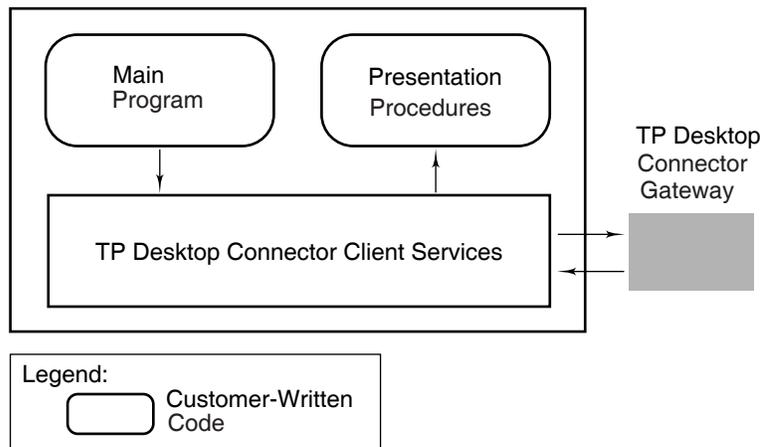
TAY-1001A-AD

4.1.3 Data Conversion

Because TP Desktop Connector software does not manipulate workspace contents, it does not constrain what data types are used on the desktop system or in the ACMS application. However, TP Desktop Connector software does not automatically handle data conversion between the OpenVMS and desktop systems. Windows data types are readily mapped to OpenVMS data types, and OpenVMS to Windows.

ACMS applications are constrained in their workspace definitions by the data types supported by CDD software. Table 4–1 shows Microsoft C and Microsoft COBOL equivalents for some of the more commonly used CDD data types.

Table 4–1 Language and CDD Data-Type Equivalents

CDD	Microsoft C	Microsoft COBOL
DATE		
NUMERIC SIZE n		$9(n)$
SIGNED BYTE	signed char	
SIGNED LONGWORD	signed long	S9(9) COMP-5
SIGNED LONGWORD SCALE -2		S9(7)V9(2) COMP-5
		(continued on next page)

Table 4–1 (Cont.)   Language and CDD Data-Type Equivalents

CDD	Microsoft C	Microsoft COBOL
SIGNED QUADWORD		S9(18) COMP-5
SIGNED QUADWORD SCALE -2		S9(16)V9(2) COMP-5
SIGNED WORD	signed short	S9(4) COMP-5
SIGNED WORD SCALE -2		S9(2)V9(2) COMP-5
TEXT SIZE n	char[n]	X(n)
UNSIGNED BYTE	unsigned char	
UNSIGNED LONGWORD	unsigned long	9(9) COMP-5
UNSIGNED QUADWORD		9(18) COMP-5
UNSIGNED WORD	unsigned short	9(4) COMP-5

Note that certain CDD data types have no direct equivalents for some languages. For example, SIGNED WORD SCALE –2 has no direct equivalent in the Microsoft C language. However, the CDD data type can be supported indirectly; for example, SIGNED WORD SCALE –2 can be supported in Microsoft C software by using a signed short and dividing it by 100 before it is displayed to the user.

The DATE data type has no direct equivalent in desktop languages. You can allocate the correct length storage for the DATE data type by using a quadword equivalent in the desktop language. However, routines for converting the DATE data type from binary to display format are not available on desktop systems. Therefore, convert a date field to a text field for the workspace using the 10-character international date format (*YYYY MM DD*) that HP DECforms and Rdb software support.

Refer to *Using CDD/Repository on VMS Systems* and the reference manual for the desktop language you are using for more information on data-type compatibility.

4.1.4 Preventing Concurrent Use

TP Desktop Connector portable API client services are *serialized* (nonreentrant) portable API client services do not allow the concurrent execution of two or more calls from a given submitter or on a given gateway connection. TP Desktop Connector prevents concurrent calls by rejecting calls made to the portable API client services while the API is actively executing instructions from the calling submitter or on the target connection from a different submitter. TP Desktop Connector does not reject calls if the API is

waiting for a reply from the gateway, thereby, allowing nonblocking execution to proceed.

To avoid serialization violations, do not issue TP Desktop Connector calls from TP Desktop Connector presentation procedures, completion routines, or any asynchronous routines (for example, ASTs or UNIX signals). The only exception is the acmsdi_complete_pp call, which you must issue from the presentation procedure in nonblocking environments.

Table 4–2 lists the status codes returned to the caller when TP Desktop Connector rejects a call due to a serialization violation.

Table 4–2 Status Codes Returned Due to Serialization Violations

Status	Call Type Currently Executing
ACMSDI_CALLACTV	acmsdi_call_task
ACMSDI_CANCELACTV	acmsdi_cancel
ACMSDI_DISPATCHACTV	acmsdi_dispatch_message
ACMSDI_SIGNINACTV	acmsdi_sign_in
ACMSDI_SIGNOUTACTV	acmsdi_sign_out

4.2 Generating Workspaces for the Client

TP Desktop Connector provides the command file MAKE_RECORDS.COM, which generates workspaces for the C and COBOL environments.

4.2.1 Generating Workspace Definitions

The MAKE_RECORDS.COM utility uses CDD to generate the record definitions of ACMS workspaces needed by client applications. This utility can generate record definitions of workspaces for:

- C client applications
- COBOL client applications

When you define workspaces in CDD, you can share their definitions between the ACMS application and the client program. Using the CDD-generated record definitions, the chances of mismatches between the ACMS application workspace definitions and the client program workspace definitions are greatly reduced.

4.2.2 Using the MAKE_RECORDS.COM Utility

The MAKE_RECORDS utility is run on OpenVMS to generate either C files, COBOL files, or both, containing the workspace definitions. These files can then be transferred down to the desktop system and incorporated into the TP Desktop Connector client program.

The MAKE_RECORDS utility presumes that you have stored your ACMS workspace definitions in a specific CDD directory. For example, Figure 4–2 shows the CDD directory structure for the AVERTZ sample application.

Figure 4–2   CDD Directory Structure
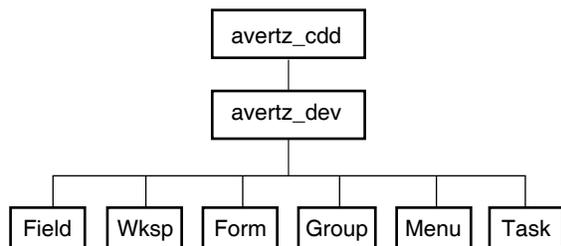
TAY-0184-AD

The definitions for each field of every AVERTZ workspace are defined in the Field directory. All of the AVERTZ workspaces are then defined (using the field definitions) in the Wksp directory.

To generate the C structure definitions for all the AVERTZ workspaces, invoke the following command (on OpenVMS):

```
$ @MAKE_RECORDS avertz_cdd:avertz_dev.wksp  C
```

where:

- *avertz_cdd* is a logical name pointing to the specific Oracle CDD dictionary.
- *avertz_dev.wksp* is the Oracle CDD directory containing the workspaces.
- C specifies the C syntax for the record definition.

For each workspace defined in the *avertz_dev.wksp* directory, a C header file is generated in the current directory.

To generate COBOL record definitions for all of the AVERTZ workspaces, invoke the following command:

```
$ @MAKE_RECORDS avertz_cdd:avertz_dev.wksp  COBOL
```

where:

- *avertz_cdd* is a logical name pointing to the specific Oracle CDD dictionary.
- *avertz_dev.wksp* is the Oracle CDD directory containing the workspaces.
- COBOL specifies the COBOL syntax for the record definition.

For each workspace defined in the avertz_dev.wksp directory, a COBOL file is generated in the current directory.

4.2.3 Generating Individual Workspace Definitions

You can also generate a record definition for a specific workspace using MAKE_ H.COM or MAKE_COBOL.COM.

As an example, one of the AVERTZ workspaces defined in CDD is VR_SITES_ WKSP. This workspace's record definition is located in the CDD directory, *avertz_cdd:avertz_dev.wksp*.

To generate a C header file for VR_SITES_WKSP, specify the following:

```
$ DEFINE CDD$DEFAULT avertz_cdd:avertz_dev.wksp
$ @MAKE_H vr_sites_wksp
```

where:

- *avertz_cdd* is a logical name pointing to the specific CDD dictionary
- *avertz_dev.wksp* is the CDD directory containing the workspace record
- *vr_sites_wksp* is the record name

This definition generates VR_SITES_WKSP.H in the current directory.

To generate a COBOL file for VR_SITES_WKSP, specify the following:

```
$ DEFINE CDD$DEFAULT avertz_cdd:avertz_dev.wksp
$ @MAKE_CBL vr_sites_wksp
```

where:

- *avertz_cdd* is a logical name pointing to the specific CDD dictionary
- *avertz_dev.wksp* is the CDD directory containing the workspace record
- *vr_sites_wksp* is the record name

This definition generates VR_SITES_WKSP.CBL in the current directory.

4.2.4 Coding Workspace Fields

If workspaces are manipulated by C in the desktop client program, some
conversions are necessary. These conversions are the responsibility of the
desktop client program.

4.2.4.1 Initializing Workspaces from ACMS

Character strings in workspaces coming from the TP Desktop Connector
Gateway for ACMS do not have a NULL terminator. If you use regular
C functions in the desktop client program to manipulate the strings in
the workspace, you need to add NULL terminators. See the examples in
the AVERTZ sample applications, particularly the convert_* functions in
wkspaces.c and m_wkspaces.c. (An alternative is to use functions such as
memset and memcpy that do not expect the NULL terminator.)

4.2.4.2 Initializing Workspaces to ACMS

Character strings in workspaces going back to the TP Desktop Connector
Gateway for ACMS must have NULL terminators removed and be padded with
spaces. The desktop client program should initialize these workspace fields to
their expected initial values, for example, zeros or spaces. Initialization can be
done using one of the following methods:

- Explicitly initialize each field in each workspace.

  If the desktop client program is written in C, use the memset function. If
  using COBOL, then the strings probably are not NULL terminated. If the
  desktop client program is written in COBOL, use the MOVE statement.

- Keep a copy of each workspace containing fields preset to their initial
  values.

  At the beginning of the workspace processing, use the C memcpy function
  or the COBOL MOVE statement to copy each field to its corresponding
  workspace.

Avoid using the C strcpy function to initialize workspaces or set values of fields
in workspaces. If you do use strcpy, make sure that the NULL terminator
is replaced with a blank character before the workspace is sent back. See
examples in the AVERTZ sample (the init_* functions in wkspaces.c or m_
wkspaces.c). In most cases, the ACMS application is not expecting NULL-
terminated strings. Use the C memset and memcpy functions to set the values
of workspaces.

4.2.5 Using Data Compression with the Portable API

TP Desktop Connector provides the call option, ACMSDI_CALL_OPT_
COMPRESS_WKSPS, to activate data compression for task calls. This call
option specifies whether or not to apply data compression to workspaces. The
following sections describe how to activate data compression. In addition, TP
Desktop Connector also provides a Data Compression Monitor that allows you
to gather statistics on the effectiveness of the compression. See Section 4.2.6
for a description of this facility.

4.2.5.1 Activating Data Compression

The ACMSDI.H include file contains the ACMSDI_CALL_OPTION_TYPE
enumeration. The call option type is included in this enumeration to support
the activation of data compression for a task call. The symbol for the call
option is **ACMSDI_CALL_OPT_COMPRESS_WKSPS**. Example 4–1 shows
the ACMSDI_CALL_OPTION_TYPE enumeration as it appears in ACMSDI.H:

Example 4–1 Compression Call Option Type

```
/*
** Call task options
*/
typedef enum {
    ACMSDI_CALL_OPT_END_LIST = 0,    /* end the options list */
    ACMSDI_CALL_OPT_OPTIMIZE_WKSPS, /* optimize network traffic on    */
                                     /* passed workspaces             */
    ACMSDI_CALL_OPT_ENABLE,          /* Pointer to enable function    */
    ACMSDI_CALL_OPT_DISABLE,         /* Pointer to disable function   */
    ACMSDI_CALL_OPT_SEND,            /* Pointer to send function      */
    ACMSDI_CALL_OPT_RECEIVE,         /* Pointer to receive function   */
    ACMSDI_CALL_OPT_TRANSCEIVE,      /* Pointer to transceive function */
    ACMSDI_CALL_OPT_REQUEST,         /* Pointer to request function   */
    ACMSDI_CALL_OPT_CHECK_VERSION,   /* Version checking routine       */
    ACMSDI_CALL_OPT_PASS_TID,        /* TID of distributed transaction */
    ACMSDI_CALL_OPT_COMPRESS_WKSPS  /* activate workspace compression */
} ACMSDI_CALL_OPTION_TYPE;
```

Prior to issuing acmsdi_call_task, the application program activates data
compression by declaring an ACMSDI_CALL_OPTION array and initializing
one of the array's elements with the call option type as shown in the following
code example:

```
ACMSDI_CALL_OPTION call_option[2];   /* call option array */

    call_option[0].option = ACMSDI_CALL_OPT_COMPRESS_WKSPS;
    call_option[1].option = ACMSDI_CALL_OPT_END_LIST;
```

The ACMSDI_CALL_OPTION array is passed as an argument on *acmsdi_call_ task*.

4.2.5.2 Specifying Data Compression for Workspaces

If you do not use optimized (unidirectional) workspaces for the task for which compression is active, data compression is attempted for all workspaces associated with the task call. In this case, the ACMSDI_WORSKPACE structure is used to describe each workspace.

When data compression is activated for a call using optimized workspaces, you can specify compression for individual workspaces with the ACMSDI_INIT_ WORKSPACE_OPT macro. This macro initializes an ACMSDI_WORKSPACE_ OPT structure. When data compression is activated, use one ACMSDI_ WORKSPACE_OPT structure to describe each workspace associated with the task call; do not use the ACMSDI_WORKSPACE structure if you want to specify compression for individual workspaces.

The following example shows the ACMSDI_WORKSPACE_OPT structure as defined in ACMSDI.H:

```
typedef struct {
    unsigned int length;
    ACMSDI_ACCESS_TYPE access;
    void *data;
} ACMSDI_WORKSPACE_OPT;
```

The data type ACMSDI_ACCESS_TYPE is defined as a single byte. ACMSDI_ ACCESS_TYPE can have one of six values as defined in ACMSDI.H:

```
#define ACMSDI_ACCESS_READ    '1'   /* read-only access */
#define ACMSDI_ACCESS_WRITE   '2'   /* write-only access */
#define ACMSDI_ACCESS_MODIFY  '3'   /* modify (read and write) */
#define ACMSDI_ACCESS_READ_COMPRESS '4'   /* read-only access with compression*/
#define ACMSDI_ACCESS_WRITE_COMPRESS '5'  /* write-only access with compression*/
#define ACMSDI_ACCESS_MODIFY_COMPRESS '6' /*modify (read and write) with compression*/
   .
   .
   .
typedef char ACMSDI_ACCESS_TYPE;
```

Access types 4, 5, and 6 support data compression.

These values specify that TP Desktop Connector client services routines and/or ACMS are to attempt to compress the workspace. Table 4–3 illustrates the values which can apply to data elements whose type is ACMSDI_ACCESS_TYPE:

Table 4–3   Portable API Access Types

Value	Specification	Description
1	ACMSDI_ACCESS_READ	read-only access
2	ACMSDI_ACCESS_WRITE	write-only access
3	ACMSDI_ACCESS_MODIFY	modify access
4	ACMSDI_ACCESS_READ_COMPRESS	read-only/compress
5	ACMSDI_ACCESS_WRITE_COMPRESS	write-only/compress
6	ACMSDI_ACCESS_MODIFY_COMPRESS	modify/compress

Using the ACMSDI_INIT_WORKSPACE_OPT macro, you initialize the ACMSDI_WORKSPACE_OPT structure for each workspace before issuing acmsdi_call_task. Then pass the ACMSDI_WORKSPACE_OPT array as an argument on acmsdi_call_task.

Example 4–2 illustrates a task call passing four workspaces. The access types for the workspaces are as follows:

- control_wksp — read-only
- empl_updates — read-only & compressed
- empl_record — write-only & compressed
- dept_record — modifiable & compressed

Example 4–2 Portable API Task Call Passing Four Workspaces

```
struct {
    char ctrl_key[5];
    char error_msg[80];
} control_wksp;

struct {
    int employee_number;
    char address_line_1[40];
    char address_line_2[40];
    char address_line_3[40];
    int insurance_code;
} empl_updates;

struct {
    int employee_number;
    int dept_number;
    char first_name[20];
    char last_name[25];
    char address_line_1[40];
    char address_line_2[40];
    char address_line_3[40];
    int number_dependents;
    int insurance_code;
} empl_record;

struct {
    int dept_number;
    char dept_name[40];
    char manager[25];
    int number_employees;
} dept_record;

ACMSDI_CALL_OPTION call_option[3];  /* call option array */
ACMSDI_WORKSPACE_OPT wksp_array[4]; /* workspace array */

    call_option[0].option = ACMSDI_CALL_OPT_OPTIMIZE_WKSPS;
    call_option[1].option = ACMSDI_CALL_OPT_COMPRESS_WKSPS;
    call_option[2].option = ACMSDI_CALL_OPT_END_LIST;
    ACMSDI_INIT_WORKSPACE_OPT (wksp_array[0],
                               control_wksp,
                               ACMSDI_ACCESS_READ);
    ACMSDI_INIT_WORKSPACE_OPT (wksp_array[1],
                               empl_updates,
                               ACMSDI_ACCESS_READ_COMPRESS);
```

Example 4–2 (Cont.)   Portable API Task Call Passing Four Workspaces

```
ACMSDI_INIT_WORKSPACE_OPT (wksp_array[2],
                           empl_record,
                           ACMSDI_ACCESS_WRITE_COMPRESS);
ACMSDI_INIT_WORKSPACE_OPT (wksp_array[3],
                           dept_record,
                           ACMSDI_ACCESS_MODIFY_COMPRESS);
acmsdi_call_task (&sub_id,
                  &call_option,  /* call option array */
                  "MYTASK",
                  "MYAPPL",
                  "",
                  stat_msg,
                  4,             /* number of workspaces */
                  &wksp_array,   /* workspace array */
                  &call_id,
                  0,0,0);
```

4.2.6 Data Compression Monitor

The Data Compression Monitor provides you with statistics about the effectiveness of TP Desktop Connector data compression. When the monitor is activated, information on each task call and each exchange step callback is gathered in a monitor log file from which you can generate a variety of reports.

You can request reports with summaries for any combination of the following entities:

- Submitter node

- User

- ACMS application

- ACMS task

The following information is captured for each task call and for each exchange step callback:

- Date and time that the call or the callback occurred

- Submitting node identifier

- Signed-in user identifier

- ACMS application name

- ACMS task name

- Call type (task call or exchange step callback type)

- Number of workspaces
- For each workspace:
  - — Access code (READ, WRITE or MODIFY)
  - — Compression code (Compressed, Uncompressable, or No Attempt to Compress)
  - — Uncompressed length
  - — Compressed length

4.2.6.1 Activating and Deactivating Compression Monitoring

To activate compression monitoring, on the node where the TP Desktop Connector Gateway for ACMS is running, define the following logical name:

```
$ DEFINE/SYSTEM ACMSDI$COMPRESSION_STATS Y
```

Note

You do not need to bring down the TP Desktop Connector Gateway for ACMS before defining this logical name.

When you define the logical ACMSDI$COMPRESSION_STATS with a value of Y or y, compression data is written to the compression monitor log file, ACMSDI$COMPRESSION.LOG. This file is written to the SYS$ERRORLOG: directory. As each ACMS task call is initiated on the server node, the gateway checks to see if the ACMSDI$COMPRESSION_STATS logical is defined. If the logical is defined and if its value is Y or y, data is gathered for the task call and for any exchange step callbacks that may be associated with the execution of the task.

To deactivate compression monitoring, redefine the logical name, ACMSDI$COMPRESSION_STATS, with a value of N or n as follows:

```
$ DEFINE/SYSTEM ACMSDI$COMPRESSION_STATS N
```

Or, you can deassign the logical name as follows:

```
$ DEASSIGN/SYSTEM ACMSDI$COMPRESSION_STATS
```

Either method causes the gateway to stop gathering compression data for tasks as they are initiated. However, compression data continues to be gathered for any task that is executing at the time of deactivation until that task has finished executing.

4.2.6.2 Creating Compression Activity Reports

To create compression activity reports, execute the Data Compression Monitor (DCM) activity reporting program by entering the following at the DCL prompt:

```
$ MCR ACMSDI$DCM
```

Note

DCM writes the report to the SYS$OUTPUT device.

DCM responds with the following prompt:

```
DCM>
```

You can select reports for all activities, or for an activity that occurred within a given timeframe. In addition, you can select reports for any combination of the following:

- Submitter node
- User ID
- ACMS application
- ACMS task

For example, to request a report for all activities from node "MYPC", running ACMS task "MYACMSTASK", from 15-SEP-2001 at 12 noon through 16-SEP-2001 at 11:45 AM, enter the following:

```
DCM> LIST /SINCE=15-SEP-2001:12:00 /BEFORE=16-SEP-2001:11:45 /NODE=MYPC -
/TASK=MYACMSTASK
```

This report shows details of all task calls for the selected task from the selected node in the given timeframe, sorted chronologically, with details for each workspace. The report also contains a summary showing the total number of bytes in the uncompressed workspaces, their compressed sizes, and the number of bytes by which the workspace sizes were reduced.

If you need only the summary information, you can add the /SUMMARY qualifier to the request. You can also direct reports to a file (as opposed to being displayed on your screen) with the /OUTPUT qualifier.

See the **HP TP Desktop Connector for ACMS** *Client Services Reference Manual* for a description of the LIST command.

4.2.6.3 Creating Customized Reports

DCM provides only one report type. However, by using the SELECT command, you can select combinations of records from the compression monitor log file and write them to a different file. You can then create a program that sorts, summarizes, and displays these records in any way you require. For example, you may want a report sorted by task name, instead of chronologically as the standard reports provide, or a report that contains only tasks associated with a given ACMS application. The following command selects records associated with the ACMS application, MYAPP, and writes the records to a file named MYAPP.DAT:

```
$ ACMSDI$DCM
DCM> SELECT MYAPP.DAT /APPLICATION=MYAPP
```

You can sort the file MYAPP.DAT, ordering it by (for example) ACMS Task, Node Name, and User ID. The sorted output can then be submitted to a user-written reporting program.

The name of the Data Compression Monitor log file is:

SYS$ERRORLOG:ACMSDI$COMPRESSION.LOG

Example 4–3 shows the layout of the records as they appear in the monitor log file and in the files created using the SELECT command.

The record consists of a fixed section of 109 bytes followed by one section for each workspace of 12 bytes. All fields are of data type char.

Example 4–3 Record Layout

```
The record format is:

Field name       Offset  Length  Description
---------------  ------  ------  ------------------------------------
Date/time stamp    0       24     Date and time record was written to
                                  the log in the following format:
                                      www mmm dd hh:mm:ss yyyy
                                      www  = day of week
                                      mmm  = month
                                      dd   = day of month
                                      hh   = hour
                                      mm   = minute
                                      ss   = second
                                      yyyy = year
                                      Example:
                                          Fri Sep 18 13:33:16 2001
```

Example 4–3 (Cont.) Record Layout

```
Node               24    20    Node identification. For TCP/IP this
                                is the IP address.

User Name          44    20    Identifier of the signed-in user.

Application Name   64    20    ACMS application name.

Task Name          84    20    ACMS task name.

Call Type          104   1     Code which identifies the type of
                                call:
                                    C = Task Call
                                    S = Send Exchange Step
                                    R = Receive Exchange Step
                                    T = Transceive Exchange Step
                                    D = TDMS Exchange Step

Direction Code     105   1     Direction in which the message was
                                sent:
                                    D = Message sent to desktop device
                                    H = Message sent to OpenVMS host

Workspace Count    106   3     Number of workspaces
```

Following the workspace count there is one 12-byte section for each workspace, starting at offset 109, in the following format:

```
Uncompressed Length   0    5    Length of workspace in uncompressed
                                 state.

Compressed Length     5    5    Length of workspace in compressed
                                 state.

Access Code          10    1    One of the following:
                                     R = Read Access
                                     W = Write Access
                                     M = Modify Access

Compression Code     11    1    One of the following:
                                     C = Workspace was compressed
                                     U = Workspace was uncompressable
                                     N = No attempt was made to compress
                                         this workspace
```

───────────────── **Note** ─────────────────

To keep the record length small, only the first 20 bytes of the User ID, Application Name and Task Name fields are captured. The fields are left-justified and, if necessary, padded with blanks on the right.

───

See the **HP TP Desktop Connector for ACMS** *Client Services Reference Manual* for the Data Compression Monitor commands.

4.2.7 Choosing the Network Software

The client services support DECnet and TCP/IP as transports (OpenVMS only). The network you use has no effect on how you write the client program. Specify the transport by linking the appropriate modules into your client program. See Section 4.8 for more information.

4.3 Writing Version-Checking Routines

TP Desktop Connector software provides entry points to standard **action routines** on the desktop system and in the gateway to allow applications to check for mismatched versions of software on the desktop system and ACMS systems.

4.3.1 Version-Checking Processing

The following action routines for version checking are defined in the TP Desktop Connector environment:

- ACMSDI_GET_VERSION on the gateway system (OpenVMS)

  The gateway calls this routine on the OpenVMS system whenever it receives an ENABLE request from the Execution Controller (EXC). The action routine can return a version string that is then passed to the desktop client program, allowing a version comparison at the desktop system.

- *acmsdi_check_version* on the client system

  The TP Desktop Connector client services call this routine whenever they receive an ENABLE request from the gateway. The action routine can check the version string passed from the ACMSDI_GET_VERSION routine on the submitter node and notify the desktop user of any inconsistency.

You can use ACMSDI_GET_VERSION and acmsdi_check_version for example, to compare the creation date of the desktop client program to the creation date of the ACMS application at run time. This helps catch errors related to mismatch of files on the desktop and the OpenVMS systems. If the desktop files are outdated, desktop users must obtain the correct files on their own. No special utility is provided.

To use the version-checking capability, you must supply complementary versions of the action routines on both the OpenVMS submitter node and the desktop system.

The action routines are customer-written. Figure 4–3 shows the processing that occurs during version checking.

Figure 4–3 Version-Checking Processing

TAY-0291A-AD
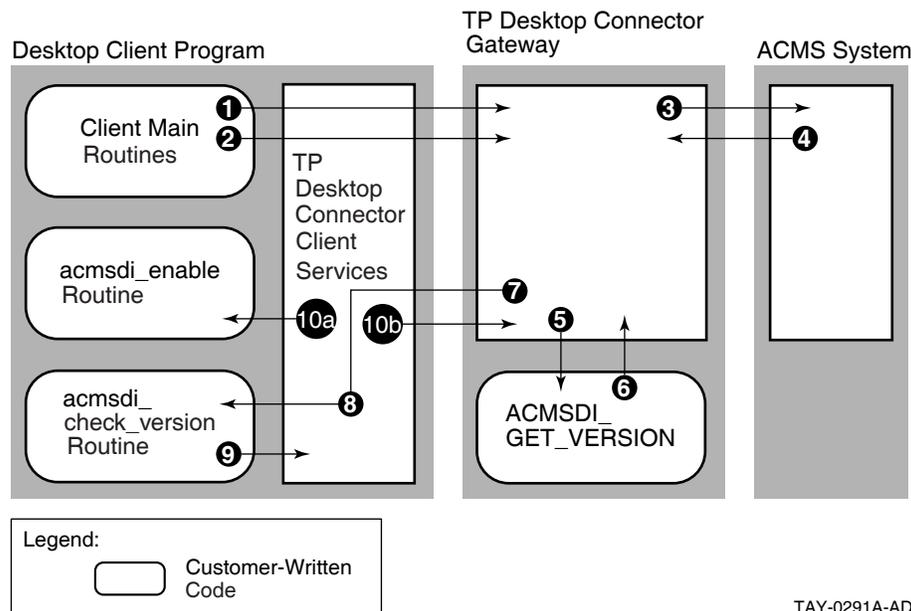
Figure 4–3 shows the following processing steps related to version checking:

1 The desktop client program requests version checking when it signs in to the ACMS system (see Section 4.3.2).

2 The desktop client program calls a task that uses FORM I/O.

3 The gateway passes the task call to the ACMS system.

4 ACMS software tells the gateway that the desktop client program has never accessed the specified form file.

5 The gateway searches for the ACMSDI_GET_VERSION image, calls the routine, and passes the form file specification (see Section 3.4).

6 The ACMSDI_GET_VERSION routine returns a string in a customer-determined format.

7 The gateway sends a combination acmsdi_enable and acmsdi_check_version message to the TP Desktop Connector client services on the TP Desktop Connector system, and includes the form file specification and the version string.

8 The TP Desktop Connector client services call the acmsdi_check_version routine in the desktop client program and pass the form file specification and the version string.

9 Depending on whether the version is acceptable, the acmsdi_check_version routine returns the FORMS_NORMAL status or any even-valued failure constant defined in the FORMS.H file.

10 The TP Desktop Connector client services can do the following:

a. If FORMS_NORMAL is returned, the TP Desktop Connector client services call the customer-written presentation procedure acmsdi_ enable and the processing continues.

b. If a failure status is returned, the TP Desktop Connector client services tell the gateway that the customer-written version check failed.

This failure appears in the audit trail log (ATL) as an error during the exchange step in the task processing.

4.3.2 Requesting Version Checking

Version checking is requested during a sign-in. Specify the ACMSDI_OPT_ CHECK_VERSION option on the acmsdi_sign_in call as the example in the *HP TP Desktop Connector for ACMS* Client Services Reference Manual shows. If version checking is enabled on the OpenVMS system that runs the gateway, the action routines are called whenever an ENABLE request is received from the application (see Section 4.3.1).

4.4 AVERTZ Sample Desktop Client Program

The AVERTZ sample desktop client program, called CLIENT.EXE, is written in Microsoft C and Microsoft COBOL. Of the numerous third-party presentation tools available, the sample uses only the forms management capabilities available in Microsoft COBOL. This is largely to ensure that the sample is as easily and widely understood as possible. The generic desktop client program routines for the sample described in Section 4.4.1 are in the C language rather than COBOL, because the TP Desktop Connector arguments are easier to manage in C.

Because portable tools can be used, much of the development work for the sample can be done on an OpenVMS system and ported to the desktop system for compilation, linking, and testing. This enables use of a single development environment for both the desktop client program and ACMS parts of the AVERTZ sample application, including using HP DECset tools.

Some presentation tools are not portable. For those portions of nonportable code, development must be done on the supporting platforms.

Samples are available from HP Services for other presentation tools. Any native presentation tool callable from COBOL or C that uses a procedural rather than an event-driven model is usable with TP Desktop Connector blocking client services.

4.4.1 AVERTZ Components

The TP Desktop Connector components of the CLIENT.EXE sample desktop client program replace the HP DECforms forms in the ACMS sample application with routines written in Microsoft COBOL and Microsoft C.

These routines in the desktop client program fall into the following categories:

- Generic client routines

  Call the specific TP Desktop Connector client services, for example, acmsdi_sign_in.

- Application-specific client routines

  Interact with the user to get information for signing in to the ACMS system, selecting tasks, signing out of the ACMS system and other processing.

- Generic presentation procedures

  Gain control during exchange steps, parse workspaces, and call application-specific presentation procedures.

- Application-specific presentation procedures

  Invoke the routines specific to each exchange step. They interact with the user to display and solicit information related to a specific exchange step.

- TP Desktop Connector client services

  Transmit and receive TP Desktop Connector messages (which are hidden from application-specific routines).

Figure 4–4 shows the categories of routines and their interaction in the sample desktop client program.

The TP Desktop Connector generic presentation procedures are analogous to the HP DECforms and TDMS services, for example, FORM$TRANSCEIVE and TSS$REQUEST.

Figure 4–4 TP Desktop Connector Sample Components
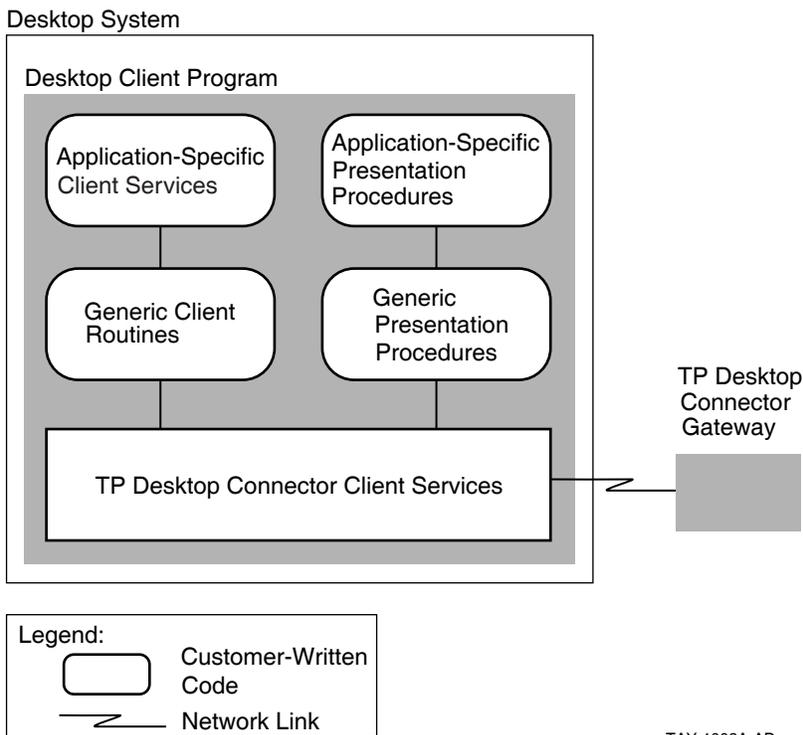
TAY-1002A-AD

4.4.2 AVERTZ Component Processing Flow

The user invokes the main routines in the source file CLIENT.C when starting the sample desktop client program. Figure 4–5 shows the processing flow of the desktop client program routines, including both the generic client routines written in C and the application-specific routines written in COBOL.

Figure 4–5 Processing Flow for Nonblocking Sample Desktop Client Program
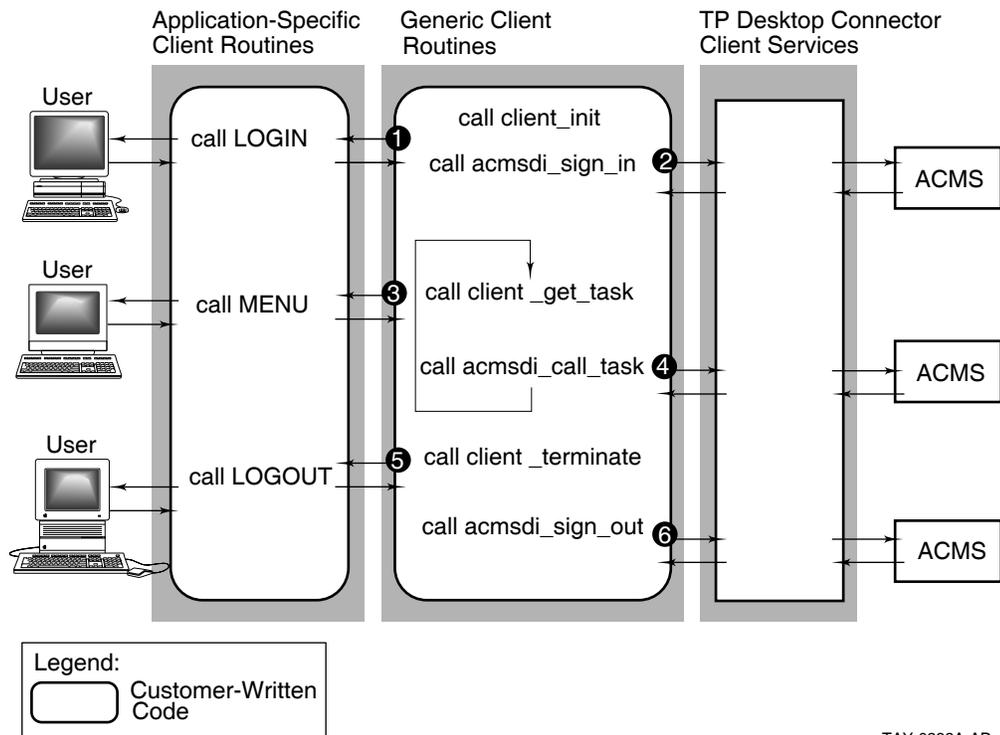
TAY-0293A-AD

A main routine sets up the structures needed for calling TP Desktop Connector software. The routine then calls the generic routines client_init, client_get_task, and client_terminate, each of which in turn invokes an application-specific COBOL program to interact with the user, as follows:

1 The generic routine client_init calls the program LOGIN.CBL to get ACMS sign-in information from the user.

2 The main routine calls the TP Desktop Connector client service acmsdi_sign_in to sign the user in to the ACMS system.

3 The main routine calls the generic routine client_get_task.

   When the sign-in completes, the user can select ACMS tasks. The generic routine client_get_task calls the application-specific COBOL program MENU to allow the user to select tasks from the reservation form.

4   The main routine calls the TP Desktop Connector client service acmsdi_
    call_task to start the ACMS task that the user selected.

    When a task completes, the application returns to the MENU routine and
    remains in this loop until the user selects EXIT.

5   The generic routine client_terminate is called when the user selects the
    option to sign out of the ACMS system.

    The generic routine client_terminate calls the application-specific COBOL
    program LOGOFF.

6   The main routine calls the TP Desktop Connector client service acmsdi_
    sign_out to sign the user out of the ACMS system and log the user out of
    the OpenVMS system. The user is signed out of the ACMS system.

The processing model for the sample TP Desktop Connector application
assumes that the user remains signed-in for extended periods of time.

4.4.3 Reusing the CLIENT.EXE Routines

If the processing model described for the sample desktop client program is
similar to the one in your solution, use the C language routines without change
for the particular solution you are building. Simply change the LOGIN.CBL,
MENU.CBL, and LOGOFF.CBL routines to use the chosen presentation tool
and modify the user interface presentation style to suit your users.

To increase the likelihood that you can use some of the sample code with
minimal changes, the CLIENT.EXE desktop client program routines are
organized into two levels: generic and application-specific. Figure 4–6 shows
the relationships among these procedures in handling exchange steps within
the AVERTZ sample.

Reserve is the only task implemented in the TP Desktop Connector version
of AVERTZ. The AVERTZ sample reserve task uses only TRANSCEIVE
statements in its task definition. For this reason, the only generic presentation
procedures that are invoked on the desktop are acmsdi_enable, acmsdi_disable,
and acmsdi_transceive. (Although the C routines do check application-specific
characteristics of the records and record identifications, they are classed as
generic.)

For the AVERTZ sample, the acmsdi_enable procedure simply returns a success
status to the TP Desktop Connector code that invokes it. This sample requires
no enable-related work.

Figure 4–6   Sample Presentation Procedures

TAY-1003A-AD
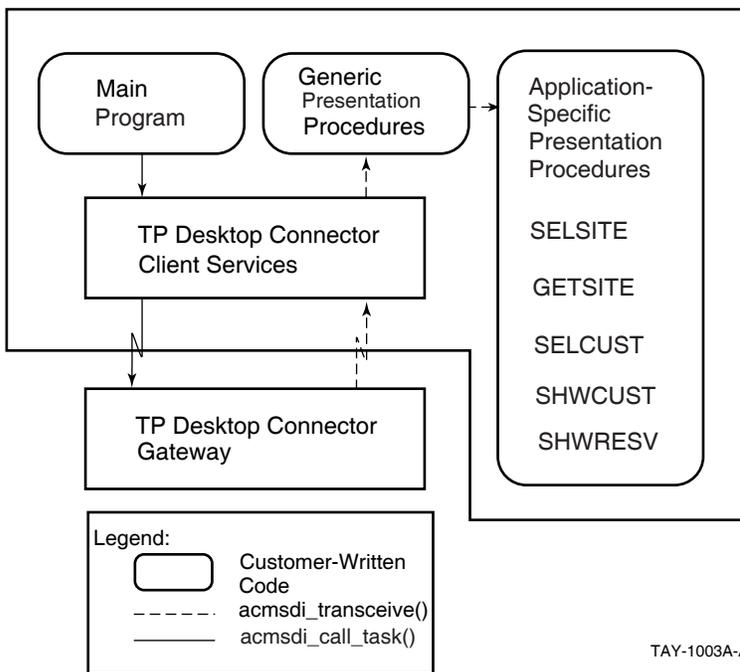
The generic acmsdi_transceive presentation procedure in the TRANS.C source
file looks at the record identifiers of the incoming message to determine what
application-specific presentation procedure to call. The generic presentation
procedure checks the validity of the record arguments and calls the appropriate
application-specific presentation procedure. The application-specific procedures
correspond directly to the exchange steps in the AVERTZ reserve task
definition shown in Example 4–4.

Example 4–4 AVERTZ Reserve Task Exchange Steps

```
REPLACE TASK AVERTZ_CDD_TASK:VR_RESERVE_TASK /LIST
   .
   .
   .
      EXCHANGE WORK IS
          IF (ctrl_key = "MLTSI") THEN
              TRANSCEIVE RECORD list_1, list_2     1
                  SENDING vr_control_wksp, vr_si_trans_array_wksp
                  RECEIVING vr_sites_wksp,
                            vr_reservations_wksp,
                            vr_customers_wksp,
                            vr_control_wksp
                  WITH SEND CONTROL vr_sendctrl_wksp;
          ELSE
              TRANSCEIVE RECORD list_3, list_2     2
                  SENDING vr_control_wksp, vr_sites_wksp
                  RECEIVING vr_sites_wksp,
                            vr_reservations_wksp,
                            vr_customers_wksp,
                            vr_control_wksp
                  WITH SEND CONTROL vr_sendctrl_wksp;
          END IF;

    ACTION
    IF (ctrl_key = "QUIT") THEN
        EXIT TASK;
    END IF;
   .
   .
   .
display_cust_info:
   EXCHANGE WORK IS
       !+
       ! Multiple customers found, use array to scroll and select one
       !-
       IF (ctrl_key = "MLTCU") THEN
               TRANSCEIVE RECORD list_5, list_6     3
               SENDING vr_control_wksp, vr_cu_trans_array_wksp,
                       vr_rental_classes_wksp, vr_sites_wksp
               RECEIVING vr_control_wksp, vr_reservations_wksp,
                       vr_customers_wksp SHADOW IS vr_customers_shadow_wksp,
                       vr_trans_wksp SHADOW IS vr_trans_shadow_wksp;
```

Example 4–4 (Cont.) AVERTZ Reserve Task Exchange Steps

```
        ELSE
        !+
        ! One or no customer found, send unique customer record
        !-
                TRANSCEIVE RECORD list_7, list_6    4
                SENDING vr_control_wksp, vr_customers_wksp, vr_trans_wksp,
                        vr_rental_classes_wksp, vr_sites_wksp
                RECEIVING vr_control_wksp, vr_reservations_wksp,
                          vr_customers_wksp SHADOW IS vr_customers_shadow_wksp,
                          vr_trans_wksp SHADOW IS vr_trans_shadow_wksp;
        END IF;

        ACTION
        CONTROL FIELD ctrl_key
                "QUIT" : EXIT TASK;
                "CHNGE": MOVE "     " TO ctrl_key;
                         GOTO STEP determine_site;
        END CONTROL;
    .
    .
    .
display_resv_no:
        EXCHANGE
                !+
                ! Display reservation # and prompt to see if want to check
                ! car out now.
                !-
                TRANSCEIVE RECORD list_8, list_9    5
                SENDING vr_control_wksp, vr_reservations_wksp
                RECEIVING vr_control_wksp,vr_reservations_wksp,
                          vr_customers_wksp;

        ACTION
        CONTROL FIELD ctrl_key
                "QUIT" : EXIT TASK;
        END CONTROL;
    .
    .
    .
END BLOCK WORK;
END DEFINITION;
```

Callouts **1** through **5** indicate the exchange steps in the task definition.

Dividing the presentation procedures into generic and application-specific routines isolates in relatively independent modules the presentation code that is unique to the application. In many cases, you can retain this overall structure for your application, writing new application-specific procedures

equivalent to GETSITE.CBL and SELCUST.CBL. Modify TRANS.C, SEND.C, RECV.C, ENABLE.C, DISABLE.C, and REQUEST.C to add each application-specific presentation procedure as an element of the case statement in the generic presentation procedure that calls it. For an application whose task definitions use only TRANSCEIVE in its exchange steps, the only generic procedure you must modify is the acmsdi_transceive procedure in TRANS.C.

See the **HP TP Desktop Connector for ACMS** *Installation Guide* for the names of the AVERTZ source directories and a list of their contents.

4.5 Writing Procedures Using Blocking TP Desktop Client Services

How you handle the user interface for signing in to the ACMS system, traversing menus, selecting tasks, and signing out of ACMS is constrained to some degree by the information you must include in the acmsdi_sign_in, acmsdi_call_task, and acmsdi_sign_out services. For example, the sign-in procedure must provide a valid OpenVMS user name and password on the acmsdi_sign_in service. To accomplish this, you can follow the AVERTZ sample application provided with TP Desktop Connector software.

4.5.1 Calling the Sign-In Service

In the CLIENT sample desktop client program, the mainline presentation code calls another routine, client_init, that actually gets the required information from the user. Alternatively, that presentation code can be included in the main part of the desktop client program.

The desktop client program must call the acmsdi_sign_in service with the correct parameters and must handle any error conditions returned. Example 4–5 shows the call in CLIENT.C to a client_init procedure and to acmsdi_sign_in.

Example 4–5 Signing In the User

```
main

char    username[MAX_USERNAME + 1] = "",
        password[MAX_PASSWORD + 1] = "",
        node[MAX_NODENAME + 1] = "",
                .
                .
                .
```

Example 4–5 (Cont.) Signing In the User

```
**  Get signin information from user
*/
client_init(node,  1
            username,
            password);
/*
**  Sign into remote ACMS node
*/
status = acmsdi_sign_in(node,   2
                        username,
                        password,
                        0,
                        &submitter_id);
/*
**  Overwrite password for security
*/
for (i = 0;  i < MAX_PASSWORD;  i++)
    password[i] = EOS;

if (status != ACMSDI_NORMAL)
{
    fprintf(stderr,"Error signing user %s into node %s.", username, node);
    exit;
}
```

The important points of Example 4–5 are as follows:

1 The call to client_init invokes a sign-in procedure that prompts the user for the following information:

   • Node name identifying the ACMS system to connect to

   • User name (the same name for both OpenVMS and ACMS) under which to log in to that node

   • Password for that user name

2 The call to acmsdi_sign_in passes the information obtained from the user to the ACMS system.

Example 4–6 shows the COBOL code in the LOGON.CBL program for obtaining the sign-in information from the application user on the desktop system.

Example 4–6 Login Program

```
PROCEDURE DIVISION USING
                   NODE-NAME,
                   USERNAME,
                   PASSWORD.

000-MAINLINE.
            .
            .
            .
*
*    Display login screen
*
     MOVE SPACES TO USERNAME, PASSWORD, NODE-NAME.
     DISPLAY COLOR-SCREEN.
     DISPLAY LOGIN-PANEL.
*
*    Get login information from user
*
     MOVE "N" TO VALID-DATA-FLAG.
     MOVE SPACES TO ERROR-MSG.
     MOVE ZEROES TO CURSOR-POSITION.
     PERFORM UNTIL VALID-DATA
         MOVE "Y" TO VALID-DATA-FLAG
         IF (ERROR-MSG NOT EQUAL SPACES)
         THEN
             DISPLAY ERROR-MSG AT LINE 25 COLUMN 1 WITH BELL
             MOVE SPACES TO ERROR-MSG
         ELSE
             DISPLAY ERROR-MSG AT LINE 25 COLUMN 1
         END-IF
         ACCEPT LOGIN-PANEL
         MOVE ZEROES TO CURSOR-POSITION
         IF (USERNAME EQUAL SPACES)
         THEN
            MOVE "N" TO VALID-DATA-FLAG
            MOVE "Username is required" TO ERROR-MSG
            MOVE 9 TO CURSOR-LINE
            MOVE 35 TO CURSOR-COLUMN
         ELSE
         IF (PASSWORD EQUAL SPACES)
         THEN
            MOVE "N" TO VALID-DATA-FLAG
            MOVE "Password is required" TO ERROR-MSG
            MOVE 10 TO CURSOR-LINE
            MOVE 35 TO CURSOR-COLUMN
         ELSE
```

(continued on next page)

Example 4–6 (Cont.) Login Program

```
      IF (NODE-NAME EQUAL SPACES)
      THEN
          MOVE "N" TO VALID-DATA-FLAG
          MOVE "Login node is required" TO ERROR-MSG
          MOVE 12 TO CURSOR-LINE
          MOVE 35 TO CURSOR-COLUMN
      END-IF
  END-PERFORM.
```

The program in Example 4–6 uses the Microsoft COBOL presentation capabilities. You can use other presentation tools that allow developing a more sophisticated user interface.

Some data validation is performed in the login program shown in Example 4–6. The presentation procedure can ensure the validity of the data before sending it to TP Desktop Connector Gateway for ACMS software. Because less invalid data is communicated, the network traffic between the desktop system and the ACMS system is minimized.

4.5.2 Enabling Password Expiration Checking

Password expiration checking allows the client application to determine if the end-user's password is expiring soon on the gateway node. If password expiration checking is enabled on both the gateway and on the client's call to the acmsdi_sign_in service, TP Desktop Connector client services indicate how many hours are left until the user's password expires on the gateway node.

To use the password expiration checking option:

1. Enable the password expiration checking in the gateway:

    a. Set the PASSWORD_EXP value in the gateway's parameters file.

    b. Determine how many days before the password expires that you want the desktop gateway to start sending password expiration warning messages to TP Desktop Connector clients.

    c. Set the value of PASSWORD_EXP with an integer value representing that number of days.

    d. Restart the gateway in order for the values in the gateway's parameters file to take effect.

    For more information on managing the gateway, see the ***HP TP Desktop Connector for ACMS*** *Gateway Management Guide*.

2. Enable the password expiration checking option in the client program's call to acmsdi_sign_in:

   a. Declare a buffer (in the client application) that Desktop services can use to write the number of hours until password expiration.

   b. Declare a sign-in options array:

      ```
      long hours_until_password_expiration;
      ACMSDI_OPTION options[2];
      ```

   c. In the sign-in options array, specify the ACMSDI_OPT_PWD_EXPIRING option and the address of the variable that will contain the hours left until password expiration:

```
options[0].option = ACMSDI_OPT_PWD_EXPIRING;
options[0].pwd_expiring_hrs.address = &hours_until_password_expiration;
options[1].option = ACMSDI_OPT_END_LIST;
```

3. Check for an ACMSDI_PWDEXPIRING return status after the call to acmsdi_sign_in( ). Applications using nonblocking services should check the status when the acmsdi_sign_in completion routine is called:

```
if (status == ACMSDI_PWDEXPRING)
{
  printf("Warning !  Your Password Will Be Expiring in %d Hours!",
         hours_until_password_expiration);
}
```

If the acmsdi_sign_in routine returns a status of ACMSDI_PWDEXPIRED, then the user's password has already expired.

4.5.3 Establishing an Exit Handler

If the desktop client program uses the standard C routines atexit or onexit to establish an exit handler, and the exit handler calls acmsdi_sign_out, make the call to atexit or onexit after the first call to acmsdi_sign_in. If this precaution is not taken, the exit handling built into TP Desktop Connector occurs before the desktop client program exit handler call to acmsdi_sign_out, the submitter or submitters are already signed out, and the desktop client program gets error statuses that can be ignored.

4.5.4 Calling Tasks and Signing Out

For examples that handle menu traversal, task selection, and sign-out, see the MENU.CBL and LOGOFF.CBL programs from the AVERTZ sample desktop client program.

4.5.5 Passing Multiple Workspaces on acmsdi_call_task

Some TP Desktop Connector client applications may need to pass workspaces on the call to the acmsdi_call_task service. Applications that use NO I/O tasks, for example, must pass workspace data as a parameter of the acmsdi_call_task service.

To do this, the application must create an array of workspace descriptors, which is then passed as the *workspaces* parameter on acmsdi_call_task.

The workspace descriptor type, ACMSDI_WORKSPACE, is used to describe the size and address of the workspace. The client application can use the ACMSDI_INIT_WORKSPACE macro to create each ACMSDI_WORKSPACE descriptor.

The workspace array should contain a workspace descriptor for each workspace that is to be passed on acmsdi_call_task. Example 4–7 illustrates how an application passes three workspaces on the acmsdi_call_task( ) service.

Example 4–7 Passing Three Workspaces

```
ACMSDI_WORKSPACE wksp_array[3];   /* Declare array of wksp descriptors */

struct {
  char ctrl_key[5];
  char message[80];
} control_wksp;

struct {
  int  id_number;
  char first_name[15];
  char last_name[25];
} employee_record;

struct {
  int  dept_id;
  char dept_name[15];
} department_record;
int status;

/*
**  Create the workspace descriptors
*/
```

(continued on next page)

Example 4–7 (Cont.)   Passing Three Workspaces

```
ACMSDI_INIT_WORKSPACE(wksp_array[0], control_wksp);
ACMSDI_INIT_WORKSPACE(wksp_array[1], employee_record);
ACMSDI_INIT_WORKSPACE(wksp_array[2], department_record);

/*
**  Pass the number of workspaces passed (3) along with
**  the array of workspace descriptors on the call to
**  acmsdi_call_task()
*/

status = acmsdi_call_task(submitter_id,
                          NULL,
                          "MY_TASK",
     "MY_APPL",
     NULL,
     call_status,
     3,
     wksp_array,
     NULL, NULL, NULL, NULL);
```

4.5.6 Using Unidirectional Workspaces on acmsdi_call_task

When an application passes workspaces on a call to the acmsdi_call_task
service, the unidirectional-workspaces option allows the application to specify
which of those workspaces are read-only workspaces, which are write-only
workspaces, and which are modify workspaces. See Section 2.4.5.1 to
determine what is meant by read-only and write-only for the purposes of
the unidirectional workspaces feature.

To use the unidirectional-workspaces option:

1. Enable the OPTIMIZE_WORKSPACES option in the client program's call
    to acmsdi_call_task.

2. Create an array of unidirectional-workspace descriptors to be passed on the
    call to acmsdi_call_task.

For unidirectional workspaces, the application must use the ACMSDI_
WORKSPACE_OPT type in order to specify an access type for each workspace.
The workspace descriptor type, ACMSDI_WORKSPACE_OPT, is used to
describe the size, address, and access type of the workspace. The access type
of each workspace must be specified as either ACMSDI_ACCESS_READ,
ACMSDI_ACCESS_WRITE, or ACMSDI_ACCESS_MODIFY.

The client application can use the ACMSDI_INIT_WORKSPACE_OPT macro to create each ACMSDI_WORKSPACE_OPT descriptor. The workspace array should contain a workspace descriptor for each workspace that is to be passed on acmsdi_call_task. Example 4–8 illustrates how an application passes three workspaces on the acmsdi_call_task( ) service.

Example 4–8 Passing Unidirectional Workspaces

```
ACMSDI_CALL_OPTION    call_options[2]; /* Declare call options array */

ACMSDI_WORKSPACE_OPT wksp_array[3];   /* Declare unidirectional wksp
                                      ** descriptors array
                                      */
struct {
  char ctrl_key[5];
  char message[80];
} control_wksp;

struct {
  int  id_number;
  char first_name[15];
  char last_name[25];
} employee_record;

struct {
  int  dept_id;
  char dept_name[15];
} department_record;

int status;

/*
** Create the unidirectional workspace descriptors
*/

ACMSDI_INIT_WORKSPACE_OPT(wksp_array[0], control_wksp, ACMSDI_ACCESS_WRITE);
ACMSDI_INIT_WORKSPACE_OPT(wksp_array[1], employee_record, ACMSDI_ACCESS_READ);
ACMSDI_INIT_WORKSPACE_OPT(wksp_array[2], department_record, ACMSDI_ACCESS_MODIFY);

/*
**  Turn on the unidirectional workspace call option
*/

call_options[0].option = ACMSDI_CALL_OPT_OPTIMIZE_WKSPS;
call_options[1].option = ACMSDI_CALL_OPT_END_LIST:
```

Example 4–8 (Cont.) Passing Unidirectional Workspaces

```
/*
** Pass the call options array, along with the number
** of workspaces passed and the array of unidirectional
** workspace descriptors, on the call to acmsdi_call_task()
*/

status = acmsdi_call_task(submitter_id,
                          call_options,
                          "MY_TASK",
        "MY_APPL",
      NULL,
      call_status,
      3,
      wksp_array,
      NULL, NULL, NULL, NULL);
```

_____ **Note** _____

The ACMSDI_CALL_OPT_OPTIMIZE_WKSPS option tells the TP
Desktop Connector client services to interpret the array of workspace
descriptors as data type ACMSDI_WORKSPACE_OPT. The workspace
optimization option and the ACMSDI_WORKSPACE_OPT must be
used together, or not at all. Using one without the other produces
unpredictable results.

_____

4.5.7 Providing Stub Routines

ACMS task definitions do not have to include exchange steps, as is the case
with tasks that include only processing steps and specify NO TERMINAL
USER I/O as the I/O method. For these tasks, you do not write presentation
procedures. However, in your desktop client program, you must supply stub
routines for all possible entry points, that is, presentation procedures and
action routines (for example, acmsdi_check_version).

The TP Desktop Connector client services refer to all the presentation
procedures and the acmsdi_check_version routine. If you do not have code
for one or more of these routines in your desktop client program, provide a stub
for the linker to use to resolve these references.

Stubs for all presentation procedures and the acmsdi_check_version routine are
provided in the file PPSTUBS.C. PPSTUBS.C provides stubs for the following
routines:

- acmsdi_disable

- acmsdi_enable

- acmsdi_receive

- acmsdi_request

- acmsdi_send

- acmsdi_transceive

- acmsdi_check_version

_____ **Note** _____

The file PPSTUBS.C is provided in the ACMSDI$COMMON directory
on the OpenVMS system. Copy this file to the desktop system when
the TP Desktop Connector client services software is installed.

_____

To use this source code, follow these steps:

1. Edit or comment out any of the presentation procedures that you
   implement as part of your desktop client program.

2. Edit or comment out the acmsdi_check_version routine if you provide your
   own version-checking routine.

3. Compile the module.

Include the resulting object module in build procedure (see Section 4.7).

4.6 Writing Presentation Procedures in a Blocking Environment

If your ACMS tasks include exchange steps, write presentation procedures in
your desktop client program to handle the interaction with the user required
by these exchange steps. TP Desktop Connector software invokes a given
presentation procedure when it receives from the desktop gateway an exchange
step message corresponding to that procedure.

Your desktop client program must include presentation procedures that
correspond to any of the exchange step types actually used in your task
definitions:

- SEND exchange steps invoke acmsdi_send.

- RECEIVE exchange steps invoke acmsdi_receive.

- TDMS READ exchange steps invoke acmsdi_read_msg.

- TDMS WRITE exchange steps invoke acmsdi_write_msg.

- TRANSCEIVE exchange steps invoke acmsdi_transceive.

- REQUEST exchange steps invoke acmsdi_request.

TP Desktop Connector client services call acmsdi_read_msg when a TDMS Read exchange is received from the TP Desktop Connector Gateway for ACMS on the host OpenVMS system. The acmsdi_read_msg presentation procedure displays the prompt, if any, sent from the ACMS task, then acquires the text from the form's Message Field to be returned to ACMS.

TP Desktop Connector client services call acmsdi_write_msg when a TDMS Write exchange is received from the TP Desktop Connector Gateway for ACMS on the host OpenVMS system. The acmsdi_write_msg presentation procedure displays the message text sent from the ACMS task in the form's Message Field. See the **HP TP Desktop Connector for ACMS** *Client Services Reference Manual* for the syntax of these procedures.

For a FORM I/O task, include the following presentation procedures as well:

- The first exchange step to specify a HP DECforms form invokes the acmsdi_enable presentation procedure.

- Closing a TP Desktop Connector session invokes the acmsdi_disable presentation procedure, if any of the tasks used FORM I/O.

See Section 4.6.1 for more information on acmsdi_enable and acmsdi_disable.

In the AVERTZ sample application, these presentation procedures in turn call routines that correspond to the specific exchange step in which the SEND, RECEIVE, TRANSCEIVE, or REQUEST keyword is used.

Example 4–9 shows the TRANSCEIVE presentation procedure, in file trans.c from the AVERTZ sample application.

Example 4–9 TRANSCEIVE Presentation Procedure

```
long acmsdi_transceive(ACMSDI_FORMS_SESSION_ID *session_id,
                       char *send_record_id,
                       long send_record_count,
```

Example 4–9 (Cont.)   TRANSCEIVE Presentation Procedure

```
                    char *recv_record_id,
                    long recv_record_count,
                    char *recv_ctl_text,
                    long *recv_ctl_text_count,
                    char *send_ctl_text,
                    long send_ctl_text_count,
                    short timeout,
                    ACMSDI_CALL_ID *call_id,
                    void *call_context,
                    ACMSDI_FORM_RECORD *send_records,
                    ACMSDI_FORM_RECORD *recv_records )    1
{
long int sts;
short int i;

    if (send_record_id == NULL)
        return (FORMS_BADARG);

    if (recv_record_id == NULL)
        return (FORMS_BADARG);

    for (i = 0; i < send_record_count; i++)
    {
        if (recv_records[i].data_record == NULL)
            return (FORMS_INVRECDES);
    }

    sts = FORMS_NORECORD;

    if ((0 == strcmp (send_record_id, "LIST_1")) &&     2
        (0 == strcmp (recv_record_id, "LIST_2")))
    {
        /*
        **  Validate arguments
        */
        if ((send_record_count != 2) ||
            (recv_record_count != 4))
            return (FORMS_BADRECCNT);

        if (send_records[0].data_length != 123)
            return (FORMS_BADRECLEN);

        if (send_records[1].data_length != 1730)
            return (FORMS_BADRECLEN);

        if (recv_records[0].data_length != 138)
            return (FORMS_BADRECLEN);

        if (recv_records[1].data_length != 144)
            return (FORMS_BADRECLEN);
```

Developing Portable API Client Programs   **4–39**

Example 4–9 (Cont.) TRANSCEIVE Presentation Procedure

```
        if (recv_records[2].data_length != 255)
            return (FORMS_BADRECLEN);

        if (recv_records[3].data_length != 123)
            return (FORMS_BADRECLEN);
  .
  .
  .
        /*
        **  Call Presentation Procedure
        */

        GETSITE (                                                3
            session_id,
            form_data,
            send_ctl_text, /** VR_SENDCTRL_WKSP **/
            send_records[0].data_record, /** VR_CONTROL_WKSP **/
            send_records[1].data_record, /** VR_SITES_WKSP **/
            recv_records[0].data_record, /** VR_SITES_WKSP **/
            recv_records[1].data_record, /** VR_RESERVATIONS_WKSP **/
            recv_records[2].data_record, /** VR_CUSTOMERS_WKSP **/
            recv_records[3].data_record  /** VR_CONTROL_WKSP **/
                    );

        sts = FORMS_NORMAL;

    } /** end if strcmp **/
                .
                .
                .

    return (sts);
}
```

The following key points are called out in the example:

1 The acmsdi_transceive presentation procedure is called when a
TRANSCEIVE exchange step executes in the ACMS task.

2 The procedure looks at the contents of the send and receive record identifier
fields to determine what second-level, application-specific routine to call.
This is termed the application-specific presentation procedure.

3 The desktop client program runs the presentation procedure.

In Example 4–10, the GETSITE.CBL program receives data from the acmsdi_
transceive presentation procedure.

_____ **Note** _____

PPGEN.COM is a sample tool that generates these generic presentation
procedures based on task definitions in the TDB. PPGEN.COM
determines the appropriate record counts, data record lengths, and so
on, for each application-specific presentation procedure. PPGEN.COM
is located in ACMSDI$EXAMPLES.

_____

Example 4–10 GETSITE Application-Specific Presentation Procedure

```
     PROCEDURE DIVISION USING
                      SESSION-ID,
                      FORM-DATA,
                      VR-SENDCTRL-WKSP,
                      CONTROL-IN,
                      SITES-IN,
                      VR-SITES-WKSP,
                      VR-RESERVATIONS-WKSP,
                      VR-CUSTOMERS-WKSP,
                      VR-CONTROL-WKSP.
  .
  .
  .
  *
  *     Distribute data
  *
       PERFORM 010-INITIALIZE-RECS
           THRU 010-INITIALIZE-RECS-EXIT.
       MOVE SITES-IN TO VR-SITES-WKSP.    1
       MOVE CORRESPONDING VR-SITES-WKSP TO FORM-DATA.
       MOVE CONTROL-IN TO VR-CONTROL-WKSP.
       ACCEPT CURRENT-DATE FROM DATE.
       MOVE CORRESPONDING CURRENT-DATE
           TO CHECKOUT-DATE-2, RETURN-DATE-2.
       MOVE THIS-CENTURY
           TO CC OF CHECKOUT-DATE-2,
               CC OF RETURN-DATE-2.
  .
  .
  .
```

Example 4–10 (Cont.) GETSITE Application-Specific Presentation Procedure

```
*
*       Collect data from screen                          2
*
        MOVE CORRESPONDING FORM-DATA TO VR-SITES-WKSP.
        MOVE CORRESPONDING FORM-DATA TO VR-CUSTOMERS-WKSP.
        MOVE CORRESPONDING FORM-DATA TO VR-RESERVATIONS-WKSP.
        MOVE CORRESPONDING RETURN-DATE
            TO VEHICLE-EXPECTED-RETURN-DATE.
        MOVE CORRESPONDING CHECKOUT-DATE
            TO VEHICLE-CHECKOUT-DATE.
    .
    .
    .
*
*       Validate site and customer
*
        IF ((CUSTOMER-ID OF FORM-DATA EQUAL ZERO) AND     3
            (CU-LAST-NAME OF FORM-DATA = SPACES AND
             CU-FIRST-NAME  OF FORM-DATA = SPACES))
        THEN
            MOVE 'N' TO VALID-DATA-FLAG
            MOVE "Either CUSTOMER ID or NAME is required"
                TO MESSAGEPANEL OF VR-CONTROL-WKSP
            MOVE 5 TO CURSOR-LINE
            MOVE 15 TO CURSOR-COLUMN
        ELSE
        IF ((SITE-ID OF FORM-DATA EQUAL ZERO) AND
            (CITY OF FORM-DATA EQUAL SPACES))
        THEN
            MOVE 'N' TO VALID-DATA-FLAG
            MOVE "Either SITE ID or CITY is required"
                TO MESSAGEPANEL OF VR-CONTROL-WKSP
            MOVE 11 TO CURSOR-LINE
            MOVE 11 TO CURSOR-COLUMN
        END-IF.

        IF (NOT VALID-DATA) THEN GO TO 100-GET-DATA-EXIT.
*
*       Validate dates and car type
*
    .
    .
    .
```

Example 4–10 (Cont.) GETSITE Application-Specific Presentation Procedure

```
     100-GET-DATA-EXIT.
         EXIT.              4
 .
 .
 .
     END PROGRAM "_GETSITE".
```

The GETSITE program does the following:

1 Displays to the user the data received from the generic presentation procedure.

2 Accepts new data.

3 Verifies the data.

4 Returns the new data to the acmsdi_transceive procedure.

4.6.1 Coding for acmsdi_enable and acmsdi_disable

If your tasks use any of the three HP DECforms exchange step types (SEND, RECEIVE, TRANSCEIVE), supply these presentation procedures:

- acmsdi_enable

  Write an acmsdi_enable routine to initialize structures that are used by a number of presentation procedures. The acmsdi_enable routine is invoked when a user calls a task that includes a HP DECforms form name that has not previously been referenced in a task. The acmsdi_enable service is called once for every user and form combination in a task, not for every iteration of the same task.

- acmsdi_disable

  Write the acmsdi_disable routine to clear structures used by the presentation procedures. The acmsdi_disable presentation procedure is invoked when the desktop code issues the acmsdi_sign_out call.

4.6.2 Coding Return Status Values

ACMS software expects specific return values from the presentation procedures. These values correspond to the HP DECforms forms or the TDMS request return values that the customer-written procedure server receives as if the exchange is actually handled by these forms products on the OpenVMS system. In writing presentation procedures, ensure that the values the desktop client program returns to the ACMS system are valid OpenVMS status values.

See Appendix B for more information on determining these status values.

4.7 Building and Debugging the Desktop Client Program

Compile and link the desktop client program as you would any program.

4.7.1 Linking the Desktop Client Program

Link the executable desktop client program with the debug qualifier initially to look for these problems:

- Errors in argument order, format, or content in your procedures or in the invocations of TP Desktop Connector client services

- Invalid return values in presentation procedures

Link the desktop client program with the following types of libraries:

- Application-specific library containing:

  - Object files of the presentation procedures and menu procedures in your desktop client program

  - Stub routines for presentation procedures and action routines that your desktop client program does not use (see Section 4.5.7)

  Use the library utility on your desktop system to create a runtime library from the .OBJ format files.

- Library that contains the TP Desktop Connector client services (ACMSDI.LIB)

  TP Desktop Connector client services are provided through DLLs only. The ACMSDI.LIB is an import library, not a static-link library.

  The ACMSDI.DLL (originally installed as ACMSDIWS.DLL) is built to use the Winsock over TCP/IP DLL provided on the Windows kit. The Winsock DLL is called WSOCK32.DLL and must be in a directory where Windows can locate it at run time.

  Follow these steps to use the TCP/IP transport:

  1. Select the file ACMSDIWS.DLL.

  2. Copy and rename it to ACMSDI.DLL.

  3. Place ACMSDI.DLL in the executable path, where it can be located by the operating system when running the application.

  4. Copy the reference library ACMSDI.LIB from the appropriate Windows directory on the TP Desktop Connector Gateway for ACMS host. See **HP TP Desktop Connector for ACMS** *Gateway Management Guide* and the **HP TP Desktop Connector for ACMS** *Installation Guide*

for directions on obtaining the files that match the Windows operating environment.

- Language library or libraries

  Use either the C runtime library, the COBOL runtime library, or both, depending on which languages you use. These libraries are supplied with the compilers.

After generating the executable file, run it as you would run any Windows program.

After you test your presentation code, relink it without the debug qualifier. See **HP TP Desktop Connector for ACMS** *Gateway Management Guide* for a description of how to manage the application after testing it.

4.7.2 Maximum Lengths for Environmental Variables

The length of values for TP Desktop Connector environment logical names *must not* be greater than the maximum lengths indicated in Table 4–4. Using logical names that have values longer than the indicated maximum length can produce unpredictable results.

For example:

```
> set ACMSDI_LOG=my_file_name
```

where *my_file_name* is 256 characters or less.

Table 4–4 Maximum Lengths for Environmental Variables

Logical Name	Maximum Length of Value
ACMSDI_LOG	256
ACMSDI_MAXBUF	4
ACMSDI_TCPIP_PORT_host	7

4.7.3 Debugging the Desktop Client Program with Tasks

First, debug the presentation code on the desktop system. When the presentation code runs, debug the desktop client program with the ACMS software.

Follow these guidelines:

- For debugging tasks, see Section 3.6.3.

- For using desktop client program logging, see **HP TP Desktop Connector for ACMS** *Gateway Management Guide*.

4.8 Using the Desktop Client Program on Other Systems

If you use a presentation tool that is portable between OpenVMS and the
runtime environment, you can port the client program to OpenVMS. Porting of
the desktop client program to the desktop system then involves relatively little
work beyond rebuilding the executable images for the desktop client program
and performing verification tests on those executable images.

5

Using Portable API Extensions for Microsoft Windows

This chapter describes developing nonblocking desktop client programs for event-driven, multitasking environments such as Microsoft Windows. Examples in this chapter use Windows SDK Version 3.1 code to illustrate the use of *HP TP Desktop Connector* for ACMS nonblocking services in an event-driven environment. However, the same techniques can be used in other Windows event-driven environments.

You may also want to structure your application to use nonblocking services if you want to support multiple active sign-ins for failover, or multiple active tasks. Chapter 4 gives a useful background in the TP Desktop Connector client services concepts and terminology. For information on designing your application with nonblocking services, see Chapter 2.

For information on configuring network transports for any of the client platforms, see the *HP TP Desktop Connector for ACMS Client Services Reference Manual*.

5.1 Event-Driven Processing

A Windows desktop client program must yield control to the Windows manager as quickly as possible so that other event processing can be handled in a timely way. TP Desktop Connector nonblocking services provide methods to facilitate yielding control:

- acmsdi_dispatch_message

  Receives requests and responses from the gateway and dispatches the corresponding procedures.

- acmsdi_complete_pp

  Sends responses to the gateway when presentation procedures complete.

Figure 5–1 shows the sequence of processing in an event-driven desktop client program.

Figure 5–1 Event-Driven Desktop Client Program Processing

Window Events	Nonblocking Desktop Client Program	TP Desktop Connector Client Services	ACMS Task
User Menu Selection	\<event-processing\>	start task	BLOCK WORK
	acmsdi_call_task()		
Timer Event	\<event-processing\>	start presentation procedure	EXCHANGE IS TRANSCEIVE
	acmsdi_dispatch_ message		
Dialog Box Displayed	acmsdi_transceive()		
User Presses OK Button	\<event-processing\>	end presentation procedure	ACTION IS
	acmsdi_complete_pp()		
Timer Event	\<event-processing\>	end task	END BLOCK
	acmsdi_dispatch_ message		
Message Box Displayed	completion_routine		
	\<event-processing\>		

Legend:

Customer-Written Code

TAY-0295A-AD

When the user selects a task from the menu, the desktop client program calls the acmsdi_call_task service to initiate a ACMS task. In the nonblocking environment, nonblocking parameters are specified in the call (see the **HP TP Desktop Connector for ACMS** *Client Services Reference Manual*). One nonblocking parameter is the completion routine, which specifies a routine address in the desktop client program that the TP Desktop Connector client service calls when the TP Desktop Connector Gateway for ACMS completes the task. Because you specify the *completion routine* parameter in the call, the TP Desktop Connector client service returns control to the desktop client program immediately after the start task request is sent to the TP Desktop Connector Gateway for ACMS.

The desktop client program does not wait for the TP Desktop Connector gateway to request ACMS software to start a task in a ACMS application. Instead, the desktop client program returns control to the Windows message-processing loop for more event processing.

The acmsdi_call_task service also returns a **call identification** that the desktop client program supplies to the acmsdi_complete_pp service. The services use the call identification to associate an active call with a submitter. A **submitter** uniquely identifies a sign-in. The acmsdi_sign_in returns the submitter ID to identify that connection or session for subsequent call-tasks and sign-out on behalf of that sign-in. In addition, the call ID, which uniquely identifies a task invocation, is passed to all of its related presentation procedures. Note that TP Desktop Connector supports, at most, one active task for each submitter.

Because of an exchange step in the I/O task, the ACMS system starts a transceive operation. The TP Desktop Connector gateway sends a message to the desktop client program to start a presentation procedure.

Run as part of the control mechanism established by the desktop client program, the acmsdi_dispatch_message service polls for the message and, when the message is received, calls the customer-written presentation procedure acmsdi_transceive as shown in Figure 5–1. The acmsdi_transceive routine runs, displays a dialog box, saves pointers to the workspaces, and returns control to Windows without the user having signaled completion.

The user then has an opportunity to edit or add data to a form in the context of this exchange step. When the user later presses OK to signal completion, the desktop client program uses the workspace pointers to store the user-entered data, and calls the acmsdi_complete_pp service to send the response status to the TP Desktop Connector Gateway for ACMS. (The desktop client program does not wait for a response from the TP Desktop Connector gateway. Generally, it returns control to the Windows message-processing loop as soon

as possible.) Meanwhile, the TP Desktop Connector gateway passes to the ACMS task any valid workspaces and the status from the completion of the presentation procedure.

When the task completes on the ACMS system, the TP Desktop Connector gateway sends the end task message to the desktop system. The TP Desktop Connector client services dispatch the end task message by calling a **completion routine** in the desktop client program. This completion routine is dispatched by the polling mechanism and acmsdi_dispatch_message.

In this event-driven processing environment, the TP Desktop Connector gateway sends a message to the desktop client program for the following reasons:

- To signal the completion of a TP Desktop Connector client service

  The desktop client program supplies the address of completion routines in the nonblocking forms of the acmsdi_sign_in, acmsdi_call_task, and acmsdi_sign_out services.

- To process an exchange step in the task

  The desktop client program supplies presentation procedures structured to complete their processing separately from starting it.

The TP Desktop Connector client services dispatch the TP Desktop Connector gateway messages to the desktop client program by the polling mechanism established.

5.2 Guidelines for Developing Windows Desktop Client Programs

The following list summarizes general requirements and guidelines for developing Windows nonblocking desktop client programs as described in this chapter:

- Write procedures to use nonblocking TP Desktop Connector client services (see Section 5.4).

  Organize functions so that every call to a TP Desktop Connector client service or a customer-written presentation procedure has a parallel completion routine.

- Set up polling within the desktop client program (see Section 5.4.2).

  Set up a mechanism to retrieve desktop messages. Use this mechanism to periodically gain control and call the acmsdi_dispatch_message service to retrieve messages from the TP Desktop Connector gateway.

- Establish session context in the desktop client program (see Section 5.4.3).

Set up structures for handling multiple sessions. Certain data must be maintained as context for a current session and inactive sessions.

- Write presentation procedures to coexist in the nonblocking environment (see Section 5.5).

  To complete all presentation procedures, use the acmsdi_complete_pp service in routines separate from the initiating routines.

- Convert data as required between the desktop system and the ACMS system (see Section 4.1.3).

The following guideline is optional when developing a nonblocking desktop client program:

- Write an action routine to perform version checking.

  The module versionw.c shows a stub version-checking routine (see Section 4.3).

If you are using Microsoft C, you can use the Programmer's WorkBench (PWB) supplied with the Microsoft C compiler. Compiling routines to include a Browser database of the desktop client-program source code provides helpful tracing and mapping capabilities similar to those provided by OpenVMS with Language-Sensitive Editor (LSE) and Source Code Analyzer (SCA).

The remainder of the chapter explains the guidelines using code from the AVERTZ sample desktop client program.

5.3 AVERTZ Sample Desktop Client Program for Microsoft Windows

The AVERTZ sample desktop client program for the Windows environment is written in Microsoft C and follows the standards and practices outlined in the Microsoft Windows Software Development Kit (SDK) documentation. The program uses a main window, icons, and dialog boxes to interact with the user.

The program was developed with the PWB software that includes mechanisms to aid debugging.

5.3.1 AVERTZ Components for Microsoft Windows

A Windows desktop client program has the same high-level components as a blocking application (see Figure 5–2). However, note the addition of the polling mechanism as well as how the application-specific presentation procedures are now broken into two parts. TP Desktop Connector supports, at most, one active exchange step per task call.

Figure 5–2   TP Desktop Connector Sample Components for Microsoft Windows

TP Desktop Connector Client Program
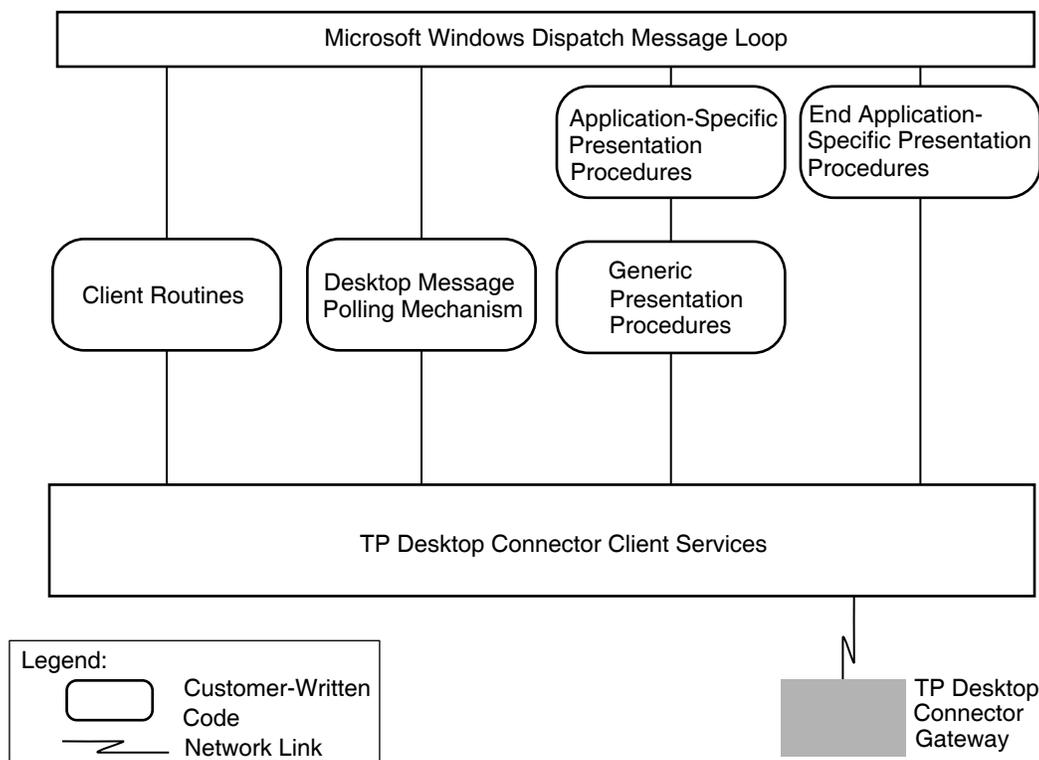
TAY-1004A-AD

The AVERTZ.EXE program includes the following functional modules:

- avertz.c

    Contains the main window function and program initialization.

- avertzpp.c

  Defines all application-specific presentation procedures for the AVERTZ desktop client program.

- resvform.c

  Manages the reservation form for interaction with the user in the AVERTZ reserve task.

- session.c

  Initializes and controls multiple user sign-ins and manages windows, icons, and menus for multiple user sessions.

- disablew.c, enablew.c, recvw.c, requestw.c, sendw.c, and transw.c

  Handles the generic presentation procedures.

The desktop client program allows the user to maintain multiple sign-ins with the AVERTZ application, VR_DA_APPL. The program controls each sign-in by associating session context with the user sign-in data.

The desktop client-program user interface presents the following menus with which the user interacts to sign in to and out of the ACMS system, run ACMS tasks, and control sessions:

- Session menu

  Signs the user in to and out of the ACMS system.

- Rental menu

  Lets the user start tasks.

- Select menu

  Allows the user to activate a session through a list of active sessions rather than by selecting an icon.

Program-defined icons represent sessions. The user can double click on an icon to bring up the form for a specific session. The icons are also visual clues to active and inactive sessions.

5.3.2 AVERTZ Component Processing Flow

The avertz.c module contains the Windows message-processing loop, program initialization, and main window function. The following important functions are performed:

- MainWndProc

  Processes Windows messages for the main window.

- client_init

  Sets up the control mechanism, a timer, for soliciting TP Desktop Connector messages.

- InitApplication

  Initializes the desktop client program and sets up window processing.

- InitInstance

  Creates the main window.

The session.c module contains the following session creation and control functions:

- NewSession

  Creates a new session, signs the user in to the ACMS system with the acmsdi_sign_in service, and adds the session to the list of active sessions.

- NewSession_Complete

  Completes processing of the NewSession function and updates the user interface. Called when the acmsdi_sign_in service completes.

- ExitSession and ExitSession_Complete

  Ends a session by calling the acmsdi_sign_out service and cleans up session data structures.

- SessionTask_Complete

  Updates the user interface accordingly. Called when a task for a given session completes.

The New command in the session menu presents a dialog box to enable the user to sign in to the ACMS system, thereby creating a new AVERTZ session. After the user is signed in to the ACMS system, the desktop client program maintains session information to track the association between a form or forms session and the corresponding ACMS submitter identification. The program also activates other menus to allow the signed-in user to select ACMS tasks. A session can have only one active task at any given time, but a user can sign in to the ACMS system many times, allowing multiple tasks to run simultaneously.

Individual modules handle the generic presentation procedures. For example, transw.c handles the acmsdi_transceive presentation procedure called from the ACMS system, validates the parameters, and dispatches control to the appropriate application-specific presentation procedure.

The avertzpp.c module handles all application-specific presentation procedures for the AVERTZ desktop client program. Each presentation procedure comprises an initial routine (for example, Trans_List2_List3) and a completion routine (for example, End_Trans_List2_List3). The generic presentation procedures call the initial routine (the first part of the presentation procedure), which generally displays the data from the workspaces sent by the ACMS system and returns. When the user signals completion (done entering data) for this presentation procedure, the corresponding completion routine is called. The completion routine collects the user-entered data and sends it to the ACMS system to complete processing of the presentation procedure.

The resvform.c module manages the reservation form for the reserve task, including field validation and initialization. The reserve routine handles Windows messages for the dialog boxes. (The reserve routine is the dialog function that Windows calls whenever the user interacts with the reservation form.) When the user clicks on the OK or cancel button in the reservation form, the reserve routine dispatches the completion routine of the current presentation procedure.

Chapter 6 describes this flow in a Motif environment. See Figure 6–3, Figure 6–4, Figure 6–5, and Figure 6–6.

5.4 Writing Client Procedures Using Nonblocking Services

Procedures using nonblocking services must be structured as described in the following sections.

5.4.1 Calling Nonblocking Services

Nonblocking forms of the services have the following differences from blocking forms:

- The routine calling the TP Desktop Connector client service supplies the address of a completion routine, the context, and the completion status in the service call.

  When the TP Desktop Connector call-completion message is received from the desktop gateway, the TP Desktop Connector client service calls the desktop client program completion routine.

- The TP Desktop Connector client service returns control to the desktop client program after a request is sent to the TP Desktop Connector gateway.

  This return allows the desktop client program to yield control to Windows for event processing to continue.

- Because the nonblocking services return control before a request completes, the desktop client program must not release storage for service argument data until after it calls the completion routine.

  If the calling routine returns before the service completes, the routine must not use volatile memory for service arguments. For example, in the C language, volatile memory includes local (automatic) variables and the arguments passed on routine calls.

Example 5–1 shows the calls in the session.c module to the acmsdi_sign_in service that creates a new session, and to the companion completion routine NewSession_Complete that completes the service. The desktop client program calls the message dispatcher to check for pending replies or requests from the TP Desktop Connector gateway. When a completion message for a client service arrives, the message dispatcher calls the completion routine in the desktop client program.

Example 5–1 Nonblocking Service Call and Completion Routine

```
BOOL FAR PASCAL NewSession(
            HWND hDlg,      /* window handle of the dialog box */
            WORD message,   /* type of message                */
            WORD wParam,    /* message-specific information    */
            LONG lParam)
{
  .
  .
  .
                status = acmsdi_sign_in(
                        session_ptr->node,
                        session_ptr->username,
                        session_ptr->password,
                        (long) 0,
                        session_ptr->submitter_id,
                        &session_ptr->completion_status,
                        NewSession_Complete,                 1
                        (void *) session_ptr);

                if (status == ACMSDI_PENDING)   2
                {
  .
  .
  .
void NewSession_Complete(void *call_context)   3
```

Example 5–1 (Cont.) Nonblocking Service Call and Completion Routine

```
{
  session_type *session_ptr;
  List          session_node;
  char          session_button_title[MAX_STRING_LENGTH];
  int           x;
  HWND          SessionIcon;
  int           i;

  /* Get Session Node For This Sign In Completion */

  session_ptr = (session_type *) call_context;

  /*
  **    Check ACMS Return Status
  **  - If failure occurred, delete session from session list and
  **    put up a message.
  */

  if (session_ptr->completion_status != ACMSDI_NORMAL)
  {
    DisplayDesktopErrorMessage(session_ptr, session_ptr->completion_status);
   .
   .
   .
```

The user triggers the call to the acmsdi_sign_in service by selecting the
OK button on the New Session dialog box. The numbers in Example 5–1
correspond to the following explanations:

1 The call specifies the completion routine address.

   Specifying the completion routine NewSession_Complete indicates a
   nonblocking service.

2 The desktop client program checks for ACMSDI_PENDING to ensure that
   the call is sent to the ACMS system.

   The user at this point is not signed in to the ACMS system. If a status
   other than ACMSDI_PENDING is returned, the completion routine is not
   called.

3 When the sign-in completes, the completion message arrives and the
   desktop client services invoke the completion routine, NewSession_
   Complete.

After the acmsdi_sign_in service returns ACMSDI_PENDING, the desktop
client program receives a **submitter identification** that is used on subsequent
calls. If a nonblocking call to a TP Desktop Connector service routine returns
a status code other than ACMSDI_PENDING, the completion routine for that
call is not invoked.

5.4.2 Setting Up Polling

In a nonblocking environment, the desktop client program must initiate a
control mechanism to poll for pending ACMS messages. To set up polling, the
desktop client program does the following:

- Activates a control mechanism when the desktop client program starts up

- Has the control mechanism call the acmsdi_dispatch_message service
  periodically

The acmsdi_dispatch_message service polls for messages from the TP Desktop
Connector gateway and calls the appropriate customer-supplied completion
routine or presentation procedure, depending on the type of TP Desktop
Connector message received. Example 5–2 shows the sample coding sequence
from the avertz.c module.

Example 5–2 Setting Up Polling

```
int PASCAL WinMain
{
   .
   .
   .
    client_init;   1
   .
   .
   .
long FAR PASCAL MainWndProc
  switch (message)
      {
      case WM_TIMER:

          if (wParam == DESKTOP_MESSAGE_TIMER)
          {
           acmsdi_dispatch_message;   2
          }

          break;
```

(continued on next page)

Example 5–2 (Cont.) Setting Up Polling

```
    .
    .
    .
       }
    .
    .
    .
void client_init(void)
{
    SetTimer(
        hWndMainWindow,
        DESKTOP_MESSAGE_TIMER,   3
        MESSAGE_CHECK_FREQUENCY,
        NULL);
}
```

Example 5–2 shows the following steps from the module avertz.c:

1   The function client_init establishes the control mechanism (see **3**).

2   Windows dispatches the event to the desktop client program.

The acmsdi_dispatch_message routine polls for gateway messages to be passed to the desktop client program.

3   The routine SetTimer specifies the main window to receive the WM_TIMER messages.

The desktop client program controls the polling interval with the constant MESSAGE_CHECK_FREQUENCY.

To notify the desktop client program that a TP Desktop Connector message from the gateway is pending, the acmsdi_dispatch_message service calls a customer-supplied completion routine or a generic presentation procedure.

5.4.3 Establishing Session Context

In a nonblocking environment, saving session context globally serves two purposes. First, it saves the local data for reuse. Second, it allows the desktop client program to coordinate with message passing between the desktop system and the ACMS system when there are multiple sign-ins. Also, saving session context globally gives you a convenient place to store session-related user interface data such as form IDs and icon IDs.

In the AVERTZ desktop client program, a user can sign in to a ACMS system in one window and initiate a task from another window. The AVERTZ desktop client program establishes context and maintains the context as session data across service calls and presentation procedures. Example 5–3 shows the definition of data in the session.h file that keeps track of session context.

Example 5–3 AVERTZ Session Context

```
extern enum request_type {
            NO_OUTSTANDING_TASK,
            TASK_IN_PROGRESS,
            SEND_CTRL_RESV_LIST,
            SEND_VR_CONTROL_WKSP,
            TRANS_LIST_3_LIST_2,
            TRANS_LIST_5_LIST_6,
            TRANS_LIST_7_LIST_6,
            TRANS_LIST_8_LIST_9,
            TRANS_VRH_RCLIST_VRH_RESV_LIST,
            TRANS_VR_CTRL_WKSP_LIST_D,
            TRANS_LIST_E_LIST_F,
            TRANS_LIST_F_LIST_F,
            TRANS_LIST_G_LIST_H};
  .
  .
  .
typedef struct {
            enum request_type   request_id;
            char                *receive_control_text;
            long                *receive_control_text_count;
            ACMSDI_FORM_RECORD *receive_record;
        } exchange_request_type;

typedef struct {
            ACMSDI_SUBMITTER_ID    *submitter_id;              1
            int                     session_id;
            ACMSDI_CALL_ID         *call_id;                   2
            char                    node[MAX_NODE_LENGTH];
            char                    username[MAX_USERNAME_LENGTH];
            char                    password[MAX_PASSWORD_LENGTH];
            char                    print_file[20];
            HWND                    session_icon;
            HWND                    resv_form;
            HWND                    vehicle_form;
            HWND                    billing_form;
```

(continued on next page)

Example 5–3 (Cont.) AVERTZ Session Context

```
        HWND                    message_window;
        exchange_request_type   *current_exchange_request; 3
        int                     completion_status;         4
        char                    task_status_message[80];   5
    } session_type;

extern session_type *init_session_list;
```

The following context data is required for a session:

1 submitter ID   Returned from the ACMS system at sign-in time

2 call ID   Returned by the acmsdi_call_task service

3 current exchange request   Needed if the program uses the same form for
                                       multiple exchange steps

4 completion status   Updated by the TP Desktop Connector client service
                                       when the task completes

5 task status message   Updated by the TP Desktop Connector client service
                                       when the task completes

The session_type structure enables the desktop client program to access data
related to a sign-in session when a Windows operation occurs. The variables
submitter_id, *call_id*, *completion_status*, and *task_status_message* are allocated
by the desktop client program and updated by TP Desktop Connector client
services. The value of the completion_status is updated just before the TP
Desktop Connector client services call the completion routine.

The session context structure is also a useful place to maintain window handles
of user interface objects related to a session. In the sample, the session context
includes window handles for the session's icon, the session's menu entry in the
select menu, and window handles for all the forms related to that session.

The AVERTZ.EXE program passes the session context to the acmsdi_call_task
service as the *call_context* parameter. Whenever the TP Desktop Connector
client services call a task completion routine or a presentation procedure on
behalf of an active ACMS task, the session context is passed to the desktop
client program. For example, you can use a session context to determine which
form in the application to update with data from an incoming presentation
procedure. Example 5–4 shows an example in the avertz.c code where the
session context is passed.

Example 5–4 Context Passed to Desktop Client Program

```
.
.
.
 status = acmsdi_call_task(
                current_session_ptr->submitter_id,
                NULL,
                "VR_RESERVE_TASK",
                AVERTZ_APPLICATION_NAME,
                NULL,
                current_session_ptr->task_status_message,
                0,
                NULL,
                current_session_ptr->call_id,
                &(current_session_ptr->completion_status),
                SessionTask_Complete,
                (void *) current_session_ptr);   1
.
.
.
```

The parameter at **1** specifies a session context to be passed. The desktop
client program can use that context to determine which form to deal with.
The session context is useful for determining which presentation procedure is
ending and which workspaces are affected (see Section 5.5).

When a presentation procedure later starts as a result of the ACMS task
executing, the session context is passed back to the desktop client program as
shown in the transw.c code in Example 5–5.

Example 5–5 Call Context Returned with Presentation Procedure

```
long int acmsdi_transceive(ACMSDI_FORMS_SESSION_ID *session_id,
                      :
                      ACMSDI_CALL_ID *call_id,
                      void *call_context,
                      ACMSDI_FORM_RECORD *send_records,
                      ACMSDI_FORM_RECORD *recv_records )
{
```

Example 5–5 (Cont.) Call Context Returned with Presentation Procedure

```
session_type *session_ptr = (session_type *) call_context;
  .
  .
  .
      save_sessions_PP_data_ptrs(
              session_ptr,
              recv_ctl_text,
              recv_ctl_text_count,
              recv_records);
      sts = Trans_List3_List2 (
              session_ptr,
              send_ctl_text, /** VR_SENDCTRL_WKSP **/
              send_records[0].data_record, /** VR_CONTROL_WKSP **/
              send_records[1].data_record, /** VR_SITES_WKSP **/
              recv_records[0].data_record, /** VR_SITES_WKSP **/
              recv_records[1].data_record, /** VR_RESERVATIONS_WKSP **/
              recv_records[2].data_record, /** VR_CUSTOMERS_WKSP **/
              recv_records[3].data_record /** VR_CONTROL_WKSP **/
                  );
  .
  .
  .
```

The code in Example 5–6 shows how session context is used to establish context
for the user interface when the user selects a session icon. The session_type
structure contains the information about the form to display for that submitter.

Example 5–6 Session Context Handling for the User Interface

```
void select_new_session(
                session_type *new_session)
{
  session_type *former_current_session_ptr = current_session_ptr;

  current_session_ptr = new_session;

 /*
 ** Hide any forms that are displayed for the former current session
 */

  if (former_current_session_ptr != NULL)
    close_session(former_current_session_ptr);
```

Example 5–6 (Cont.) Session Context Handling for the User Interface

```
/*
**  Redraw the icons of the former and new current session
**  (The WM_DRAWITEM case in MainWindowProc will redraw
**    the icon buttons when & if their select state changes.
**    Therefore, here we turn the select state OFF for the former
**    current session, and turn the select state ON for the
**    new current session.
*/
 if (former_current_session_ptr != NULL)
    SendMessage(former_current_session_ptr->session_icon,
              BM_SETSTATE, FALSE, NULL);

 SendMessage(current_session_ptr->session_icon,
            BM_SETSTATE, TRUE, NULL);


/*
**      Determine which menus, menu items must be enabled and disabled
*/
 if (former_current_session_ptr != NULL)
   UncheckSessionInSelectMenu(former_current_session_ptr->session_id);

 CheckSessionInSelectMenu(current_session_ptr->session_id);

 if ((current_session_ptr->
        current_exchange_request)->
          request_id == NO_OUTSTANDING_TASK)
 {
     EnableSessionExit;
     EnableRentalMenu;
     DisableSearchMenu;
 }

 else
 {
     DisableSessionExit;
     DisableRentalMenu;
     if ((current_session_ptr->
       current_exchange_request)->
         request_id == TRANS_LIST_3_LIST_2)
```

(continued on next page)

Example 5–6 (Cont.) Session Context Handling for the User Interface

```
        EnableSearchMenu;
    else
        DisableSearchMenu;
  }
}
```

5.4.4 Writing a Call to Other Nonblocking Services

A call to the nonblocking acmsdi_call_task or acmsdi_sign_out service must
follow the rules described for other nonblocking services (see Section 5.4.1).
The calling routine specifies the submitter identification returned from the
acmsdi_sign_in service.

The acmsdi_call_task service returns a **call identification** and call context
that are used in any completion routine (see the ***HP TP Desktop Connector
for ACMS*** *Client Services Reference Manual*), presentation procedure, or
acmsdi_complete_pp service call.

5.4.5 Canceling Active Tasks

TP Desktop Connector allows client programs, written with nonblocking
services, to cancel active tasks running on the gateway node. Being able to
cancel active tasks allows you to create applications that provide a CANCEL
function for the user. The main advantage of being able to cancel a task is
to permit the user to work on other applications, if the response from the
gateway is not immediate. For example, if the user starts a transaction on the
database, you can display three buttons in the dialog box:

- OK — To start the transaction

- ABORT — To abort the dialog without starting any task

- CANCEL — To cancel the task after it has been started

These features are available with through the portable API client services
acmsdi_cancel service. See ***HP TP Desktop Connector for ACMS*** *Client
Services Reference Manual* for a description of this client service.

You cannot use a cancel service in exchange steps. If you call a cancel during a
presentation procedure, TP Desktop Connector returns the message "ACMSDI_
EXCHACTV". If you issue a cancel while another cancel is already in progress,
TP Desktop Connector returns the message "ACMSDI_CANCELACTV". The
cancel completion routine is guaranteed to be called before the task completion
routine.

5.5 Writing Nonblocking Presentation Procedures

Writing a presentation procedure in a nonblocking environment differs from writing presentation procedures in a blocking environment. A nonblocking presentation procedure does the following:

- Performs its processing and yields control to Windows without the user having signaled completion.

  In a blocking environment, the desktop client program waits for completion.

- Signals its completion and passes completion status to the desktop gateway using a TP Desktop Connector service, acmsdi_complete_pp.

  In a blocking environment, the desktop client program can wait for task completion status.

In a nonblocking environment, presentation procedures are generally divided as follows:

- Initial routine that displays data on the screen.

- Processing that releases control to Windows so that the user can interact with the form.

- Completion routine that gathers the user-entered data upon the completion signal and calls the acmsdi_complete_pp service to pass status and data back to the TP Desktop Connector gateway.

Typically, the initial and completion routines are separate so that data can be obtained from the user. If user action is not required, such as in a stub routine, the initial routine can call the acmsdi_complete_pp service, and the completion routine is not necessary.

Example 5–7 shows pseudocode from several modules in the AVERTZ sample desktop client program to indicate the flow of processing a presentation procedure.

Example 5–7 Nonblocking Presentation Procedure Pseudocode

```
long int acmsdi_transceive(ACMSDI_FORMS_SESSION_ID *session_id, 1
                   char *send_record_id,
                   long send_record_count,
                   char *recv_record_id,
                   long recv_record_count,
                   char *recv_ctl_text,
                   long *recv_ctl_text_count,
```

Example 5–7 (Cont.) Nonblocking Presentation Procedure Pseudocode

```
                    char *send_ctl_text,
                    long send_ctl_text_count,
                    short timeout,
                    ACMSDI_CALL_ID *call_id,
                    void *call_context,
                    ACMSDI_FORM_RECORD *send_records,
                    ACMSDI_FORM_RECORD *recv_records )
{
 session_type *session_ptr = (session_type *) call_context;   2
  .
  .
  .
   if ((0 == strcmp (send_record_id, "LIST_3")) &&
       (0 == strcmp (recv_record_id, "LIST_2")))
  .
  .
  .
      /*
      **  Save Pointers To Exchange Step's Receive Data
      **  And Call Presentation Procedure
      */

      save_sessions_PP_data_ptrs(    3
              session_ptr,
              recv_ctl_text,
              recv_ctl_text_count,
              recv_records);

      sts = Trans_List3_List2 (    4
              session_ptr,
              send_ctl_text, /** VR_SENDCTRL_WKSP **/
              send_records[0].data_record, /** VR_CONTROL_WKSP **/
              send_records[1].data_record, /** VR_SITES_WKSP **/
              recv_records[0].data_record, /** VR_SITES_WKSP **/
              recv_records[1].data_record, /** VR_RESERVATIONS_WKSP **/
              recv_records[2].data_record, /** VR_CUSTOMERS_WKSP **/
              recv_records[3].data_record /** VR_CONTROL_WKSP **/
            );
    }
int Trans_List3_List2 (    5
          session_type            *session_ptr, ... )
{
  .
  .
  .
   enable_initial_fields(session_ptr->resv_form);
```

Example 5–7 (Cont.) Nonblocking Presentation Procedure Pseudocode

```
   enable_resv_push_buttons(session_ptr->resv_form);

   return(FORMS_NORMAL);    6
}
   .
   .
   .
    return (sts);    7
}
```

The code shown in Example 5–7 does the following:

1   The desktop client program is called at the acmsdi_transceive interface.

The reserve task in the AVERTZ sample application triggers an exchange step. Refer to the reserve task code shown in Example 4–4 for places where presentation procedures are called. The acmsdi_transceive generic presentation procedure defined in transw.c is called through the polling mechanism. The workspaces from the ACMS system are passed to the AVERTZ desktop client program.

2   The procedure establishes the session context by doing a type conversion on the call context value.

3   The program saves the addresses of the write and modify arguments.

The acmsdi_transceive function parses the workspaces to determine which application-specific presentation procedure to call.

4   The generic presentation procedure acmsdi_transceive calls the application-specific presentation procedure Trans_List3_List2.

5   The Trans_List3_List2 function defined in the avertzpp.c module gets control to solicit data from the user.

The routine creates the dialog box and displays the data passed in the workspaces. (The function reserve defined in resvform.c, not shown, paints the screen and receives control from the Windows message queue.)

6   Control passes to the acmsdi_transceive function.

After the data to display is sent to the dialog box, control is returned to the generic function.

7   Control passes to Windows.

The desktop client program allows message processing for other activities to continue.

At this point, the user can enter data in the dialog box and the desktop client program no longer has control.

To signal that data entry is complete and to pass status back to the TP Desktop Connector gateway, the user clicks on the OK button in the dialog box some time after the desktop client program yields control to Windows. Example 5–8 shows the processing in the resvform.c module when the user either signals completion or cancels the operation.

Example 5–8 Presentation Procedure Completion Pseudocode

```
      case WM_COMMAND:                       /* message: received a command */

          switch (wParam) {
            case IDOK    :      1
            case IDCANCEL :
              {
                exchange_request_type *exchange_request =
                 current_session_ptr->current_exchange_request;

                switch (exchange_request->request_id) {     2
  .
  .
  .
                  case TRANS_LIST_3_LIST_2           :   3
                        End_Trans_List3_List2(current_session_ptr,
                                             wParam);
                        return(TRUE);
  .
  .
  .
void End_Trans_List3_List2(          4
          session_type *session_ptr,
          WORD          button_pressed)

{
  receive_record    = (session_ptr->current_exchange_request)->receive_record;

  sites_wksp        = (vr_sites_wksp *) (receive_record[0].data_record);
  reservations_wksp = (vr_reservations_wksp *) (receive_record[1].data_record);
  customers_wksp    = (vr_customers_wksp *) (receive_record[2].data_record);
  control_wksp      = (vr_control_wksp *) (receive_record[3].data_record);
  .
  .
  .
```

Example 5–8 (Cont.) Presentation Procedure Completion Pseudocode

```
        acmsdi_complete_pp(session_ptr->call_id, FORMS_NORMAL);    5

        MessageBox(
          session_ptr->resv_form,
          "Reservation Data Has Been Submitted.\nWait For Return Data  ...  ",
          " ",
          MB_OK | MB_ICONINFORMATION);
    .
    .
    .
  return;    6
}
```

The code in Example 5–8 does the following:

1 Windows passes a message to the desktop client program when the user clicks on the OK button in the dialog box.

The reserve function in resvform.c parses the Windows command and determines which presentation procedure completion routine to call based on the current exchange request saved for the current session.

2 The session context determines which presentation procedure is completing.

3 The desktop client program calls the second part of the presentation procedure.

Based on the pending exchange request, the reserve function calls the routine End_Trans_List3_List2 in avertzpp.c.

4 The End_Trans_List3_List2 routine gains control.

Using the pointers to workspaces saved when the presentation procedure began, the routine collects new data entered from the dialog box. The session context is passed along to the application-specific presentation procedure completion routine. The completion routine can determine which workspaces to update and which call identification to pass to the acmsdi_complete_pp service.

5   The End_Trans_List3_List2 routine sends the updated arguments to the TP Desktop Connector gateway.

To send a reply to the TP Desktop Connector gateway, the routine calls the acmsdi_complete_pp service, specifying an *OpenVMS* completion status and the call identification that the TP Desktop Connector client services passed into the program.

6   Control returns to Windows.

5.6 Writing Memory Allocation Routines

The desktop client program allocates and manages memory while coexisting with the TP Desktop Connector client services and other software on the desktop platform.

The TP Desktop Connector client services use the malloc and calloc functions. However, Windows converts these functions to the *LocalAlloc* function, which is limited to the 64K bytes in the local heap.

Because messages sent to and received from the TP Desktop Connector Gateway for ACMS can be quite large (depending on the size of workspaces), message buffer allocation can fail, if the desktop client program is already using a substantial portion of the local heap. (Remember that the local heap is also used for the stack, static data, and global data items.)

Message buffers associated with presentation procedures persist for the duration of the call, including a period where control returns to Windows while the user enters data. The use of malloc/calloc with large workspaces can severely restrict the amount of memory available to the desktop client program in the local heap.

The TP Desktop Connector client services permit you to specify your own allocation and free routines for message buffers. These are passed in to TP Desktop Connector client services using the options parameter on the acmsdi_sign_in call by specifying the ACMSDI_OPT_MALLOC_ROUTINE and ACMSDI_OPT_FREE_ROUTINE options (see *HP TP Desktop Connector for ACMS Client Services Reference Manual*).

Do not use GlobalAlloc in the TP Desktop Connector client services memory management routines.

5.7 Building and Debugging Windows Desktop Client Programs

The build procedure for nonblocking Windows client programs is identical to the procedure for blocking client programs. See the build instructions in Section 4.7.

5.8 Debugging the Nonblocking Desktop Client Program with Tasks

First, debug the presentation code on the desktop system. Use the second monitor recommended for Windows environments. When the presentation code runs, debug the desktop client program with the ACMS software. Follow these guidelines:

- For debugging tasks, see Section 3.6.3.

- For using logging in troubleshooting, see **HP TP Desktop Connector for ACMS** *Gateway Management Guide*.

5.8.1 Using a Debugger to Step Through the Microsoft Windows Sample Application

To get a better feel for the flow of the nonblocking Desktop application, use a debugger to step through the Microsoft Windows sample provided on the kit.

Set a breakpoint at the following functions:

- avertz.c

  — Search for "IDM_NEW_SESSION", and set a breakpoint at the first C statement after it.

  — Search for "IDM_END_SESSION", and set a breakpoint at the first C statement after it.

  — Search for "IDM_CREATE_RESV", and set a breakpoint at the first C statement after it.

- session.c

  — NewSession

  — NewSession_Complete

  — SessionTask_Complete

  — ExitSession

  — ExitSession_Complete

- enablew.c

    — acmsdi_enable

- disablew.c

    — acmsdi_disable

- transw.c

    — acmsdi_transceive

- avertzpp.c

    — Trans_List3_List2

    — End_Trans_List3_List2

    Optionally, you can also set breakpoints to the other presentation procedures that can be invoked as part of the RESERVE task, for example:

    — Trans_List5_List6

    — End_Trans_List5_List6

    — Trans_List7_List6

    — End_Trans_List7_List6

    — Trans_List8_List9

    — End_Trans_List8_List9

- resvform.c

    — Search for "IDOK" in the dialog function, Reserve. Set a breakpoint at the first C statement after it.

6

Using Portable API Extensions for OSF/Motif

This chapter describes how to develop nonblocking desktop client programs for event-driven, multitasking environments such as OSF/Motif and X Windows.

The sample described in this chapter uses X Windows and the OSF/Motif toolkit. However, the guidelines presented here apply to any X-applications, including Open Look applications. Chapter 4 gives a useful background in the Desktop services provided in the portable API. Although the focus in that chapter is on Microsoft Windows, the basic principles apply to OSF/Motif and X Windows as well. For information on designing your application with nonblocking services, see Chapter 2.

For information on configuring network transports for any of the client platforms, see **HP TP Desktop Connector for ACMS** *Gateway Management Guide*.

6.1 Event-Driven Processing

A Motif desktop client program must yield control to the X Windows system as quickly as possible so that other event processing can be handled in a timely way. The following TP Desktop Connector nonblocking services provide methods to facilitate yielding control:

- acmsdi_dispatch_message

  Receives requests and responses from the gateway and dispatches the corresponding procedures.

- acmsdi_complete_pp

  Sends responses to the gateway when presentation procedures complete.

Figure 6–1 shows the sequence of processing in an event-driven desktop client program.

Figure 6–1 Event-Driven Desktop Client Program Processing
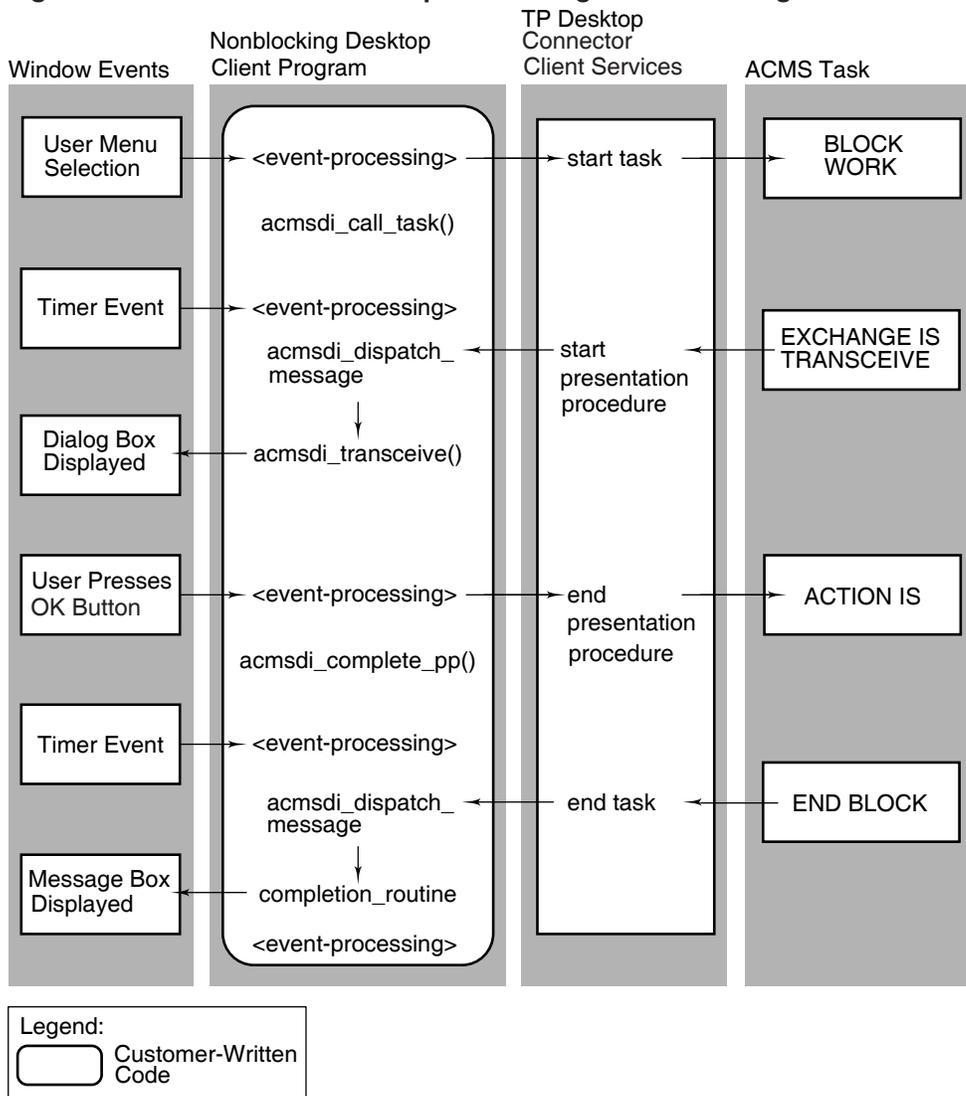
When the user selects a task from a menu, the desktop client program calls the acmsdi_call_task service to initiate an ACMS task. In the nonblocking environment, you specify nonblocking parameters in the call (see Section 9.5). For example, the address of a completion routine is a nonblocking parameter that you can include in the desktop client program. The TP Desktop Connector client service calls the completion routine when the TP Desktop Connector Gateway for ACMS completes the task. Because the *completion routine* parameter is specified in the call, the TP Desktop Connector client service is treated like a nonblocking service and returns control to the desktop client program immediately after the start task request is sent to the TP Desktop Connector gateway. The desktop client program does not wait for the gateway to request ACMS software to start a task in an ACMS application. Instead, the desktop client program returns control to XtMainLoop for more event processing.

The TP Desktop Connector client service also returns a call identification that the desktop client program supplies to the acmsdi_complete_pp service. The services use the call identification to associate an active call with a submitter. Note that TP Desktop Connector supports, at most, one active task for each submitter.

Because of an exchange step in the I/O task, the ACMS system starts a transceive operation. The gateway sends a message to the desktop client program to start a presentation procedure.

The acmsdi_dispatch_message service runs as part of the polling mechanism established by the desktop client program. The acmsdi_dispatch_message service polls for the message and, when the message is received, calls the customer-written presentation procedure acmsdi_transceive. The acmsdi_transceive routine runs, saves pointers to the workspaces, displays a dialog box, and returns control to X Windows without the user having to signal completion.

The user then has an opportunity to edit or add data to a form in the context of an exchange step. When the user later clicks on OK to signal completion, the desktop client program uses the saved workspace pointers to store the user-entered data, and calls the acmsdi_complete_pp service to send the workspaces and response status to the gateway. (The desktop client program does not wait for a response from the gateway. It returns control to XtMainLoop as soon as possible.) Meanwhile, the gateway passes to the ACMS task the workspaces and the status from the completion of the presentation procedure.

When the task completes on the ACMS system, the gateway sends the end task message to the desktop system. The TP Desktop Connector client services dispatch the end task message by calling a completion routine in the desktop client program. This completion routine is dispatched by the polling mechanism and acmsdi_dispatch_message.

The gateway sends a message to the desktop client program under two conditions:

- To signal the completion of a TP Desktop Connector client service

  The desktop client program supplies the address of completion routines in the nonblocking forms of the acmsdi_sign_in, acmsdi_call_task, and acmsdi_sign_out services.

- To process an exchange step in the task

  Client services define entry points for presentation procedures that correspond to all possible exchange steps. The desktop client program supplies the code that handles these presentation procedures.

The TP Desktop Connector client services dispatch the gateway messages to the desktop client program by the polling mechanism established.

6.2 Guidelines for Developing X Windows Desktop Client Programs

The following list summarizes general requirements and guidelines for developing X Windows nonblocking desktop client programs:

- Write procedures to use nonblocking TP Desktop Connector client services (see Section 6.4).

  Organize functions so that every call to a TP Desktop Connector service or a customer-written presentation procedure has a parallel completion routine.

- Set up polling within the desktop client program (see Section 6.4.2).

  Set up a mechanism to retrieve desktop messages. Use this mechanism to periodically gain control and call the acmsdi_dispatch_message service to retrieve messages from the gateway.

- Establish session context in the desktop client program (see Section 6.4.3).

  Set up structures for handling multiple sessions. Certain data must be maintained as context for a current session and inactive sessions.

- Write presentation procedures to coexist in the nonblocking environment (see Section 6.6).

  You must complete all presentation procedures using the acmsdi_complete_pp service.

- Convert data as required between the desktop system and the ACMS system (see Section 4.1.3).

Optionally, you can do the following when developing a desktop client program:

- Write an action routine to perform version checking.

  The module version.c shows a stub version-checking routine (see Section 4.3).

The remainder of the chapter explains the guidelines using code from the AVERTZ sample desktop client program.

6.3 AVERTZ Sample Desktop Client Program for X Windows

The Motif AVERTZ sample desktop client program is written in C and follows the standards and practices outlined in the *OSF/Motif Style Guide*. The program uses a main window, icons, and dialog boxes to interact with the user.

6.3.1 AVERTZ Components for X Windows

A nonblocking X Windows desktop client program has some of the same high-level components as a blocking Desktop client (see Figure 6–2). Note the addition of the polling mechanism as well as the application-specific presentation procedures, which are split into two parts.

The m_avertz.exe program includes the following functional modules:

- m_avertz.c

  Contains the main window support routines and the program initialization.

- m_avertzpp.c

  Defines all application-specific presentation procedures for the AVERTZ desktop client program.

- m_resvform.c

  Manages the reservation form for interaction with the user in the AVERTZ reserve task.

- m_session.c

  Initializes and controls multiple user sign-ins and manages windows, icons, and menus for multiple user sessions.

- m_disable.c, m_enable.c, m_receive.c, m_request.c, m_send.c, and m_transceive.c

  Handle the generic presentation procedures.

Figure 6–2 TP Desktop Connector Sample Components for X Windows

TP Desktop Connector Client Program
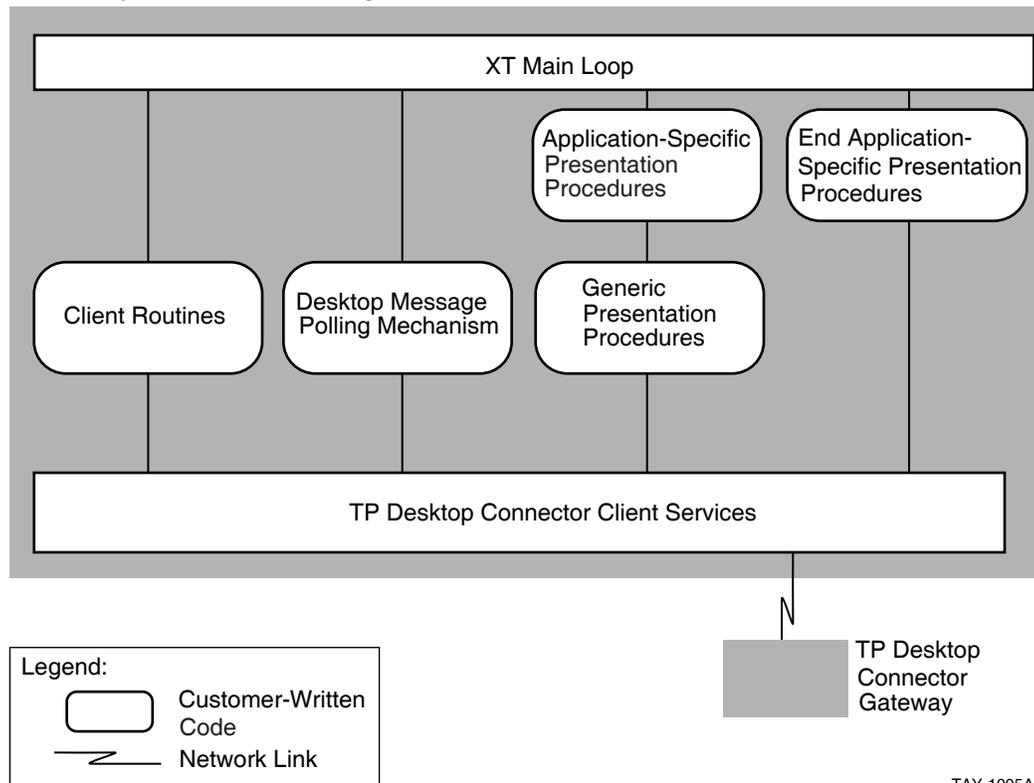
TAY-1005A-AD

The desktop client program allows the user to maintain multiple sign-ins with the AVERTZ application, VR_DA_APPL. The program controls each sign-in by associating session context with the user sign-in data.

The desktop client program user interface presents menus with which the user interacts to sign in to and out of the ACMS system, run ACMS tasks, and control sessions:

- Session menu

  Signs the user in to and out of the ACMS system.

- Rental menu

  Lets the user start tasks.

- Search menu

  Allows the user to select site information from a list of available sites.

- Select menu

  Allows the user to activate a session through a list of active sessions rather than by selecting an icon.

Program-defined icons represent sessions. The user can double click on an icon to bring up forms associated with a specific session. The icons are also visual clues to active and inactive sessions.

6.3.2 AVERTZ Component Processing Flow

The m_avertz.c module contains the program initialization, XtMainLoop, and support for the main window. The following important functions are performed:

- main

  Initializes the Motif toolkit, registers callbacks, fetches and displays the main window, and polls for X-events.

- client_init

  Sets up the polling mechanism for soliciting TP Desktop Connector messages.

- CreateReservation

  Invoked when the user selects the Create Reservation menu item. Starts the Reserve task in the ACMS application, VR_DA_APPL, by calling acmsdi_call_task.

The m_session.c module contains the following session creation and control functions:

- NewSession

  Invoked when the user presses the OK button in the session login dialog box. Creates a new session, signs the user in to the ACMS system with the acmsdi_sign_in service, and adds the session to the list of active sessions.

- NewSession_Complete

  Called when the acmsdi_sign_in service completes. Completes processing of the NewSession function and updates the user interface accordingly.

- ExitSession and ExitSessionComplete

  Ends a session by calling the acmsdi_sign_out service and cleaning up session data structures.

- SessionTask_Complete

  Called when a task for a given session completes. Updates the user interface accordingly.

The New command in the session menu presents a dialog box to enable the user to sign in to the ACMS system, thereby creating a new AVERTZ session. After the user is signed in to the ACMS system, the desktop client program maintains session information to track, for example, the association between the form (or forms) of a given session and the corresponding ACMS submitter identification. The program also activates other menus to allow the signed-in user to select ACMS tasks. A session can have only one active task at any given time, but a user can sign in to the ACMS system many times, allowing multiple tasks to run simultaneously.

Individual modules handle the generic presentation procedures. For example, m_transceive.c handles the acmsdi_transceive presentation procedure called from the ACMS system, validates the parameters, and dispatches control to the appropriate application-specific presentation procedure.
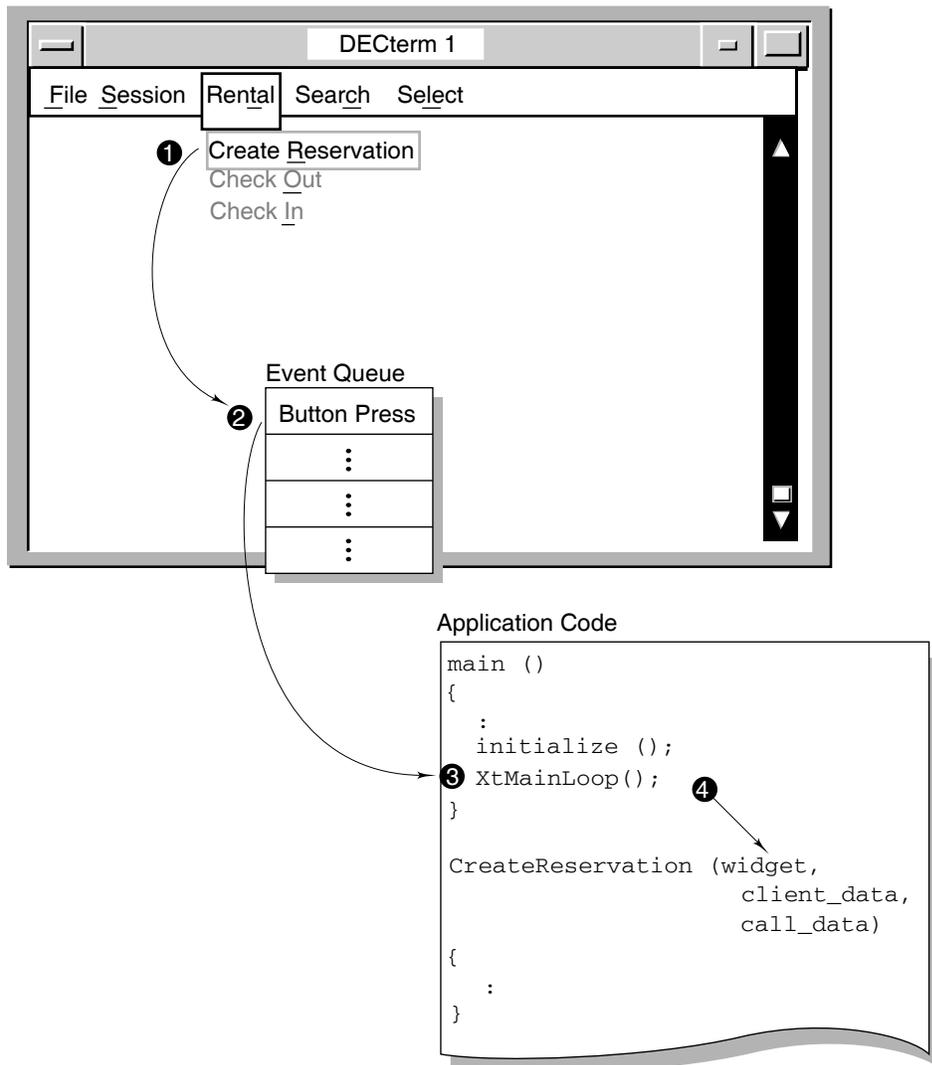
The m_avertzpp.c module handles all application-specific presentation procedures for the AVERTZ desktop client program. Each presentation procedure comprises an initial routine, for example, Trans_List3_List2, and a completion routine, for example, End_Trans_List3_List2. The generic presentation procedures call the initial routine (the first part of the presentation procedure), which generally displays the data from the workspaces sent by the ACMS system and returns. Control returns to XtMainLoop so that the user can enter and receive data.

When the user signals completion (done entering data) for this presentation procedure, the corresponding completion routine is called. It collects the user-entered data and sends it to the ACMS system to complete processing of the presentation procedure.

The m_resvform.c module manages the reservation form for the reserve task, including field validation and initialization. When the user clicks on the OK or cancel button in the reservation form, the ResvFormExchangeComplete( ) routine dispatches the completion routine of the current presentation procedure. The following series of figures (Figure 6–3, Figure 6–4, Figure 6–5, and Figure 6–6) present the flow of a nonblocking service.

Figure 6–3 shows the user selecting the Create Reservation task from the Rental menu.

Figure 6–3 User Selects a Task


```
main ()
{
   :
  initialize ();
  XtMainLoop();
}

CreateReservation (widget,
                   client_data,
                   call_data)
{
   :
}
```

TAY-0307-AD

The following callouts describe the actions in Figure 6–3:

1 The end user selects the CreateReservation command.

2 The X Windows system indicates this event by sending a ButtonPress event to the X Windows event queue.

3 In the application's main( ) module, XtMainLoop continuously looks at the X-event queue for incoming events. XtMainLoop pulls the ButtonPress event off the event queue.

4 XtMainLoop determines that the widget callback registered for this event is the function CreateReservation( ). XtMainLoop then dispatches the CreateReservation( ) function in response to the event.

Figure 6–4 shows the flow of code in the nonblocking service.

Figure 6–4   Nonblocking Service

Client Application Code

Server Application Code

❷ acmsdi_call_task(Submitter_id, ❸ ───────→ The VR_RESERVE_TASK Begins
```
                    ⋮
                  "VR_RESERVE_TASK",
                    ⋮
                  call_id,
                  completion status,
                  session_task_complete
                  current_session_ptr};
}
```
End Task

❶ client_init(app_context)
```
{
   XtAppAddTimeOut(app_context,
                  message_frequency,
                  check_for_messages,
                  app_context);
}
```

TP Desktop Connector
Message Queue

```
Check_for_messages()
{
```
❻ acmsdi_dispatch_message();
```
   XtAppAddTimeOut(app_context,
                  message_frequency,
                  check_for_messages,
                  app_context);

 SessionTask_complete (Context)
{
  :
}
```

End Task ❹

❺

TAY-0305-AD

The following callouts describe the flow of code in Figure 6–4:

1 To support nonblocking services, the application sets up a mechanism to call acmsdi_dispatch_message at periodic intervals. This is normally done at the application's initialization time. The client_init( ) function sets up the mechanism that periodically calls check_for_messages( ).

2 Inside the CreateReservation( ) function, the application initiates the VR_RESERVE_TASK task by making a call to acmsdi_call_task. To indicate that this is a nonblocking call, the completion routine parameter is set to the function pointer, SessionTask_Complete. The application passes the current_session_ptr as context to be received back when the completion routine SessionTask_Complete is called.

3 When acmsdi_call_task is invoked, TP Desktop Connector client services send a Start Task message to the ACMS application. When the ACMS application receives the message, VR_RESERVE_TASK begins.

4 When VR_RESERVE_TASK completes, TP Desktop Connector services send an End Task message to the TP Desktop Connector message queue on the desktop system.

5 Meanwhile, check_for_messages( ) is being invoked at regular intervals. When it calls acmsdi_dispatch_message, TP Desktop Connector client services looks for incoming messages on the message queue. When it pulls the End Task message off the message queue, it determines that the completion routine that was specified for this service is SessionTask_Complete( ).

6 Then acmsdi_dispatch_message dispatches the function SessionTask_Complete( ) in response to the message. The context specified on the acmsdi_call_task call is passed back as a parameter. The application can use this application-specific context, for example, to have the user interface reflect that the task completed.

Figure 6–5 shows the I/O processing of a nonblocking service.

Figure 6–5 I/O Processing for a Nonblocking Service/Part 1

Client Application Code

```
Main()
{
   :
  XtMainLoop;
}

check_for_messages()
{
  acmsdi_dispatch_message();
  return (FALSE);    ❸
}

acmsdi_transceive( :
                 send_record_id,
                 recv_record_id,
                  :
                 context,
                  :
                 send_wksps,
                 recv_wksps,
                  :
                  )
{
  session_ptr=(session_type *)context;
  if ((send_record_id=="List3") !!
      (recv_record_id=="List2"))  ❹
  {
    save_recv_wksp_ptrs_in_session_context(
                               session_ptr,
                               recv_wksps);

    trans_List3_List2(session_ptr,  ❺
                     send_wksps);
  }
  :
}

trans_List3_List2(sesion_ptr,
                 send_wksps)
{
  session_ptr_current_exchange_step=transList3List2;
  display_form_with_data_sent_from_ACMS
                          (send_wksps);   ❻
  enable_fields_related_to_trans_list3_list2();
  /* Now control will return to XtMainLoop */
}
```

Server Application Code

```
The task begins

❶ Exchange Step

Transceive List3 List2

The Exchange Step
  has Completed
   ⋮
   ⋮
The task ends
```

TP Desktop Connector
Message Queue

❷

| Start transceive |
| :⋮ |
| :⋮ |
| :⋮ |

TAY-0326-AD

The following callouts describe the I/O processing flow for a task that has just been invoked:

1 In Figure 6–5, the first exchange step of VR_RESERVE_TASK is Transceive List3 List2. When this exchange step is encountered, an

acmsdi_transceive message is sent to the TP Desktop Connector Message Queue on the desktop system.

2 Meanwhile, acmsdi_dispatch_message( ) is being called at regular intervals.

3 When TP Desktop Connector pulls the start transceive message off the message queue, it dispatches the presentation procedure, acmsdi_ transceive( ).

4 The acmsdi_transceive routine looks at the values of the send_record_id and the receive_record_id to determine which specific transceive procedure is being requested. (In Figure 6–5, Transceive_List3_List2 is the specific transceive procedure specified in the ACMS task, so the send_record_id is List 3 and the receive_record_id is List 2.) Before leaving the acmsdi_ transceive routine, the application must save the pointers to the receive workspaces. These pointers are needed later when the application is ready to load the receive workspaces with the user's data input.

5 The specific transceive routine Trans_List3_List2( ) is called.

6 The specific transceive routine puts up a form and enables the fields associated with this transceive exchange. When this routine completes, control returns to XtMainLoop( ).

Figure 6–6 shows the form displayed by the Trans_List3_List2( ) routine.

Figure 6–6 I/O Processing for a Nonblocking Service/Part 2
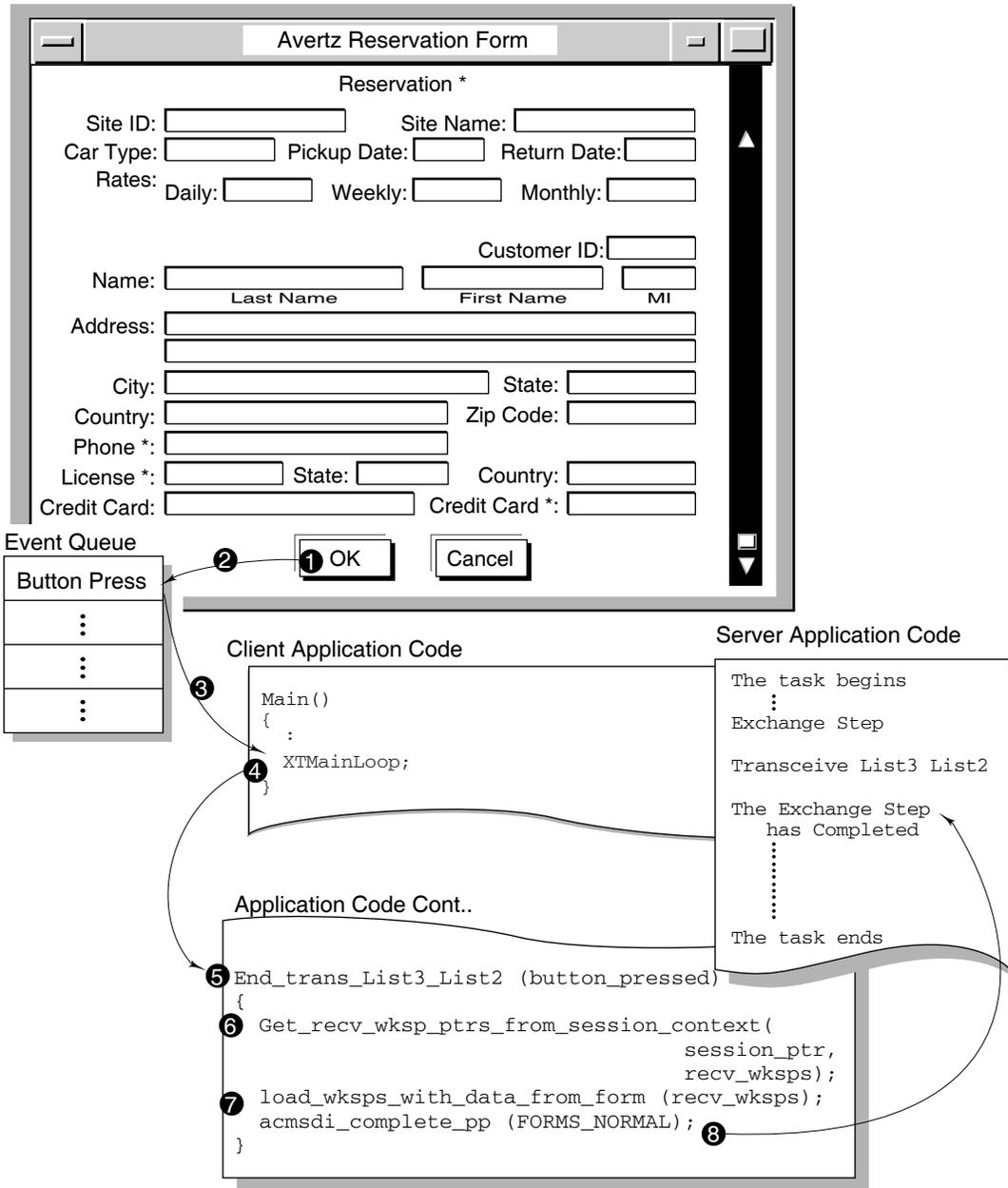
TAY-0327-AD

The following callouts described the processing in Figure 6–6:

1   The user enters data into various fields of the form, and when finished entering data, the user clicks on the OK button.

2   X Windows puts this ButtonPress event on the X-event queue.

3   XtMainLoop( ) pulls the ButtonPress event off the event queue.

4   XtMainLoop( ) determines that the callback registered for this event is in the function End_Trans_List3_List2( ).

5   An intermediate routine called ResvFormExchange Complete is invoked. It determines which specific presentation procedure is completing.

6   In End_Trans_List3_List2, the pointers to the receive workspaces are retrieved (the application stored them away at the beginning of the transceive processing). See Figure 6–5, Callout 4. The user's data input is retrieved from the form and loaded into the receive workspaces.

7   When the receive workspaces are ready to be sent back to ACMS, the application notifies the Desktop client services by issuing a call to acmsdi_complete_pp( ).

8   TP Desktop Connector then sends the workspaces and a completion status back to the ACMS application.

6.4 Writing Client Procedures Using Nonblocking Services

The following sections describe how to structure procedures that use nonblocking services.

6.4.1 Calling Nonblocking Services

Nonblocking forms of the services have the following differences from blocking forms:

•   The routine calling the desktop service supplies the address of a completion routine, the context, and the completion status in the service call.

    When TP Desktop Connector receives the call-completion message from the gateway, TP Desktop Connector calls the desktop client program completion routine.

•   The TP Desktop Connector client service returns control to the desktop client program after sending a request to the gateway.

    This return allows the desktop client program to yield control to X Windows, so that event processing can continue.

- Because the nonblocking services return control before a request completes, the desktop client program must not release storage for service argument data until after it calls the completion routine.

  If the calling routine returns before the service completes, the routine must not use volatile memory for service arguments. For example, in the C language, volatile memory includes local (automatic) variables and the arguments passed on routine calls.

The desktop client program periodically calls acmsdi_dispatch_message( ) to check for pending replies or requests from the gateway. When the client service completion message arrives, the acmsdi_dispatch_message( ) calls the completion routine in the desktop client program.

Example 6–1 shows the calls in the m_session.c module to the acmsdi_sign_in service that creates a new session, and to the companion completion routine NewSession_Complete that completes the service.

Example 6–1 Nonblocking Service Call and Completion Routine

```
extern void NewSession(
      Widget    widget,
      int      *client_data,
            XtPointer call_data)
{

   .
   .
   .

   status = acmsdi_sign_in(
               session_ptr->node,
               session_ptr->username,
               session_ptr->password,
               (long) 0,
               session_ptr->submitter_id,
               &session_ptr->completion_status,
               NewSession_Complete,                1
               (void *) session_ptr);

    if (status == ACMSDI_PENDING)    2
    {
   .
   .
   .
extern void NewSession_Complete(void *call_context)    3
```

Example 6–1 (Cont.) Nonblocking Service Call and Completion Routine

```
{
  session_type *session_ptr;
  Widget        session_icon;
  int           i;

  /* Get Session Node For This Sign In Completion */

  session_ptr = (session_type *) call_context;

  /*
  ** Check ACMS Return Status
  **  - If failure occurred, put up the error message and
  ** delete session from session list.
  */

  if (session_ptr->completion_status != ACMSDI_NORMAL)
  {
    DisplayDesktopErrorMessage(session_ptr, session_ptr->completion_status);

    delete_session_context(session_ptr);

  }
  else
  {

   .
   .
   .
```

The acmsdi_sign_in service call is triggered by the user selecting the OK
button in the New Session dialog box:

1 The call specifies the completion routine address.

   Specifying the completion routine NewSession_Complete indicates a
   nonblocking service.

2 The desktop client program checks for ACMSDI_PENDING to ensure that
   the call is sent to the ACMS system.

   The user at this point is *not* signed in to the ACMS system. If a status
   other than ACMSDI_PENDING is returned, the completion routine is not
   called.

3 When the sign-in completes, the completion message arrives and the
   desktop service calls the completion routine to handle the status.

After the acmsdi_sign_in service returns ACMSDI_PENDING, the desktop client program receives a **submitter identification** that is used on subsequent calls. If a nonblocking call to a TP Desktop Connector client service routine returns a status code other than ACMSDI_PENDING, the completion routine for that call is not invoked.

6.4.2 Setting Up Polling

In a nonblocking environment, the desktop client program must initiate a control mechanism to poll for pending ACMS messages. To set up polling, the desktop client program does the following:

- Activates a control mechanism when the desktop client program starts up

- Has the control mechanism call the acmsdi_dispatch_message service periodically

The acmsdi_dispatch_message service polls for messages from the gateway and calls the appropriate customer-supplied completion routine or presentation procedure, depending on the type of TP Desktop Connector message received.

Example 6–2 illustrates how to set up polling using a timer event.

Example 6–2 Setting Up Polling Using a Timer Event

```
/*****************************************************************************
    FUNCTION:    client_init

    SUMMARY:     Initiates timer mechanism to handle incoming Desktop messages.
*****************************************************************************/

static void client_init(XtAppContext app_context) 1
{

 /*
 **  Initiate a timer event to call check_for_messages() after a
 **  specified delay (MESSAGE_CHECK_FREQUENCY). The check_for_messages()
 **  routine will then call acmsdi_dispatch_message to process any
 **  incoming Desktop messages (this includes End-Of-Desktop-Service
 **  messages and presentation procedure invocation messages).
 */

 DesktopMessageTimer = XtAppAddTimeOut(app_context,
                                       MESSAGE_CHECK_FREQUENCY,
                                       check_for_messages,
                                       app_context);

}
```

Example 6–2 (Cont.) Setting Up Polling Using a Timer Event

```
/*****************************************************************************
    FUNCTION:   check_for_messages

    SUMMARY:    Checks for incoming Desktop messages using the
                TP Desktop Connector client service, acmsdi_dispatch_message().

    COMMENTS:   This routine is invoked as a result of a timer event
                and it is responsible for setting the timer to call
                itself again.
*****************************************************************************/
static void check_for_messages(XtAppContext app_context) 2
{
  int status;

  /*
  ** Dispatch any incoming TP Desktop Connector messages:
  **  If there is a service completion message, a completion
  **  routine for that service will be dispatched. If there
  **  is a start exchange step message, a corresponding
  **  presentation procedure will be dispatched. If there
  **  are no pending Desktop messages, acmsdi_dispatch_message()
  **  will return immediately.
  */

  status = acmsdi_dispatch_message(); 3

  if (status != ACMSDI_NORMAL)
    printf("Warning! Error status code (%d) received on ACMSDI_DISPATCH_MESSAGE.\n",
           status);

  /*
  ** Reset the timer that calls check_for_messages()
  */

  DesktopMessageTimer = XtAppAddTimeOut(app_context,
                                        MESSAGE_CHECK_FREQUENCY,
                                        check_for_messages,
                                        app_context);

}
```

Example 6–2 shows the following steps from the module m_avertz.c.

1 The function client_init establishes the control mechanism (see **3**).

2 The client_init routine initiates a timer-driven callback routine, check_for_messages, which X Windows invokes after a specified period of time.

3 The acmsdi_dispatch_message routine polls for gateway messages to be passed to the desktop client program.

To notify the desktop client program that a TP Desktop Connector message from the gateway is pending, the acmsdi_dispatch_message service calls a customer-supplied completion routine or a generic presentation procedure.

6.4.3 Establishing Session Context

In a nonblocking environment, saving session context globally serves two purposes. First, it saves the local data for reuse. Second, it allows the desktop client program to deal with message passing between the desktop system and the ACMS system when there are single or multiple sign-ins. Also, saving session context globally gives you a convenient place to store session-related user interface data such as form IDs and icon IDs.

In the AVERTZ desktop client program, a user can sign in to a ACMS system in one window and initiate a task from another window. The AVERTZ desktop client program establishes context and maintains the context as session data across service calls and presentation procedures. Example 6–3 shows the definition of data in the session.h file that keeps track of sign-in context.

Example 6–3 AVERTZ Session Context

```
extern enum request_type {
            NO_OUTSTANDING_TASK,
            TASK_IN_PROGRESS,
            SEND_CTRL_RESV_LIST,
            SEND_VR_CONTROL_WKSP,
            TRANS_LIST_3_LIST_2,
            TRANS_LIST_5_LIST_6,
            TRANS_LIST_7_LIST_6,
            TRANS_LIST_8_LIST_9,
            TRANS_VRH_RCLIST_VRH_RESV_LIST,
            TRANS_VR_CTRL_WKSP_LIST_D,
            TRANS_LIST_E_LIST_F,
            TRANS_LIST_F_LIST_F,
            TRANS_LIST_G_LIST_H};
   .
   .
   .
typedef struct {
            enum request_type   request_id;
            char                *receive_control_text;
```

Example 6–3 (Cont.) AVERTZ Session Context

```
          long               *receive_control_text_count;
          ACMSDI_FORM_RECORD *receive_record;
      } exchange_request_type;

typedef struct {
          ACMSDI_SUBMITTER_ID    *submitter_id;  1
          int                     session_id;
          ACMSDI_CALL_ID         *call_id;        2
          char                    node[MAX_NODE_LENGTH];
          char                    username[MAX_USERNAME_LENGTH];
          char                    password[MAX_PASSWORD_LENGTH];
          Widget                  session_icon;
          Widget                  icon_pixmap;
          Widget                  session_menu_item;

          Widget                  resv_form;
          Widget                  resv_form_ids[MAX_RESV_FORM_IDS];
          Widget                  vehicle_form;
          Widget                  billing_form;
          exchange_request_type  *current_exchange_request;  3
          int                     completion_status;          4
          char                    task_status_message[80];   5
          List                    message_boxes;
      } session_type;
```

The following context data is required for a session:

1	submitter ID	Returned from the ACMS system at sign-in time
2	call ID	Returned by the acmsdi_call_task service
3	current exchange request	Needed if the program uses the same form for different exchange steps
4	completion status	Updated by the TP Desktop Connector client service when the task completes
5	task status message	Updated by the TP Desktop Connector client service when the task completes

The session_type structure enables the desktop client program to access data related to a sign-in session when a Windows operation occurs. The variables *submitter_id*, *call_id*, *completion_status*, and *task_status_message* are allocated by the desktop client program and updated by TP Desktop Connector client services. The value of the completion_status is updated just before the TP Desktop Connector client services call the completion routine.

The session context structure can also be a useful place to maintain widget IDs of user interface objects related to a session. In the sample, the session context includes widget IDs for the session icon, the session's menu entry in the select menu, widget IDs for all the forms associated with the session, and all the fields in those forms, as well as all message boxes.

The m_avertz program passes the session context to the acmsdi_call_task service as the *call_context* parameter. Whenever the TP Desktop Connector client services call a task completion routine or a presentation procedure on behalf of an active ACMS task, the session context is passed to the desktop client program. For example, you can use a session context to determine which form in an application to update with data from an incoming presentation procedure. Example 6–4 shows an example in the m_avertz.c code where the session context is passed.

Example 6–4 Context Passed to Desktop Client Program

```
.
.
.
 status = acmsdi_call_task(
              current_session_ptr->submitter_id,
              NULL,
              "VR_RESERVE_TASK",
              AVERTZ_APPLICATION_NAME,
              NULL,
              current_session_ptr->task_status_message,
              0,
              NULL,
              current_session_ptr->call_id,
              &(current_session_ptr->completion_status),
              SessionTask_Complete,
              (void *) current_session_ptr);   1
.
.
.
```

The parameter at **1** specifies the session context to be passed. When a presentation procedure later starts as a result of the ACMS task executing, the session context is passed back to the desktop client program, as shown in the m_transceive.c code in Example 6–5.

The desktop client program can use that context to determine which form to deal with. The session context is also useful for determining which presentation procedure is in progress or is ending, and which workspaces are affected (see Section 5.5).

Example 6–5 Call Context Returned with Presentation Procedure

```
long int acmsdi_transceive(ACMSDI_FORMS_SESSION_ID *session_id,
   .
   .
   .
                           ACMSDI_CALL_ID *call_id,
                           void *call_context,
                           ACMSDI_FORM_RECORD *send_records,
                           ACMSDI_FORM_RECORD *recv_records )
{
session_type *session_ptr = (session_type *) call_context;
   .
   .
   .
```

The sample m_session.c code in Example 6–6 shows how session context is
used to establish context for the user interface when the user selects a session
icon.

The session_type structure contains the information about the form to display
for that submitter.

Example 6–6 Session Context Handling for the User Interface

```
extern void select_new_session(
               session_type  *new_session)
{
  session_type *former_current_session_ptr = current_session_ptr;

  current_session_ptr = new_session;

  /*
  ** If the session that was selected was the current session anyway,
  ** then there is no need to make any updates to the UI, so return.
  */

  if (current_session_ptr == former_current_session_ptr)
    return;
```

Example 6–6 (Cont.) Session Context Handling for the User Interface

```
/*
** Hide any forms that are displayed for the former current session
*/
 if (former_current_session_ptr != NULL)
   close_session(former_current_session_ptr);

/*
**  Redraw the icons of the former and new current session
*/
 if (former_current_session_ptr != NULL)
   redraw_session_icon(former_current_session_ptr);

 redraw_session_icon(current_session_ptr);
/*
** The current session is designated in the 'Select' menu
** with a shaded diamond.
**
** Remove the shaded diamond from the menu entry of the former
** current session and add the shaded diamond to the menu entry
** of the new current session.
*/

if (former_current_session_ptr != NULL)
 UncheckSessionInSelectMenu (former_current_session_ptr->session_menu_item);

 CheckSessionInSelectMenu (current_session_ptr->session_menu_item);

/*
**  Determine which menus, menu items must be enabled and disabled
*/
 if ((current_session_ptr->
        current_exchange_request)->
          request_id == NO_OUTSTANDING_TASK)
 {
     EnableSessionExit();
     EnableRentalMenu();
     DisableSearchMenu();
 }
 else
```

Example 6–6 (Cont.) Session Context Handling for the User Interface

```
{
    DisableSessionExit();
    DisableRentalMenu();
    if ((current_session_ptr->
      current_exchange_request)->request_id == TRANS_LIST_3_LIST_2)

        EnableSearchMenu();
    else
        DisableSearchMenu();
}
}
```

6.4.4 Writing a Call to Other Nonblocking Services

A call to the nonblocking acmsdi_call_task or acmsdi_sign_out service must
follow the rules described for other nonblocking services (see Section 5.4.1).
The calling routine specifies the submitter identification returned from
the acmsdi_sign_in service. The acmsdi_call_task service returns a **call
identification** and call context that are used in any completion routine (see
HP TP Desktop Connector for ACMS *Client Services Reference Manual*),
presentation procedure, or acmsdi_complete_pp service call.

6.5 Canceling Tasks

TP Desktop Connector allows client programs, written with nonblocking
services, to cancel active tasks running on the gateway node. Being able to
cancel active tasks allows you to create applications that provide a CANCEL
function for the user. The main advantage of being able to cancel a task is
to permit the user to work on other applications, if the response from the
gateway is not immediate. For example, if the user starts a transaction on the
database, you can display three buttons in the dialog box:

- OK — To start the transaction

- ABORT — To abort the dialog without starting any task

- CANCEL — To cancel the task after it has been started

This functionality is available through the portable API client service acmsdi_
cancel. See ***HP TP Desktop Connector for ACMS*** *Client Services Reference
Manual* for a description of this client service.

You cannot use a cancel service in exchange steps. If you call a cancel during a presentation procedure, TP Desktop Connector returns the message "ACMSDI_EXCHACTV". If you issue a cancel while another cancel is already in progress, TP Desktop Connector returns the message "ACMSDI_CANCELACTV". The cancel completion routine is guaranteed to be called before the task completion routine.

6.6 Writing Nonblocking Presentation Procedures

Writing a presentation procedure in a nonblocking environment differs from writing presentation procedures in a blocking environment. A nonblocking presentation procedure does the following:

- Performs its processing and yields control to X Windows before the user signals completion

  In a blocking environment, the desktop client program waits for completion.

- Signals its completion and passes completion status to the desktop gateway using a TP Desktop Connector service, acmsdi_complete_pp

  In a blocking environment, the desktop client program can wait for task completion status.

In a nonblocking environment, presentation procedures are generally divided as follows:

- Initial routine that displays data on the screen

- Processing that releases control to X Windows so that the user can interact with the form

- Completion routine that gathers the user-entered data upon the completion signal and calls the acmsdi_complete_pp service to pass status and data back to the gateway

Typically, the initial and completion routines are separate, so that data can be obtained from the user. If user action is not required as, for example, in a stub routine, the initial routine can call the acmsdi_complete_pp service, and the completion routine is not necessary.

Example 6–7 shows pseudocode from several modules in the AVERTZ sample desktop client program to indicate the flow of processing a presentation procedure.

Example 6–7 Nonblocking Presentation Procedure Pseudocode

```
long int acmsdi_transceive(ACMSDI_FORMS_SESSION_ID *session_id,  1
                        char *send_record_id,
                        long send_record_count,
                        char *recv_record_id,
                        long recv_record_count,
                        char *recv_ctl_text,
                        long *recv_ctl_text_count,
                        char *send_ctl_text,
                        long send_ctl_text_count,
                        short timeout,
                        ACMSDI_CALL_ID *call_id,
                        void *call_context,
                        ACMSDI_FORM_RECORD *send_records,
                        ACMSDI_FORM_RECORD *recv_records )
{
 session_type *session_ptr = (session_type *) call_context;  2
   .
   .
   .
   if ((0 == strcmp (send_record_id, "LIST_3")) &&
       (0 == strcmp (recv_record_id, "LIST_2")))
   .
   .
   .
       /*
       **  Save Pointers To Exchange Step's Receive Data
       **  And Call Presentation Procedure
       */

       save_sessions_PP_data_ptrs(   3
               session_ptr,
               recv_ctl_text,
               recv_ctl_text_count,
               recv_records);

       sts = Trans_List3_List2 (   4
               session_ptr,
               send_ctl_text, /** VR_SENDCTRL_WKSP **/
               send_records[0].data_record, /** VR_CONTROL_WKSP **/
               send_records[1].data_record, /** VR_SITES_WKSP **/
               recv_records[0].data_record, /** VR_SITES_WKSP **/
               recv_records[1].data_record, /** VR_RESERVATIONS_WKSP **/
               recv_records[2].data_record, /** VR_CUSTOMERS_WKSP **/
               recv_records[3].data_record /** VR_CONTROL_WKSP **/
             );
```

Example 6–7 (Cont.) Nonblocking Presentation Procedure Pseudocode

```
   }
int Trans_List3_List2 (   5
          session_type          *session_ptr,...)
{
   .
   .
   .
   enable_initial_fields(session_ptr->resv_form);

   enable_resv_push_buttons(session_ptr->resv_form);

   return(FORMS_NORMAL);   6
}
   .
   .
   .
    return (ts);   7
}
```

The code shown in Example 6–7 does the following:

1   The desktop client program is called at the acmsdi_transceive interface.

The reserve task in the VR_DA_APPL application triggers an exchange step. Refer to the reserve task code shown in Example 4–4 for places where presentation procedures are called. The acmsdi_transceive generic presentation procedure defined in m_transceive.c is called through the polling mechanism. The workspaces from the ACMS system are passed to the AVERTZ desktop client program.

2   The procedure establishes the session context by doing a type conversion on the call context value.

The acmsdi_transceive function parses the workspaces to determine which application-specific presentation procedure to call.

3   The program saves the addresses of the write and modify arguments.

4   The generic presentation procedure acmsdi_transceive calls the application-specific presentation procedure Trans_List3_List2.

5   The Trans_List3_List2 function defined in the m_avertzpp.c module gains control to solicit data from the user.

The routine creates the dialog box and displays the data passed in the workspaces.

6   Control returns to the acmsdi_transceive function.

After the data to display is sent to the dialog box, control is returned to the generic function.

7 Control returns to X Windows.

The desktop client program allows event processing for other activities to continue.

At this point, the user can enter data in the dialog box and the desktop client program no longer has control.

To signal that data entry is complete and to pass status back to the gateway, the user clicks on the OK button in the dialog box some time after the desktop client program yields control to X Windows. Example 6–8 shows the processing in the m_resvform.c module when the user either signals completion or cancels the operation.

Example 6–8 Presentation Procedure Completion Pseudocode

```
extern void ResvFormExchangeComplete (       1
                  Widget    widget,
              int     *client_data,
                  XtPointer call_data)

{
  int    button_pressed = *client_data;

  exchange_request_type *exchange_request =
       current_session_ptr->current_exchange_request;
  switch (exchange_request->request_id) {    2

   .
   .
   .
              case TRANS_LIST_3_LIST_2     :    3
                End_Trans_List3_List2(current_session_ptr,
                                      button_pressed);
                    break;
   .
   .
   .
}

void End_Trans_List3_List2(                    4
     session_type *session_ptr,
     int    button_pressed)
```

(continued on next page)

Example 6–8 (Cont.) Presentation Procedure Completion Pseudocode

```
{
  .
  .
  .
 receive_record    = (session_ptr->current_exchange_request)->receive_record;

 sites_wksp      = (vr_sites_wksp *) (receive_record[0].data_record);
 reservations_wksp = (vr_reservations_wksp *) (receive_record[1].data_record);
 customers_wksp    = (vr_customers_wksp *) (receive_record[2].data_record);
 control_wksp     = (vr_control_wksp *) (receive_record[3].data_record);
  .
  .
  .
 acmsdi_complete_pp(session_ptr->call_id, FORMS_NORMAL);   5

 create_session_message(
         session_ptr,
         AvertzMainWindow,
         "Reservation Data Has Been Submitted. \nWait For Return Data...",
         " ",
         XmDIALOG_WORKING,
         NULL,
         NULL);
  .
  .
  .
}                             6
```

The code in Example 6–8 does the following:

1 X Windows passes a message to the desktop client program when the user clicks on the OK button in the dialog box.

The reserve function in resvform.c parses the X Windows command and determines which presentation procedure completion routine to call, based on the current exchange request saved for the current session.

2 The session context determines which presentation procedure is completing.

3 The desktop client program calls the second part of the presentation procedure.

Based on the pending exchange request, the reserve function calls the routine End_Trans_List3_List2 in m_avertzpp.c.

4 The End_Trans_List3_List2 routine gains control.

Using the pointers to workspaces saved when the presentation procedure began, the routine collects new data entered from the dialog box. The session context is passed along to the application-specific presentation procedure completion routine. The completion routine can determine which workspaces to update and which call identification to pass to the acmsdi_complete_pp service.

5 The End_Trans_List3_List2 routine sends the updated arguments to the gateway.

To send a reply to the gateway, the routine calls the acmsdi_complete_ pp service, specifying an OpenVMS completion status and the call identification that the TP Desktop Connector client services passed into the program.

6 Control returns to X Windows.

6.7 Special Handling of Workspaces for RISC Client Applications

The RISC architecture for Alpha or I64 systems requires that data references be **naturally aligned**. That is, short words (2 bytes) must be on an even byte boundary. Long words (4 bytes) must be accessed on a boundary evenly divisible by 4.

When an Alpha or an I64 Client defines a C structure, it creates padding in the structure, if necessary, to ensure that each field complies with these requirements. (The padding is not visible to you.)

An ACMS task running on either an OpenVMS Alpha or an OpenVMS I64 system, however, does not impose such restrictions on its data objects and does not pad its structures. The problem arises when data, defined on one of these machines, is transmitted across the network to the other machine, and interpreted using the same C structure definitions. In a TP Desktop Connector application, this is a concern only when ACMS workspaces are being sent (in either direction) between the client program on a RISC machine and the ACMS application (OpenVMS).

Example 6–9 shows how a sample application client program deals with incoming workspaces (or send records). Before calling the application-specific presentation procedure (Trans_List3_List2), the generic presentation procedure (acmsdi_transceive) dynamically allocates structures for all the incoming workspaces (the send_records array).

The data in the send_records array is byte copied field by field into the newly allocated workspaces (load_control_wksp( )). These workspaces are then passed to Trans_List3_List2, where their contents are used to update the display. Once Trans_List3_List2( ) returns, acmsdi_transceive( ) assumes that the workspaces it dynamically allocated are no longer needed and it frees them.

Example 6–9 OpenVMS to RISC Structure Byte Copy

```
void load_control_wksp(
        vr_control_wksp *control_wksp,
           char             *data_ptr)
{

  memcpy(&control_wksp->ctrl_key,
         data_ptr,
         sizeof(control_wksp->ctrl_key));
  data_ptr = data_ptr + sizeof(control_wksp->ctrl_key);

  memcpy(&control_wksp->current_entry,
         data_ptr,
         sizeof(control_wksp->current_entry));
  data_ptr = data_ptr + sizeof(control_wksp->current_entry);
   .
   .
   .
}
```

Example 6–9 (Cont.) OpenVMS to RISC Structure Byte Copy

```
long int acmsdi_transceive(ACMSDI_FORMS_SESSION_ID *session_id,
                     char *send_record_id,
                     long send_record_count,
                     char *recv_record_id,
                     long recv_record_count,
                     char *recv_ctl_text,
                     long *recv_ctl_text_count,
                     char *send_ctl_text,
                     long send_ctl_text_count,
                     short timeout,
                     ACMSDI_CALL_ID *call_id,
                     void *call_context,
                     ACMSDI_FORM_RECORD *send_records,
          ACMSDI_FORM_RECORD *recv_records )
{
   .
   .
   .
   if ((0 == strcmp (send_record_id, "LIST_3")) &&
       (0 == strcmp (recv_record_id, "LIST_2")))
    {
   .
   .
   .
         /*
         ** Create the send workspace structures and load their
         ** fields with the data in the send_records array.
         */

         vr_control_wksp *send_control_wksp =
              (vr_control_wksp *) malloc(sizeof(vr_control_wksp));

         vr_sites_wksp   *send_sites_wksp  =
              (vr_sites_wksp *) malloc(sizeof(vr_sites_wksp));

         load_control_wksp(send_control_wksp,
                        (char *) send_records[0].data_record);

         load_sites_wksp(send_sites_wksp,
                        (char *) send_records[1].data_record);
```

Example 6–9 (Cont.) OpenVMS to RISC Structure Byte Copy

```
sts = Trans_List3_List2 (
 session_ptr,
 send_ctl_text,               /** VR_SENDCTRL_WKSP **/
 send_control_wksp,           /** VR_CONTROL_WKSP **/
 send_sites_wksp,             /** VR_SITES_WKSP **/
 recv_records[0].data_record, /** VR_SITES_WKSP **/
 recv_records[1].data_record, /** VR_RESERVATIONS_WKSP **/
 recv_records[2].data_record, /** VR_CUSTOMERS_WKSP **/
 recv_records[3].data_record);/** VR_CONTROL_WKSP **/

        free(send_control_wksp);
 free(send_sites_wksp);
      }
 .
 .
 .
}
```

Example 6–10 shows how a client deals with outgoing workspaces (or receive
records). In the sample application, the workspaces are not allocated until it is
time to pull the data off the form. In the presentation procedure's completion
routine, End_Trans_List3_List2( ), the structures for all the presentation
procedure's outgoing workspaces (receive records) are allocated. They are then
initialized to guarantee that string fields are padded with blanks.

These structures are then loaded with data retrieved from the form. When
the data retrieval is complete, the contents of the structure are byte copied
field by field to the receive_records array (unload_control_wksp( )). (This
receive_records array is the same receive_records array that was originally
passed into the acmsdi_transceive( ) routine.) Finally, the application calls
acmsdi_complete_pp to send the contents of the receive_records array back to
the ACMS application.

Example 6–10 RISC to OpenVMS Structure Byte Copy

```
void unload_control_wksp(
    vr_control_wksp *control_wksp,
    char            *data_ptr)
{
```

Example 6–10 (Cont.) RISC to OpenVMS Structure Byte Copy

```
  memcpy(data_ptr,
         control_wksp->ctrl_key,
         sizeof(control_wksp->ctrl_key));
  data_ptr = data_ptr + sizeof(control_wksp->ctrl_key);

  memcpy(data_ptr,
         &(control_wksp->current_entry),
         sizeof(control_wksp->current_entry));
  data_ptr = data_ptr + sizeof(control_wksp->current_entry);

   .
   .
   .
}
void End_Trans_List3_List2(
     session_type *session_ptr,
     int     button_pressed)
{
  receive_record    = (session_ptr->current_exchange_request)->receive_record;

  sites_wksp     = (vr_sites_wksp *) malloc(sizeof(vr_sites_wksp));
  reservations_wksp =
      (vr_reservations_wksp *)  malloc(sizeof(vr_reservations_wksp));
  customers_wksp    = (vr_customers_wksp *) malloc(sizeof(vr_customers_wksp));
  control_wksp      = (vr_control_wksp *) malloc(sizeof(vr_customers_wksp));
  if( ((int) sites_wksp == NULL)          ||
      ((int) reservations_wksp == NULL) ||
      ((int) customers_wksp == NULL)      ||
      ((int) control_wksp == NULL))
  {
    DisplayWarningBox(
        AvertzMainWindow,
        "Application Has Run Out Of Memory. \n\nUnable To Continue.",
        "WARNING!!");

    return;
  }

  /*
  ** Initialize all the workspace fields (set all the
  ** characters in the character arrays to blanks, etc.).
  */
  init_sites_wksp(sites_wksp);
  init_reservations_wksp(reservations_wksp);
  init_customers_wksp(customers_wksp);
  init_control_wksp(control_wksp);
```

Example 6–10 (Cont.) RISC to OpenVMS Structure Byte Copy

```
  .
  .
  .
 data_missing = get_initial_fields(session_ptr->resv_form,
      sites_wksp,
      reservations_wksp,
      customers_wksp);

  .
  .
  .
 if (!data_missing)
 {
  .
  .
  .

    /*
    ** Move the data collected in the workspace structures
    ** to the location of the original receive records
    */

    unload_sites_wksp(sites_wksp, receive_record[0].data_record);
    unload_reservations_wksp(reservations_wksp,
                             receive_record[1].data_record);
    unload_customers_wksp(customers_wksp, receive_record[2].data_record);
    unload_control_wksp(control_wksp, receive_record[3].data_record);

  .
  .
  .

    acmsdi_complete_pp(session_ptr->call_id, FORMS_NORMAL);

 }
 free(sites_wksp);
 free(reservations_wksp);
 free(customers_wksp);
 free(control_wksp);
}
```

6.8 Writing Memory Allocation Routines

The desktop client program allocates and manages memory while coexisting with the TP Desktop Connector client services and other software on the desktop platform. By default, the TP Desktop Connector client services use malloc( ) and free( ). However, you do not need to use these services for environments other than Windows. The TP Desktop Connector client services permit you to specify your own allocation and free routines for message

buffers. These are passed in to TP Desktop Connector client services using the options parameter on the acmsdi_sign_in call by specifying the ACMSDI_OPT_ MALLOC_ROUTINE and ACMSDI_OPT_FREE_ROUTINE options (see *HP TP Desktop Connector for ACMS* *Client Services Reference Manual*).

6.9 Building and Debugging Motif Desktop Client Programs

For guidelines for building TP Desktop Connector client programs for OpenVMS and Tru64 UNIX, see the makefiles in the appropriate directory for your platform.

6.9.1 Debugging the Nonblocking Desktop Client Program with Tasks

Debug the presentation code on the desktop system. When the presentation code runs, debug the desktop client program with the ACMS software. Follow these guidelines:

- For debugging tasks, see Section 3.6.3.

- For using logging in troubleshooting (see *HP TP Desktop Connector for ACMS* *Gateway Management Guide*).

6.9.2 Using a Debugger to Step Through the Motif Sample Application

To get a better feel for the flow of a nonblocking Desktop application, use a debugger to step through the Motif sample provided on the kit. Set breakpoints in the various files at the following functions:

- m_avertz.c

    — CreateReservation

- m_session.c

    — NewSession

    — NewSession_Complete

    — SessionTask_Complete

    — ExitSession

    — ExitSession_Complete

- m_enable.c

    — acmsdi_enable

- m_disable.c

    — acmsdi_disable

- m_transceive.c

  — acmsdi_transceive

- m_avertzpp.c

  — Trans_List3_List2

  — End_Trans_List3_List2

  Optionally, you can also set breakpoints to the other presentation procedures that can be invoked as part of the RESERVE task, for example:

  — Trans_List5_List6

  — End_Trans_List5_List6

  — Trans_List7_List6

  — End_Trans_List7_List6

  — Trans_List8_List9

  — End_Trans_List8_List9

- m_resvform.c

  — ResvFormExchangeComplete

7

Forced Nonblocking Extension to the Portable API

This chapter describes how to use the forced nonblocking feature of TP Desktop Connector client services to create applications using presentation packages such as Visual Basic.

The topics in this chapter are:

- Benefits of forced nonblocking
- Portable API extensions
- ACMSDI_FORM_RECORD_BIND structure
- ACMSDI_WORKSPACE_BIND structure
- Structures declared in memory
- Forced nonblocking flow of control
- Forced nonblocking sample application

7.1 Benefits of Forced Nonblocking

Certain desktop application development tools, such as Visual Basic, cannot handle ACMS exchange steps, because these tools do not support callbacks from environments such as the C language. Because these tools do not support pointer types, they cannot accept arguments that are passed by reference, such as form records. TP Desktop Connector exchange step callbacks expect the called presentation procedures to accept arguments that are passed by reference. The TP Desktop Connector portable API nonblocking execution is activated by passing the address of a completion routine to the acmsdi_call_task service.

The TP Desktop Connector portable API has been extended to support both exchange steps and nonblocking execution of task calls for development tools that do not support pointer data types, or whose memory management routines relocate data. In addition, the extension to the portable API allows support of acmsdi_cancel service, which must be issued in a nonblocking environment. The extension to the portable API provides a way for these tools to obtain a pointer (a 32-bit integer) to their workspace buffers using the acmsdi_return_pointer service.

7.2 Portable API Extensions for Forced Nonblocking

The extensions to the portable API are:

- ACMSDI_OPT_NONBLK option type

  An ACMSDI_OPTION type associated with the acmsdi_sign_in service, this option type indicates that all calls issued for the session being created are to be nonblocking, even though no completion address is supplied. The session created is known as a **forced nonblocking session**.

  When the session is a forced nonblocking session, all calls are nonblocking even though you do not specify a completion address. Forced nonblocking sessions do not require a completion address. In fact, if you specify a completion address when specifying ACMSDI_OPT_NONBLK, you cause an error condition.

- acmsdi_poll service

  This client service returns one of the values in the following table:

Table 7–1 Values Returned from acmsdi_poll

Value	Description
ACMSDI_CANCEL_DONE	Task cancel call complete.
ACMSDI_DONE	Sign-in, sign-out, or task call complete.
ACMSDI_ENABLE_EXCH	Enable exchange step has arrived.
ACMSDI_EXEC	Call still executing; no message available.
ACMSDI_READY	No call executing; no message available.
ACMSDI_RECV_EXCH	Receive exchange step has arrived.

Table 7–1 (Cont.)   Values Returned from acmsdi_poll

Value	Description
ACMSDI_REQUEST_EXCH	TDMS request exchange step has arrived.
ACMSDI_SEND_EXCH	Send exchange step has arrived.
ACMSDI_TDMS_READ_EXCH	TDMS read exchange has arrived.
ACMSDI_TDMS_WRITE_EXCH	TDMS write exchange has arrived.
ACMSDI_TRCV_EXCH	Transceive exchange step has arrived.

Use the following acmsdi_poll services instead of acmsdi_dispatch_message during forced nonblocking sessions:

- acmsdi_bind_msg service

  You can use this client service if the status value returned from acmsdi_poll is either a TDMS read or a TDMS write exchange. The client application can call an acmsdi_bind_msg service call from the TP Desktop Connector Gateway for ACMS on the host OpenVMS system. In this case, the acmsdi_poll service performs one of the following functions:

  - Acquires the prompt text, if any, associated with a TDMS read exchange.

  - Sends the message text associated with a TDMS read exchange.

  - Acquires the message text associated with a TDMS write exchange.

  If the prompt or message text is being acquired, the text is truncated when the buffer supplied is not large enough to hold the entire text. If the buffer is larger that the text being acquired, the text is left-justified in the buffer and right-filled with blank characters.

  The acmsdi_bind_msg call is optional. However, if you do not issue this call, you cannot process arguments received from the gateway or send arguments back to the gateway.

  See *HP TP Desktop Connector for ACMS* Client Services Reference Manual for the syntax of this call.

- acmsdi_complete_call service

  This client service obtains the completion arguments for acmsdi_call_task, acmsdi_sign_in, acmsdi_sign_out, and acmsdi_cancel when acmsdi_poll detects completion status for these services.

- A set of client services, one associated with each exchange step type except TDMS read and TDMS write:

    — acmsdi_bind_enable_args

    — acmsdi_bind_receive_args

    — acmsdi_bind_request_args

    — acmsdi_bind_send_args

    — acmsdi_bind_transceive_args

    These services pass pointers to arguments associated with exchange steps, except for forms records, send control text, receive control text, and forms session identifiers. Memory locations for all arguments must exist in the caller's address space. The caller passes the arguments by reference, thus passing pointers to the arguments.

    You issue these client service calls after acmsdi_poll returns a value indicating that an exchange step has arrived (except TDMS read and TDMS write). TP Desktop Connector copies the arguments to the caller's memory locations.

- acmsdi_bind_session_id service

    This client service, used during an enable exchange step, sends the forms session identifier argument to TP Desktop Connector.

- A set of client services for sending and receiving forms records and workspaces associated with exchange steps:

    — acmsdi_bind_receive_recs

        This service moves the receive forms records, including receive control text associated with exchange steps, to TP Desktop Connector. The buffers are located and copied from the caller's address space. The service passes an ACMSDI_FORM_RECORD_BIND array as an argument. Workspace pointers in ACMSDI_FORM_RECORD_BIND arrays are acquired with the acmsdi_return_pointer service.

    — acmsdi_bind_request_wksps

        This service moves workspaces associated with TDMS request exchange steps either to the caller's memory, or from the caller's memory to TP Desktop Connector. It passes an array of a type of workspace structure, an ACMSDI_WORKSPACE_BIND array, as an argument. Workspace pointers in ACMSDI_WORKSPACE_BIND arrays are acquired with the acmsdi_return_pointer service. See *HP TP Desktop Connector for ACMS* Client Services Reference Manual for more information on this structure.

— acmsdi_bind_send_recs

This service moves send forms records associated with exchange steps, including send control text, to the caller's memory. It passes an array of a type of form record structure, an ACMSDI_FORM_RECORD_BIND array, as an argument. Forms record pointers in the ACMSDI_FORM_ RECORD_BIND array are acquired with the acmsdi_return_pointer service. See *HP TP Desktop Connector for ACMS* Client Services *Reference Manual* for more information on this structure.

In a forced nonblocking session, the acmsdi_disable callback to the client application does not occur as part of the acmsdi_sign_out processing. Instead, the client application must clear the structures used by presentation procedures without being prompted by the acmsdi_disable callback. In addition, in a forced nonblocking session, the acmsdi_check_version callback to the client does not occur. Instead, TP Desktop Connector adds the form version to the argument list acquired by the application using the acmsdi_bind_enable_ args service. The client application can do the version checking during enable exchange step processing.

See *HP TP Desktop Connector for ACMS* Client Services Reference Manual for the syntax and description of these client services for the forced nonblocking environment.

7.3 Forced Nonblocking Programming Considerations

The following sections discuss programming considerations when using forced nonblocking mode.

7.3.1 Establishing a Forced Nonblocking Session

To establish a forced nonblocking session, you request nonblocking calls without specifying a completion address with the acmsdi_sign_in service call. Instead, you specify the ACMSDI_OPT_NONBLK option. If the sign-in succeeds, all calls for the session are nonblocking. Follow these steps to initialize call options to request forced nonblocking calls. (Note the example code, written in Visual Basic syntax, has been altered to make it more readable.)

1. Declare the options array:

```
Static options(2) As ACMSDI_OPTION
```

2. Specify forced nonblocking calls in the options array:

```
options(0).option = ACMSDI_OPT_NONBLK
options(1).option = ACMSDI_OPT_END_LIST
```

3. Pass the options array as the options parameter on acmsdi_sign_in:

```
status = acmsdi_sign_in(submitter_node$,    ' node to sign-in to
                        username_str$,      ' user signing in
                        password_str$,      ' user's password
                        options(0),         ' options array
                        submitter_id,       ' submitter identifier
                        ByVal 0%,           ' final completion status
                        0&,                 ' completion routine
                        ByVal 0&)           ' call context
```

The session is a nonblocking call even though the completion routine address is 0, because the options argument is set to ACMSDI_OPT_NONBLK. Also, all subsequent calls for the session are nonblocking. A successful return status is ACMSDI_PENDING, indicating that the sign-in call has been sent to the back end.

The final completion status is passed as a long integer with a value of 0, which TP Desktop Connector interprets as a null pointer. This is done because its memory location can change before the task completes. When the task completes, you acquire the final completion status with the acmsdi_complete_call service.

7.3.2 Canceling a Task from a Forced Nonblocking Session

Presentation tools, like Visual Basic, which do not support pointer types, need a mechanism that enables them to cancel a task without specifying a completion routine address. For these tools, TP Desktop Connector provides the polling service, acmsdi_poll, to detect the completion of a task cancellation.

When a client application detects the completion of a task cancellation, the completion status argument acquired with the acmsdi_complete_call service contains the final status of the task cancellation. ACMSDI reports the status of a task cancellation call before the final status of the task call.

7.3.3 Polling for Messages

The acmsdi_poll service is required in a forced nonblocking session, in place of the acmsdi_dispatch_message service. Unlike acmsdi_dispatch_message, which passes no arguments, the acmsdi_poll service passes the submitter identifier created at sign-in, and, optionally, a location where a pointer to the context of the task or cancel call can be returned.

You must issue the acmsdi_poll service for a specific submitter, because acmsdi_poll does not dispatch any messages, but returns the message type of one message received from the back end. In contrast, the acmsdi_dispatch_message service dispatches any number of messages (calls any number of completion routines or presentation procedures) with a single call from the

user. Thus, your application must issue acmsdi_poll calls for all submitters it controls.

You can use the pointer to the call context to determine to which call the returned status refers. You must declare, and pass by reference, the memory for the call context in the client application. Because it is unlikely that the client application supports pointer types, the returned pointer is seen as a 32-bit integer. Using the acmsdi_return_pointer service, you can obtain a pointer to the call context as a 32-bit integer. The client application can then compare the two pointers to identify the call. The pointer is an optional argument, therefore, you can pass a zero value indicating that no call context is returned.

7.3.4 Obtaining Completion Arguments

When acmsdi_poll detects the completion of an acmsdi_sign_in, acmsdi_sign_out, acmsdi_call_task, or acmsdi_cancel call in the forced nonblocking environment, you obtain the arguments form the backend using the acmsdi_complete_call service. The final completion status is the only argument to obtain for acmsdi_sign_in, acmsdi_sign_out, and acmsdi_cancel. For acmsdi_call_task, in addition to the final completion status, you also obtain the ACMS status message and task argument workspaces.

The third argument in the acmsdi_complete_call service is an ACMSDI_CALL_ID structure, representing the call ID of the original task call. This argument is required for calls issued to obtain acmsdi_call_task completion argument, but must be NULL for all other call types.

Obtain task argument workspaces by passing an array of ACMSDI_WORKSPACE or ACMSDI_WORKSPACE_OPT structures. You must pass the same array that you passed on the original acmsdi_call_task service call. However, because memory management routines may have relocated the buffers, you must renew the workspace pointers in the ACMSDI_WORKSPACE or ACMSDI_WORKSPACE_OPT structures using the acmsdi_return_pointer service prior to issuing acmsdi_complete_call.

The acmsdi_complete_call is required. Until you issue the acmsdi_complete_call service call, a task call is not considered complete.

The C-language prototype for acmsdi_complete_call is as follows:

```
int acmsdi_complete_call (ACMSDI_SUBMITTER_ID *subm_id,   /*read – required*/
                          int    * completion status,  /*write – required*/
                          ACMSDI_CALL_ID *call_id,     /*read – optional*/
                          char   *status_message,      /*write – optional*/
                          void   *workspace);          /*write – optional*/
```

7.4 Forced Nonblocking Exchange Step Handling

When acmsdi_poll returns a value indicating that an exchange step request
has arrived from the back end, you can issue one of six services to retrieve the
write-only arguments:

- acmsdi_bind_enable_args
- acmsdi_bind_msg
- acmsdi_bind_receive_args
- acmsdi_bind_request_args
- acmsdi_bind_send_args
- acmsdi_bind_transceive_args

For enable exchange steps, you can issue the service, acmsdi_bind_session_id,
to send the Forms Session ID to ACMS.

TP Desktop Connector does not require that you issue these service calls.
However, if you do not issue them, the client application cannot examine the
write-only arguments, and value of blanks are sent to ACMS for Forms Session
ID. TP Desktop Connector does require that you issue the acmsdi_poll service
to read the messages from the back end, and that you issue the acmsdi_
complete_pp service to signal completion of exchange step processing.

Note

Read-only and write-only in reference to arguments are always from
the callee's point of view. For these services, read-only arguments are
read from the client application's memory and write-only arguments
written to the client application's memory. The roles of caller and callee
are reversed from the exchange step callbacks, and therefore, their
read-only and write-only attributes are reversed, with one exception.
The submitter identifier is always the first argument and is always
read-only for these services.

7.4.1 Enable Exchange Arguments

When acmsdi_poll returns ACMSDI_ENABLE_EXCH, the client application can issue an acmsdi_bind_enable_args service call to retrieve the write-only arguments. You can issue an acmsdi_bind_session_id to send the forms session ID argument to ACMS. For both calls, the first argument, the submitter ID, is a read-only argument and must represent the same submitter for which acmsdi_poll returned ACMSDI_ENABLE_EXCH.

The argument for acmsdi_bind_enable_args are the same as the arguments for the acmsdi_enable presentation procedure with three exceptions:

* One additional argument, the form's version, is included on the acmsdi_bind_enable_args call. The version is included so that version checking can be performed by the client application as part of the form's enable processing.

Note

The acmsdi_check_version callback is not supported for forced nonblocking sessions.

---

* The call context is not included because it is returned by the acmsdi_poll service call.

* The forms session ID argument is not included because it is a read-only argument that is sent to ACMS using the acmsdi_bind_session_id service call.

You must declare memory for all arguments in the client application and pass them by reference.

The C-language prototype for acmsdi_bind_enable_args follows:

```
int acmsdi_bind_enable_args (ACMSDI_SUBMITTER_ID *sub_id,   /*read-required*/
                             char  *file_specification,  /*write-optional*/
                             char  *form_specification,  /*write-optional*/
                             char  *form_version,        /*write-optional*/
                             char  *forms_print_file,    /*write-optional*/
                             char  *forms_language,      /*write-optional*/
                             ACMSDI_CALL_ID  **call_id);
```

After the acmsdi_bind_enable_args call has successfully executed, the write-only arguments contain the value passed from the TP Desktop Connector client services.

The C-language prototype for acmsdi_bind_session_id follows:

```
int acmsdi_bind_session_id (ACMSDI_SUBMITTER_ID *sub_id,   /*read-required*/
                            ACMSDI_FORMS_SESSION_ID *session_id);
```

Example 7–1, written in Visual Basic, illustrates:

- Issuance of acmsdi_poll

- Receiving return code ACMSDI_ENABLE_EXCH

- Issuance of acmsdi_bind_enable_args

- Issuance of acmsdi_bind_session_id to send forms session identifier to ACMS

Example 7–1 Visual Basic Sample

```
Dim call_id As acmsdi_call_id
Dim subm_id As acmsdi_sub_id
Dim call_id_retr As Long
Dim fs_id As acmsdi_forms_session_id
Dim filespec As String * 256
Dim formspec As String * 256
Dim formversion As String * 256
Dim formprint As String * 256
Dim formlang As String * 256
Static status As Integer
status = acmsdi_poll(subm_id, ByVal 0&)
If status = ACMSDI_ENABLE_EXCH Then
     status = acmsdi_bind_enable_args(subm_id,
                                 filespec,
                                 formspec,
                                 formversion,
                                 formprint,
                                 formlang,
                                 call_id_retr)
```

Example 7–1 (Cont.) Visual Basic Sample

```
    If status = ACMSDI_NORMAL Then
                  >>> Process Enable arguments <<<
          fs_id.session_id = "FORMS_SESS000251"    'set forms session id
          status = acmsdi_bind_session_id(subm_id, fs_id) 'send it to ACMS
          If status <> ACMSDI_NORMAL Then
                  >>> Error processing <<<
          End If
          status = acmsdi_complete_pp(call_id, FORMS_NORMAL) 'end exchange
    Else
          >>> Error processing >>>
    End If
End If
```

7.4.2 TDMS Read Exchange Step Arguments

When acmsdi_poll returns ACMSDI_TDMS_READ_EXCH (a TDMS read
exchange step), the client application can issue an acmsdi_bind_msg to retrieve
the prompt text and a second acmsdi_bind_msg to send the message text from
the forms message field. You must declare memory for both arguments in the
client application and pass them by reference.

The C-language prototype for acmsdi_bind_msg follows:

```
int acmsdi_bind_msg (ACMSDI_SUBMITTER_ID *subm_id,   /*read-required*/
                     short direction,                /*read-required*/
                     short length,                   /*read-required*/
                     char *text,           /*read or write-required*/
                     ACMSDI_CALL_ID **call_id);      /*write-optional*/
```

The direction argument is a code indicating one of the following:

- 1 — indicates that the prompt text is being acquired from ACMS.

- 0 — indicates that the message text is being sent to ACMS.

After acquiring the prompt text by issuing an acmsdi_bind_msg with direction
equal to 1, the prompt may be displayed on the form. The client application
then waits for the user to enter message text in the form's message field.
After the user indicates that the message text is completely entered, the client
application issues a second acmsdi_bind_msg with direction equal to 0 to send
the message text to the server.

The first argument, the submitter ID, is a read-only argument and must
represent the same submitter for which acmsdi_poll returned ACMSDI_
TDMS_READ_EXCH.

Example 7–2 illustrates the following:

- Issuance of acmsdi_bind_msg

- Receiving of return code ACMSDI_TDMS_READ_EXCH

- Issuance of acmsdi_bind_msg to retrieve the prompt text

- Issuance of acmsdi_bind_msg to send the message text

Example 7–2 ACMSDI_TDMS_READ_EXCH Sample

```
Dim subm_id As acmsdi_sub_id
Dim call_id As acmsdi_call_id
Dim direction As Integer
Dim msg_len As Integer
Dim msg_text As String * 80
Dim prompt_len As Integer
Dim prompt_text As String * 40
Dim status As Integer
status = acmsdi_poll(subm_id, ByVal o&)
If status = ACMSDI_TDMS_READ_EXCH Then
      direction = 1
      prompt_len = 40
 status = acmsdi_bind_msg(subm_id,
                              direction,
                              prompt_len,
                              prompt_text,
                              ByVal 0&)
      If status = ACMSDI_NORMAL Then
       '
           ' display prompt on the form and wait for user
           ' to enter message text in the form's message field
  '
  direction = 0
  msg_len = 80
  status = acmsdi_bind_msg(subm_id,
                               direction,
                               msg_len,
                               msg_text,
                               ByVal 0&)
  acmsdi_complete_pp(call_id, TSS_NORMAL)
 Else
  >>> Error processing <<<
 End If
End If
```

7.4.3 TDMS Write Exchange Step Arguments

When acmsdi_poll returns ACMSDI_TDMS_WRITE_EXCH (a TDMS write exchange step) the client application can issue an acmsdi_bind_msg to retrieve the message text. You must declare memory for the message text argument in the client application and pass it by reference.

The C-language prototype for acmsdi_bind_msg follows:

```
int acmsdi_bind_msg (ACMSDI_SUBMITTER_ID *subm_id,  /*read-required*/
            short direction,              /*read-required*/
            short length,                 /*read-required*/
            char *text,          /*read or write-required*/
            ACMSDI_CALL_ID **call_id);    /*write-optional*/
```

The direction argument is a code indicating that the message text is being retrieved from ACMS. Its value is 1 to indicate retrieval as opposed to sending of the text. After acquiring the message text, the message may be displayed on the form.

The first argument, the submitter ID, is a read-only argument and must represent the same submitter for which acmsdi_poll returned ACMSDI_ TDMS_WRITE_EXCH.

Example 7–3 illustrates the following:

- Issuance of acmsdi_poll

- Receiving of return code ACMSDI_TDMS_WRITE_EXCH

- Issuance of acmsdi_bind_msg to retrieve the message text

7.4.4 Receiving Exchange Arguments

When acmsdi_poll returns ACMSDI_RECV_EXCH, the client application can issue and acmsdi_bind_receive_args service call to retrieve the write-only arguments. The first argument, the submitter ID, is the read-only argument and must represent the same submitter for which acmsdi_poll returned ACMSDI_RECV_EXCH. You can compare the second argument, forms session ID, to the forms session ID sent to ACMS by acmsdi_bind_session_id, to determine the forms session established earlier during an enable exchange.

The arguments for acmsdi_bind_receive_args are the same as the arguments for the acmsdi_receive presentation procedure except that the receive control text and the send control text are not included. These arguments are treated

Example 7–3 ACMSDI_TDMS_WRITE_EXCH Sample

```
Dim subm_id As acmsdi_sub_id
Dim call_id As acmsdi_call_id
Dim direction As Integer
Dim msg_len As Integer
Dim msg_text As String * 80
Dim status As Integer
status = acmsdi_poll(subm_id, ByVal 0&)
If status = ACMSDI_TDMS_WRITE_EXCH Then
      direction = 1
      msg_len = 80
 status = acmsdi_bind_msg(subm_id,
                                direction,
                                msg_len,
                                msg_text,
                                ByVal 0&)
      If status = ACMSDI_NORMAL Then
       '
            ' display the message on the form's message field
  '
  acmsdi_complete_pp(call_id, TSS_NORMAL)
 Else
  >>> Error processing <<<
 End If
End If
```

as forms records, and, therefore, are handled by acmsdi_bind_receive_recs and acmsdi_bind_send_recs.

You must declare memory for all arguments in the client application and pass them by reference.

The C-language prototype for acmsdi_bind_receive_args follows:

```
int acmsdi_bind_receive_args (ACMSDI_SUBMITTER_ID *sub_id, /*read-required*/
                        ACMSDI_FORMS_SESSION_ID *fs_id,  /*write-optional*/
                        char *receive_record_id,         /*write-optional*/
                        long *receive_record_count,      /*write-optional*/
                        short *timeout,                  /*write-optional*/
                        ACMSDI_CALL_ID **call_id);
```

After the acmsdi_bind_receive_args has successfully executed, the write-only arguments contain the values passed from TP Desktop Connector client services. The client application has the receive record identifier and knows which set of forms records it needs to send back to ACMS. You can send receive forms records, including receive control text, to TP Desktop Connector using the acmsdi_bind_receive_recs service. You can obtain the send control text using the acmsdi_bind_send_recs service.

Example 7–4, written in Visual Basic, illustrates:

- Issuance of acmsdi_poll

- Receiving return code ACMSDI_RECV_EXCH

- Issuance of acmsdi_bind_receive_args

Example 7–4 ACMSDI_RECV_EXCH Sample

```
Dim Call_id As acmsdi_call_id
Dim subm_id As acmsdi_sub_id
Dim forms_sess_id As acmsdi_forms_session_id
Dim recv_rec_id As String * 256
Dim recv_rec_count As Long
Dim timeout As Integer
Dim call_id_retr As Long
Dim status As Integer
status = acmsdi_poll(subm_id, ByVal 0&)
If status = ACMSDI_RECV_EXCH Then
     status = acmsdi_bind_receive_args(subm_id,
                             forms_sess_id,
                             recv_rec_id,
                             recv_rec_count,
                             timeout,
                             call_id_retr)
     If status = ACMSDI_NORMAL Then             'if all OK and ...
        If form_sess_id.session_id = "FORMS_SESS000251" Then  ' ...my session
             >>> Process Receive Exchange arguments for my form >>>
             acmsdi_complete_pp(call_id, FORMS_NORMAL)
        End If
     Else
             >>> Error processing <<<
     End If
End If
```

7.4.5 Requesting Exchange Step Arguments

When acmsdi_poll returns ACMSDI_REQUEST_EXCH (a TDMS exchange step) the client application can issue an acmsdi_bind_request_args to retrieve the write-only arguments. You must declare memory for all arguments in the client application and pass them by reference.

The C-language prototype for acmsdi_bind_request_args follows:

```
int acmsdi_bind_request_args (ACMSDI_SUBMITTER_ID *subm_id,  /*read-required*/
                              char *request_name,            /*write-optional*/
                              long *workspace_count,         /*write-optional*/
                              ACMSDI_CALL_ID **call_id);     /*write-optional*/
```

After the acmsdi_bind_request_args call has successfully executed, the write-only arguments contain the values passed from TP Desktop Connector client services. The client application has the request name and knows which set of workspaces it will be receiving and sending back to ACMS. Use acmsdi_bind_request_wksps to receive workspaces from TP Desktop Connector. You must must modify these workspaces appropriately before issuing a second acmsdi_bind_request_wksps call to send them back to ACMS.

The first argument, the submitter ID, is a read-only argument and must represent the same submitter for which acmsdi_poll returned ACMSDI_REQUEST_EXCH.

Example 7–5 illustrates the following:

- Issuance of acmsdi_poll

- Receiving of return code ACMSDI_REQUEST_EXCH

- Issuance of acmsdi_bind_request_args

Example 7–5   ACMSDI_REQUEST_EXCH Sample

```
Dim subm_id As acmsdi_sub_id
Dim call_id As acmsdi_call_id
Dim request_name As String *64
Dim workspace_count As Long
Dim call_id_retr As Long
Dim call_id_ref As Long
Dim status As Integer
call_id_ref = acmsdi_return_pointer(call_id)
status = acmsdi_poll(subm_id, ByVal 0&)
If status = ACMSDI_REQUEST_EXCH Then
     status = acmsdi_bind_request_args(subm_id,
                                  request_name,
                                  workspace_count,
                                  call_id_retr)
```

Example 7–5 (Cont.) ACMSDI_REQUEST_EXCH Sample

```
    If status = ACMSDI_NORMAL Then            'if all OK and ...
        If call_id_retr = call_id_ref Then   ' ... it;s the call I made
                '
                ' check the request name so we'll know which workspaces we
                ' are dealing with
                '
                If request_name = "MY_REQUEST" Then
                        >>> Process Request Exchange arguments <<<
                        acmsdi_complete_pp(call_id, TSS_NORMAL)
                End If
        End If
    Else
        >>> Error processing <<<
    End If
End If
```

7.4.6 Send Exchange Step Arguments

When acmsdi_poll returns ACMSDI_SEND_EXCH, the client application can
issue an acmsdi_bind_send_args to retrieve the write-only arguments. You
must declare the memory for all arguments in the client application and pass
them by reference.

The first argument, the submitter ID, is a read-only argument and must
represent the same submitter for which acmsdi_poll returned ACMSDI_SEND_
EXCH. The C-language prototype for acmsdi_send_args follows:

```
int acms_bind_send_args (ACMSDI_SUBMITTER_ID *subm_id,    /*read-required*/
                    ACMSDI_FORMS_SESSION_ID *fs_id,       /*write-optional*/
                    char  *send_record_id,                /*write-optional*/
                    long  *send_record_count,             /*write-optional*/
                    short *timeout,                       /*write-optional*/
                    ACMSDI_CALL_ID **call_id);            /*write-optional*/
```

After the acmsdi_bind_send_args call has successfully executed, the write-
only arguments contain the values passed from TP Desktop Connector client
services. The client application has the send record identifier and knows which
set of forms records it will be receiving from ACMS. You can receive send
forms records from TP Desktop Connector using the acmsdi_bind_send_recs
service. You can send receive control text to TP Desktop Connector using the
acmsdi_bind_receive_args.

Example 7–6, written in Visual Basic, illustrates:

- Issuance of acmsdi_poll

- Receiving of return code ACMSDI_SEND_EXCH

- Issuance of acmsdi_bind_send_args

Example 7–6 ACMSDI_SEND_EXCH Sample

```
Dim call_id As acmsdi_call_id
Dim subm_id As acmsdi_sub_id
Dim forms_sess_id As acmsdi_forms_session_id
Dim send_rec_id As String * 256
Dim Send_rec_count As Long
Dim timeout As Integer
Dim call_id_retr As Long
Static status As Integer
status = acmsdi_poll(subm_id, ByVal 0&)
If status = ACMSDI_SEND_EXCH Then
     status = acmsdi_bind_send_args(subm_id,
                                    forms_sess_id,
                                    send_rec_id,
                                    send_rec_count,
                                    timeout,
                                    call_id_retr)
     If status = ACMSDI_NORMAL Then           ' if all OK and ...
            If forms_sess_id = "FORMS_SESS000251" Then  ' ... my session
                  >>> Process Send Exchange arguments <<<
                  acmsdi_complete_pp(call_id, FORMS_NORMAL)
            End If
     Else
             >>> Error Processing <<<
     End If
End If
```

7.4.7 Transceive Exchange Step Arguments

When acmsdi_poll returns ACMSDI_TRCV_EXCH, the client application can
issue an acmsdi_bind_transceive_args service call to retrieve the write-only
arguments. You must pass memory by reference for all arguments in the client
application. The first argument, submitter ID, is a read-only argument and
must represent the same submitter for which acmsdi_poll returned ACMSDI_
TRCV_EXCH.

The C-language prototype for acmsdi_bind_transceive_args follows:

```
int acmsdi_bind_transceive_args (ACMSDI_SUBMITTER_ID *subm_id, /*read-required*/
                      ACMSDI_FORMS_SESSION_ID *fs_id,  /*write-optional*/
                      char  *send_record_id,           /*write-optional*/
                      long  *send_record_count,        /*write-optional*/
                      char  *receive_record_id,        /*write-optional*/
                      long  *receive_record_count,     /*write-optional*/
                      short *timeout,                  /*write-optional*/
                      ACMSDI_CALL_ID  **call_id);       /*write-optional*/
```

After the acmsdi_bind_transceive_args call has successfully executed, the
write-only arguments contain the values passed from TP Desktop Connector
client services. The client application has the send record identifier and knows
which set of forms records it needs to send back to ACMS. Use the acmsdi_
bind_send_recs service to receive forms records, including send control text,
from TP Desktop Connector. Use acmsdi_bind_receive_recs to send receive
forms records, including receive control text, to TP Desktop Connector. You can
issue both of these calls before issuing and acmsdi_complete_pp service call.

Example 7–7, written in Visual Basic, illustrates:

- Issuance of acmsdi_poll

- Receiving of return code ACMSDI_TRCV_EXCH

- Issuance of acmsdi_bind_transceive_args

Example 7–7 ACMSDI_TRCV_EXCH Sample

```
Dim call_id As acmsdi_call_id
Dim subm_id As acmsdi_sub_id
Dim forms_sess_id As acmsdi_forms_session_id
Dim send_rec_id As String * 256
Dim send_rec_count As long
Dim recv_rec_id As String * 256
Dim recv_rec_count As Long
Dim timeout As Integer
Dim call_id_retr As Long
Dim status As Integer
```

Example 7–7 (Cont.)   ACMSDI_TRCV_EXCH Sample

```
status = acmsdi_poll(subm_id, ByVal 0&)
If status = ACMSDI_TRCV_EXCH Then
     status = acmsdi_bind_transceive_args(subm_id,
                                          forms_sess_id,
                                          send_rec_id,
                                          send_rec_count,
                                          recv_rec_id,
                                          recv_rec_count,
                                          timeout,
                                          call_id_retr)
     If status = ACMSDI_NORMAL Then              ' if all OK and ...
        If forms_sess_id.session_id = "FORMS_SESS000251" Then  ' ...my session
               >>> Process Transceive Exchange arguments <<<
        End If
     Else
        >>> Error processing <<<
     End If
End If
```

7.5 Sending and Receiving Forms Records and Workspaces

TP Desktop Connector provides three forced nonblocking services to send and receive forms records (for HP DECforms type exchanges) and workspaces (for TDMS type exchanges) to and from TP Desktop Connector. These services are:

- acmsdi_bind_send_recs

  Receives forms records and control text from TP Desktop Connector during a HP DECforms type of exchange.

- acmsdi_bind_receive_recs

  Sends forms records and control text to TP Desktop Connector during a HP DECforms type of exchange.

- acmsdi_bind_request_wksps

  Sends and receives workspaces during a TDMS type of exchange.

You invoke these services after you retrieve exchange step arguments with one of the services discussed in Section 7.4, because the client application knows what forms record or workspace array is required to be sent and received.

In addition to forms records, you can receive send control text and receive control text from and sent to TP Desktop Connector using these services. TP Desktop Connector does not require that you issue these service calls. However, if you do not issue them, the client application is not able to examine

forms records or workspaces sent from TP Desktop Connector and forms
records and workspaces sent to TP Desktop Connector will contain default
values.

TP Desktop Connector does require that you issue acmsdi_poll to read the
messages from the back end, and that you issue acmsdi_complete_pp to signal
completion of exchange step processing and to write reply messages to the back
end.

Note

In the following sections, send forms records and send control text
are received from TP Desktop Connector. Conversely, receive forms
records and receive control text are sent to TP Desktop Connector. The
adjectives send and receive when applied to forms records and control
text are from the back-end perspective.

7.5.1 Receiving Send Forms Records and Control Text

Send forms records are forms records that are sent from ACMS to the client
application during a send or transceive exchange. Use acmsdi_bind_send_recs
to cause send forms record data to be moved to the client application's buffer
from TP Desktop Connector. You can also use the acmsdi_bind_send_recs
service to cause send control text to be copied to the application's buffers from
TP Desktop Connector.

The send record identifier, which you can retrieve with the acmsdi_bind_send_
args and acmsdi_bind_transceive_args, implicitly defines the number and types
of the forms records. The acmsdi_bind_send_recs service passes an array of
ACMSDI_FORM_RECORD_BIND structures as one of its arguments. Each
ACMSDI_FORM_RECORD_BIND structure contains two pointers; one to the
data record and one to the shadow record. You must declare buffers for these
forms records in the client application. If a shadow record is not in use, its
pointer can be NULL.

ACMSDI_FORM_RECORD_BIND structures also contain the lengths of the
buffers in the client application and a field, initially set to zero, in which ACMS
Desktop returns the actual length of the forms records. If the forms record
length is greater than the buffer length, forms records are truncated in the
buffer. If the forms record length is less than the buffer length, the buffer is
not completely filled by forms record data.

You can use the acmsdi_bind_send_recs service to request that send control text be copied to the application from TP Desktop Connector. If the second argument has a value of 1, send control text is to be copied. A value of 0 specifies that send control text is not to be copied. If you specify the send control text, you must specify the corresponding ACMSDI_FORM_RECORD_ BIND structure as the first one in the array. When the call terminates, the rec_len field of the ACMSDI_FORM_RECORD_BIND structure contains the send control text count as opposed to the send control text length.

The first argument, the submitter ID, is a read-only argument and must represent the same submitter for which the acmsdi_poll returned ACMSDI_ SEND_EXCH or ACMSDI_TRCV_EXCH.

The C-language prototype for acmsdi_bind_send_recs follows:

```
int acmsdi_bind_send_recs (ACMSDI_SUBMITTER_ID *subm_id,  /*read-required*/
                int bind_send_ctrl_text,                  /*read-required*/
                ACMSDI_FORM_RECORD_BIND *send_rec_array); /*write-required*/
```

You must declare and initialize the array of ACMSDI_FORM_RECORD_BIND structures before issuing the acmsdi_bind_send_recs call. You can obtain pointers to the forms record buffers by using the acmsdi_return_pointer service. Initialize forms records pointers immediately prior to each issuance of the acmsdi_bind_send_recs service, to assure that they are pointing to the current locations of the forms record buffers.

Example 7–8 illustrates:

- Declaration of two send forms record buffers (one of which has a shadow record)

- Declaration of a send control text buffer

- Declaration and initialization of the ACMSDI_FORM_RECORD_BIND

- Call to acmsdi_bind_send_recs

Example 7–8 Sending Forms Records

```
'
' employee forms record type definition
'
Type employee_rec
     badge_nbr As Long
     name As String * 35
     ss_nbr As String * 11
End Type
'
' employee shadow type definition (one byte for each employee record field)
'
Type employee_shadow
     badge_nbr As String * 1
     name As String * 1
     ss_nbr As String * 1
End Type
'
' control forms record type definition
'
Type ctrl_rec
     ctrl_count As Integer
     total_emps As Integer
End Type
'
' control text type definition
'
Type ctrl_text
     ctrl_string(5) As String * 5
End Type
'
' forms record and send control text declarations
'
Dim empl As employee_rec
Dim empl_shdw As employee_shadow
Dim ctrl As ctrl_rec
Dim send_ctrl_text As ctrl_text
```

Example 7–8 (Cont.) Sending Forms Records

```
'
' ACMSDI_FORM_RECORD_BIND array for send forms records
'
Dim send_recs(3) As ACMSDI_FORM_RECORD_BIND
'
' other declarations
'
Dim subm_id As acmsdi_sub_id
Static status As Long
Const NULL = 0
Dim ctrl_text_flag As Integer
Dim send_ctrl_text_count As Integer
'
' initialize the send forms records array using acmdi_return_pointer to get
' forms record buffer pointers (looking like 32-bit integers)
'
send_recs(0).buffer_len = Len(send_ctrl_text)
send_recs(0).record_len = 0
send_recs(0).data_record = acmsdi_return_pointer(send_ctrl_text)
send_recs(0).shadow_buffer_len = 0
send_recs(0).shadow_rec_len = 0
send_recs(0).shadow_record = NULL
send_recs(1).buffer_len = Len(empl)
send_recs(1).rec_len = 0
send_recs(1).data_record = acmsdi_return_pointer(empl)
send_recs(1).shadow_buffer_len = Len(empl_shdw)
send_recs(1).shadow_rec_len = 0
send_recs(1).shadow_record = acmsdi_return_pointer(empl_shdw)
send_recs(2).buffer_len = Len(ctrl)
send_recs(2).rec_len = 0
send_recs(2).data_record = acmsdi_return_pointer(ctrl)
send_recs(2).shadow_buffer_len = 0
send_recs(2).shadow_rec_len = 0
send_recs(2).shadow_record = NULL
'
' set control text flag to indicate that we want to send control text retrieved
'
ctrl_text_flag = 1
'
' call to get send forms records
'
```

Example 7–8 (Cont.) Sending Forms Records

```
status = acmsdi_bind_send_recs(subm_id, ctrl_text_flag, send_recs(0))
send_ctrl_text_count = send_recs(0).rec_len
If send_recs(1).buffer_len < send_recs(1).rec_len Then
        >>> employee record truncated >>>
ElseIf send_recs(1).buffer_len > send_recs(1).rec_len Then
        >>> employee record buffer not completely filled <<<
Else
        >>> employee record exactly fits buffer <<<
End If
```

7.5.2 Sending Receive Forms Records and Control Text

Receive forms records are forms records that are sent from the client application to ACMS during a receive or transceive exchange. Use acmsdi_bind_receive_recs to send the client application's receive forms record buffer contents to TP Desktop Connector. You can also use acmsdi_bind_receive_recs service to cause receive control text to be copied to TP Desktop Connector from the application's buffers.

The receive record identifier argument, retrieved by either acmsdi_receive_args or acmsdi_bind_transceive_args, implicitly defines the number and types of the forms records. The acmsdi_bind_receive_recs service passes an array of ACMSDI_FORM_RECORD_BIND structures as one of its arguments. Each ACMSDI_FORM_RECORD_BIND structure contains two pointers; one to the data record and one to the shadow record. You must declare buffers for these forms records in the client application. If a shadow record is not in use, its pointer can be NULL.

ACMSDI_FORM_RECORD_BIND structures also contain the lengths of the buffers in the client application and a field, initially set to zero, in which TP Desktop Connector returns the actual length of the forms records. If the forms record length is greater than the buffer length, the buffer is not large enough to provide data for the entire forms record. If the forms record length is less than the buffer length, not all of the data in the buffer is transmitted to the back end.

You can use the acmsdi_bind_receive_args service to request that receive control text be copied to TP Desktop Connector. If the second argument has a value of 1, receive control text is copied. A value of 0 specifies that receive control text is not copied. If you specify the receive control text, you must specify its corresponding ACMSDI_FORM_RECORD_BIND structure as the first one in the array. Initialize the rec_len field to contain the receive control text count.

The first argument, the submitter ID, is a read-only argument and must represent the same submitter for which acmsdi_poll returned ACMSDI_RECV_EXCH or ACMSDI_TRCV_EXCH.

The C-language prototype for acmsdi_bind_receive_recs follows:

```
int acmsdi_bind_receive_recs (ACMSDI_SUBMITTER_ID *subm_id, /*read-required */
        int bind_receive_ctrl_text,                         /*read-required*/
        ACMSDI_FORM_RECORD_BIND *recv_rec_array);           /*read-required*/
```

You must declare and initialize the array of ACMSDI_FORM_RECORD_BIND structures before issuing the acmsdi_bind_receive_recs call. You can obtain pointers to the forms record buffers using the acmsdi_return_pointer service. Initialize forms record pointers immediately before each issuance of acmsdi_bind_receive_recs service to assure that they are pointing to the current locations of the forms record buffers.

Example 7–9 illustrates:

- Declaration of two receive forms record buffers (one of which has a shadow record)

- Declaration of a receive control text buffer

- Declaration and initialization of the ACMSDI_FORM_RECORD_BIND array

- Call to acmsdi_bind_receive_recs

Example 7–9 Receiving Forms

```
'
' employee forms record type definition
'
Type employee_rec
     badge_nbr As Long
     name As String * 35
     ss_nbr As String * 11
End Type
'
' employee shadow type definition (one byte for each employee record field)
'
Type employee_shadow
     badge_nbr As String * 1
     name As String * 1
     ss_nbr As String * 1
End Type
```

Example 7–9 (Cont.) Receiving Forms

```
'
' control forms record type definition
'
Type ctrl_rec
    ctrl_count As Integer
    total_emps As Integer
End Type
'
' control text type definition
'
Type ctrl_text
    ctrl_string(5) As String * 5
End Type
'
' forms record declarations
'
Dim empl As employee_rec
Dim empl_shdw As employee_shadow
Dim ctrl As ctrl_rec
Dim send_ctrl_text As ctrl_text
'
' ACMSDI_FORM_RECORD_BIND array for receive forms records
'
Dim recv_recs(3) As ACMSDI_FORM_RECORD_BIND
'
' other declarations
'
Dim subm_id As acmsdi_sub_id
Static status As Long
Const NULL = 0
Dim ctrl_text_flag As Integer
Dim send_ctrl_text_count As Integer
```

Example 7–9 (Cont.) Receiving Forms

```
'
' initialize receive control text
'
receive_ctrl_text_count = 2
receive_ctrl_text.ctrl_string(0) = "AB001"
receive_ctrl_text.ctrl_string(1) = "CC599"
'
' initialize the receive forms records array using acmdi_return_pointer to get
' forms record buffer pointers (looks like 32-bit integers)
'
recv_recs(0).buffer_len = Len(receive_ctrl_text)
recv_recs(0).record_len = receive_ctrl_text_count
recv_recs(0).data_record = acmsdi_return_pointer(receive_ctrl_text)
recv_recs(0).shadow_buffer_len = 0
recv_recs(0).shadow_rec_len = 0
recv_recs(0).shadow_record = NULL
recv_recs(1).buffer_len = Len(empl)
recv_recs(1).rec_len = 0
recv_recs(1).data_record = acmsdi_return_pointer(empl)
recv_recs(1).shadow_buffer_len = Len(empl_shdw)
recv_recs(1).shadow_rec_len = 0
recv_recs(1).shadow_record = acmsdi_return_pointer(empl_shdw)
recv_recs(2).buffer_len = Len(ctrl)
recv_recs(2).rec_len = 0
recv_recs(2).data_record = acmsdi_return_pointer(ctrl)
recv_recs(2).shadow_buffer_len = 0
recv_recs(2).shadow_rec_len = 0
recv_recs(2).shadow_record = NULL
'
' set control text flag to indicate that we want to receive control text sent
'
ctrl_text_flag = 1
'
' call to send receive forms records
'
status = acmsdi_bind_receive_recs(subm_id, ctrl_text_flag, recv_recs(0))
If recv_recs(1).buffer_len < recv_recs(1).rec_len Then
        >>> client application buffer is too small  >>>
ElseIf recv_recs(1).buffer_len > recv_recs(1).rec_len Then
        >>> client application buffer is too large  <<<
Else
        >>> client application buffer is exactly the right size <<<
End If
```

7.5.3 Sending and Receiving TDMS Request Workspaces

Request workspaces are workspaces that are sent from ACMS to the client application and sent back to ACMS during a TDMS Request exchange. Use acmsdi_bind_request_wksps to cause request workspace data to be copied to the client application's buffers from TP Desktop Connector. After the workspaces have been modified, use acmsdi_bind_request_wksps service a second time to send back the modified contents to TP Desktop Connector.

The request name argument, retrieved by the acmsdi_bind_request_args service, implicitly defines the number and types of the workspaces. The acmsdi_bind_request_wksps service passes an array of ACMSDI_WORKSPACE_BIND structures as one of its arguments. Each ACMSDI_WORKSPACE_BIND structure contains a pointer to the workspace buffer. The buffers must be declared in the client application.

ACMSDI_WORKSPACE_BIND structures also contain the lengths of the buffers in the client application and a field, initially set to zero, in which TP Desktop Connector returns the actual length of the workspaces. If the workspace length is less than the buffer length, the buffer is not filled completely by workspace data.

The second argument is an integer indicating the direction in which the workspaces are sent. A value of 1 indicates that the workspaces are copied into the application's buffers from TP Desktop Connector. A value of 0 indicates that workspaces are copied to TP Desktop Connector from the application's buffers.

The first argument, the submitter ID, is a read-only argument and must represent the same submitter for which acmsdi_poll returned ACMSDI_REQUEST_EXCH.

The C-language prototype for acmsdi_bind_request_wksps follows:

```
int acmsdi_bind_request_wksps (ACMSDI_SUBMITTER_ID *subm_id, /*read-required*/
      int direction,                              /*read=required*/
      ACMSDI_WORKSPACE_BIND  *req_wksp_array);    /*read-write-required*/
```

You must declare and initialize the array of ACMSDI_WORKSPACE_BIND structures issuing the acmsdi_bind_request_wksps call. Obtain pointers to the workspace buffers using the acmsdi_return_pointer service. You must initialize the workspace pointers before each issuance of the acmsdi_bind_request_wksps service to assure that they are pointing to the current locations of the workspace buffers.

Example 7–10 illustrates:

- Declaration of two TDMS request workspace buffers

- Declaration and initialization of the ACMSDI_WORKSPACE array

- Call to acmsdi_bind_request_wksps

Example 7–10 TDMS Sample

```
'
' employee workspace type definition
'
Type employee_wksp
     badge_nbr As Long
     name As String *35
     ss_nbr As String * 11
End Type
'
' control workspace definition
'
Type ctrl_wksp
     ctrl_count As Integer
     total_emps As Integer
End Type
'
' workspace definitions
'
Dim empl As employee_wksp
Dim ctrl As ctrl_wksp
'
' ACMSDI_WORKSPACE_BIND array for TDMS request workspaces
'
' other declarations
'
Dim request_wksps(2) As ACMSDI_WORKSPACE_BIND
Dim status As Long
Const TO_ACMS = 0
Const FROM_ACMS = 1
```

Example 7–10 (Cont.)   TDMS Sample

```
'
' initialize the TDMS request workspaces array using acmsdi_return_pointer to
' get workspace buffer pointers (looks like 32-bit integers)
'
request_wksps(0).buffer_len = Len(empl)
request_wksps(0).wksp_len = 0
request_wksps(0).data = acmsdi_return_pointer(empl)
request_wksps(1).buffer_len = Len(ctrl)
request_wksps(1).wksp_len = 0
request_wksps(1).data = acmsdi_return_pointer(ctrl)
'
' call to get TDMS request workspaces
'
status = acmsdi_bind_request_wksps(subm_id, FROM_ACMS, request_wksps(0))
If request_wksps(0).buffer_len < request_wksps(0).wksp_len Then
      >>>  employee record truncated <<<
ElseIf request_wksps(0).buffer_len > request_wksps(0).wksp_len Then
      >>> employee record buffer not completely filled <<<

Else
      >>> employee record exactly fits buffer <<<
      >>> modify workspaces as required <<<
    '
    ' Having modified the workspaces, now send them back to ACMS
    request_wksps(0).data = acmsdi_return_pointer(empl) 'update empl pointer
    request_wksps(0).data = acmsdi_return_pointer(ctrl) 'update ctrl pointer
    status = acmsdi_bind_request_wksps(subm_id, TO_ACMS, request_wksps(0))
End If
```

7.6 Forced Nonblocking Flow of Control

The following steps illustrates a typical flow of control for a transceive exchange step during a forced nonblocking session:

1. The client application issues an acmsdi_sign_in call, specifying the option ACMSDI_OPT_NONBLK. This option indicates that the session is a forced nonblocking session.

2. The client application issues an acmsdi_call_task without a completion address. Because it is a forced nonblocking session, TP Desktop Connector ACMSDI expects the acmsdi_poll service calls to check for messages from the back-end application, instead of the acmsdi_dispatch_message service.

Note _____

Issuing an acmsdi_dispatch_message call during a forced nonblocking session causes an error condition, because TP Desktop has no completion address at which to dispatch upon task completion.
_____

3. ACMSDI creates a call task message and sends it to the back-end application.

4. ACMSDI returns ACMSDI_PENDING status to the client application, indicated that the task call has been successfully sent to the back-end application.

5. The client application issues an acmsdi_poll service call and receives the status, ACMSDI_EXEC, indicating that the task is still executing on the back-end application.

6. A message arrives from the back-end application.

7. The client application issues another acmsdi_poll service call to check for a message from the back-end application.

8. In response to the acmsdi_poll call, ACMSDI determines that a transceive exchange step request has been received from the back-end application and returns the status, ACMSDI_TRCV_EXCH, to the client application.

9. The client application issues an acmsdi_bind_transceive_args service, passing transceive request arguments by reference.

10. In response to the acmsdi_bind_transceive_args call, ACMSDI moves the arguments to the client application's memory locations.

11. From the send and receive record IDs gathered in step 10, the client application knows which forms records are used. The client application issues a series of acmsdi_return_pointer calls to obtain pointers to the send forms records to be placed in the ACMSDI_FORM_RECORD_BIND arrays.

12. The client application issues an acmsdi_bind_send_recs service, passing the ACMSDI_FORM_RECORD_BIND array for send forms records constructed in step 11.

13. ACMSDI copies send forms records to the client application's memory locations, including send control text. The client application can now display send forms record data and acquire data from receive forms records in its presentation procedure.

14. The client application issues a series of acmsdi_return_pointer calls to obtain pointers to the receive forms records to be placed in its ACMSDI_FORM_RECORD_BIND arrays.

15. The client application issues an acmsdi_bind_receive_recs call, passing receive forms records to ACMSDI with the ACMSDI_FORM_RECORD_BIND array constructed for receive forms records in step 14.

16. ACMSDI copies receive forms records from the client application's memory locations, including receive control text.

17. The client application issues an acmsdi_complete_pp service call to indicate that the presentation procedure has completed, passing the status code to be returned to the back-end application.

18. ACMSDI creates a transceive exchange step response message from the receive forms records copied from the client application in step 16. The transceive response message is sent to the back-end application. The client application can now resume polling for additional exchange steps or task completion messages.

7.6.1 Structures Declared in Client Application Memory

You allocate certain structures in the client application's memory and pass them by reference to TP Desktop Connector. These structures are:

- submitter identifier (ACMSDI_SUBMITTER_ID)

  Pass this structure by reference to TP Desktop Connector as an argument on acmsdi_sign_in service calls. TP Desktop Connector fills in its fields. It contains a pointer to a submitter structure, which TP Desktop Connector uses as a session control block. It is subsequently passed by reference as an argument to acmsdi_call_task and acmsdi_sign_out service calls, and used by TP Desktop Connector to identify the session.

- call identifier (ACMSDI_CALL_ID)

  Pass this structure by reference to TP Desktop Connector as an argument on acmsdi_call_task service calls. TP Desktop Connector fills in its fields. It contains a pointer to a call structure, which TP Desktop Connector uses as a call control block. It is subsequently passed by reference as an argument on acmsdi_cancel and acmsdi_complete_pp service calls, and used by TP Desktop Connector to identify the task call.

- call context (void *)

  Pass this object, created by the client application, to TP Desktop Connector as an argument on acmsdi_call_task, acmsdi_cancel, acmsdi_sign_in, and acmsdi_sign_out service calls. This object provides additional information used by client completion routines and presentation procedures to establish context for the call.

7.6.2 Differences Between Standard and Forced Nonblocking

In standard nonblocking mode, you pass structures by reference as arguments on calls from TP Desktop Connector to client application presentation procedures and completion routines. The pointers to these structures, originally passed to TP Desktop Connector, are passed back to the client routines. The client routines then use these pointers to locate the original structures, which point them to the submitters and calls referenced by the callbacks. Because applications which require forced nonblocking mode do not support callback routines or pointers, different rules apply.

TP Desktop Connector uses the submitter identifier on the service calls to identify the submitter for which the call is issued. For example, the acmsdi_poll service passes the submitter identifier as a read-only argument. In response, TP Desktop Connector returns the latest message type received from the back-end system for that submitter or a return code indicating that no message is available for the submitter. This differs from the acmsdi_dispatch_message service, which dispatches all messages that arrive from the back-end for all submitters.

The services acmsdi_bind_*xxxx*_args and acmsdi_poll retrieve the call identifier and call context arguments, pointers to structures existing in the client application's memory. Client applications treat these pointers as 32-bit integers; therefore, for these arguments to be useful to the client application, use the acmsdi_return_pointer service to obtain reference pointers (as 32-bit integers) for the structures.

Note

Because data can be moved by memory management routines, you must issue acmsdi_return_pointer calls in the same procedure where you issued the original call. Then compare the values of the arguments against the reference pointers to determine which structure is being referenced.

Example 7–11, written in BASIC, illustrates the creation of a call identifier on an acmsdi_call_task service call and its later identification during retrieval of the arguments of an enable exchange callback. The example assumes the following:

- Submitter identifier, established by the acmsdi_sign_in call, identifies the TP Desktop Connector signed-in session.

- Call identifier is used as the call context at task completion.

- Session is forced nonblocking.

Example 7–11 Creation of a Call Identifier

```
Dim subm_id As acmsdi_sub_id          ' submitter id structure
Dim call_id As acmsdi_call_id         ' call id structure

Dim call_id_ref As Long               ' reference to call id
Dim call_id_retr As Long              ' call id retrieved from enable
Dim call_ctxt As Long                 ' call context retrieved using acmsdi_poll
Dim forms_sess_id As acmsdi_forms_session_id
Dim filespec As String * 256
Dim formspec As String * 256
Dim formversion As String * 256
Dim formprint As String * 256
Dim formlang As String * 256
Static status as Integer
Static final_status As Long
call_id_ref = acmsdi_return_pointer(call_id)
task_name$ = "MY_TASK"
appl_name$ = "MY_APPL"
status = acmsdi_call_task(subm_id,  ' submitter id structure created at sign-in
                ByVal 0&,           ' null call options
                task_name$,         ' task name
                appl_name$,         ' application name
                0&,                 ' null selection string
                status_msg,         ' status message return location
                0&,                 ' no workspace count
                ByVal 0&,           ' no workspace pointer
                call_id,            ' call id structure
                ByVal 0&,           ' null final completion status location
                0&,                 ' null completion routine pointer
                call_id)            ' call context same as call id
```

(continued on next page)

Example 7–11 (Cont.) Creation of a Call Identifier

```
If status = ACMSDI_PENDING Then     ' If nonblocking call sent to back-end
    Do                              ' loop while the task is executing
          status = acmsdi_poll(subm_id, call_ctxt)   ' Note: Instead of a Do
          if status <> ACMSDI_EXEC                    ' Loop, the acmsdi_poll
                Exit Do                               ' and code following
          End If                                      ' would probably be
                                                      ' issued in a process
    Loop                                              ' kicked off by a timer
If status = ACMSDI_ENABLE_EXCH Then  ' If we have an enable exchange step
    '
    ' call to retrieve enable arguments
    '
    status = acmsdi_bind_enable_args(subm_id,     ' submitter id
                            filespec,     ' file specification
                            formspec,     ' forms specification
                            formversion,  ' form version
                            formprint,    ' print file
                            formlang,     ' form language
                            call_id_retr) ' call id
    '
    ' check call id against out reference id.  If they match,
    ' this enable callback is ours
    '
    If call_id_retr = call_id_ref Then
                >>> Enable processing <<<
            forms_sess_id,session_id = "SESSION 12345678"      ' session id
            status = acmsdi_bind_session_id(subm_id, forms_sess_id) 'send it
            acmsdi_complete_pp(call_id, FORMS_NORMAL) ' finish Enable
                                                      ' processing
    End If
ElseIf status = ACMSDI_DONE Then     ' If we have task completion
    '
    ' check call context against our call reference id.  If it matches, this
    ' is a completion for our task
    '
    If call_ctxt = call_id_ref Then
            >>> Task completion processing >>>
    End If
End If
    >>> Handle other statuses from acmsdi_poll <<<
```

7.7 Forced Nonblocking Sample Application

The forced nonblocking sample allows you to initiate API calls and to respond
to exchange steps as they arrive from the back end, thus demonstrating the
use of forced nonblocking API calls. As each exchange step arrives, the sample

asks whether or not you want to "bind" the exchange arguments by using one or more API calls. If you choose to bind the arguments, they are retrieved from TP Desktop Connector and displayed in a text box.

Note

TP Desktop Connector does not require that exchange step arguments be bound. You can skip the binding of the basic arguments and bind only exchange step forms records or workspaces, although this sample does not demonstrate this feature.

The forced nonblocking sample application is written in Visual Basic and contains the files listed in the following table:

Table 7–2 Forced Nonblocking Sample Files

File Name	Description
FNBSAMPL.MAK	Visual Basic project file
FORM1.FRM	Main form
FRMLOGIN.FRM	Form used to login to TP Desktop Connector
FRMRECVC.FRM	Form used to determine receive control text values
ACMSDI.BAS	TP Desktop Connector global declarations[1]
CONSTANT.TXT	Visual Basic global declarations[2]
FORMS.BAS	TP Desktop Connector HP DECforms-style message codes and text[1]
MTEXT.BAS	TP Desktop Connector message text initialization procedures
NONBLK.BAS	Global declarations specific to this sample application

[1]This file was installed on your system as part of the installation of TP Desktop Connector Windows APIs. See **HP TP Desktop Connector for ACMS** *Installation Guide*.

[2]This file was installed with Visual Basic.

7.7.1 Starting the Forced Nonblocking Sample

Before you execute the sample, follow these steps:

1. Install TP Desktop Connector on the host server computer and on your PC.

2. Install the forced nonblocking sample on your PC. **HP TP Desktop Connector for ACMS** *Installation Guide* describes how to verify the installation of the forced nonblocking sample.

3. Install the appropriate TP Desktop Connector Dynamic Link Library (DLL) for the transport to be used, in the directory which contains the forced nonblocking sample or set a directory path to locate the DLL. Name the DLL "acmsdi.dll".

   Also, you must install the DLL "di_cnv.dll" (used for localization) in that directory or on that directory path. If no localization is required, a stub DLL is provided.

4. Check that ACMS is started on the host computer.

5. Create and start the ACMS application and task called by the sample on the ACMS system. Execute the following command at the $ prompt on the host computer where TP Desktop Connector has been installed:

   ```
   $ @ACMSDI$EXAMPLES_ACMS:FNBTASK
   ```

6. Check that the TP Desktop Connector Gateway for ACMS software is started on the host computer.

7. To start the sample execution, bring up the sample project, fnbsampl.mak, under Visual Basic and depress F5 or select Start from the Run menu. The main form will be displayed.

7.7.2 The Main Form

The main form contains a set of buttons that allows you to activate various API calls. It also contains an options box labeled "Exchange Args", which has two option buttons by which you can choose whether or not you wish to "bind" exchange step arguments.

The main form contains the following text boxes:

- Last Call

  Displays the name of the most current API call that was issued except for the acmsdi_poll call, which is continuously issued while polling is active.

- Current Exchange

  Displays the type of exchange step currently active, if any.

- Messages

  Gives you instructions and hints as to the next steps you can take.

- Arguments

  Displays exchange step arguments, forms records, and workspaces as they arrive from the backend.

The activation buttons are discussed in the following sections.

7.7.3 Starting and Stopping Polling

The polling button starts and stops polling. When polling is active the acmsdi_ poll API call is periodically issued in a timer-activated Sub procedure named Timer1_Timer. This button is labeled "Start Polling" if polling is not active or "Stop Polling" if polling is active. In addition, a text box below the button contains the word "Polling" when polling is active. You can start or stop polling at any time using this button. Polling is also automatically stopped and started at appropriate times in the application's procedures.

If polling is not active, task call, task cancel, sign-in and sign-out completions are not detected, nor the arrival of exchange steps. If the application appears to stall, it may be because polling is not active. If this is the case you can click this button to start polling.

Hint

When designing your own application, do not allow the user to start and stop polling but rather, automatically start and stop polling in the application's procedures. When polling is active, hide the polling activity from the user. For example, do not get into a polling loop which continuously displays the fact that there is no active task.

7.7.4 Forced Nonblocking Sample Sign In

To sign into TP Desktop Connector, click the button labeled "Sign In". The sample displays the Sign In form (FRMLOGIN.FRM) allowing you to enter the user name, password, and host node name. The "Cancel" and "OK" buttons allow you to complete the sign-in attempt or to cancel the sign-in attempt if you change your mind.

If you choose to complete the sign-in attempt, the application issues the acmsdi_sign_in API call. The Sign-In form is hidden and the application waits until acmsdi_poll recognizes the sign-in completion. At that time, an acmsdi_ complete_call API call is issued to retrieve the sign-in call status. The code for this call can be found in the SigninComplete procedure.

A dialog box is displayed to inform you of the success or failure of the sign-in attempt. If the dialog box does not appear in a reasonable amount of time, it may be because polling is stopped. If this is the case, simply click the button labeled "Start Polling".

7.7.5 Calling the ACMS Task from Sample

After you have successfully signed in, you can call the ACMS task, "FNBTASK", by clicking the "Call Task" button. The text box below this button contains the words "Task Executing". The sample application issues an acmsdi_call_task API call in the cmdCallTask_Click procedure.

Dialog boxes are displayed announcing the arrival of exchange steps. Exchange step arrivals are detected by the acmsdi_poll service, so polling must be active when the "Call Task" button is clicked. Examine the code found in the Timer procedure of the Timer1 object (Timer1_Timer) to see how polling is handled for this sample application.

7.7.6 Forced Nonblocking Exchange Steps

The ACMS task, FNBTASK, issues three HP DECforms-style exchange steps; transceive, send and receive, in that order. Because these are HP DECforms-style exchanges, the first exchange step to actually arrive for a given signed-in session is an enable exchange. If this were an actual HP DECforms session, this exchange provides the information HP DECforms needs to initialize the first form. However, in the visual basic environment, enable exchanges are often ignored.

Exchange steps in this sample are acknowledged using the F1 and F2 keys. The KeyUp procedure of the Form object (Form_KeyUp) contains the code which responds to the F1 and F2 keys.

TP Desktop Connector does not require that you issue any of the various "binds" which are used to retrieve and send exchange step arguments. However, the acmsdi_complete_pp call is required to signal the end of exchange step processing. For this sample, the code which issues this call is found in the CompleteExchange procedure.

7.7.6.1 Forced Nonblocking Enable Exchange Step

After clicking the OK button, which announces the arrival of the enable exchange, the Message text box contains instructions as to the next step you can take. You must select "Bind" or "Continue" from the Exchange Args options box and press the F1 key. If you select "Bind", the enable exchange arguments are read from TP Desktop Connector with the acmsdi_bind_enable_args call, and they are displayed in the Arguments text box. If you select "Continue", acmsdi_bind_enable_args is not issued and the application proceeds directly to the transceive exchange step.

If you choose to bind the enable arguments, they are displayed in the Arguments text box. Instructions are displayed in the Messages text box. The next step is press the F2 key.

You then see a dialog box, which announces that a Session Identifier has been sent to TP Desktop Connector. This session identifier is useful if your application is written to handle HP DECforms sessions. The session identifier is sent with an acmsdi_bind_session_id call. The code that issues this call is found in the Form_KeyUp procedure.

7.7.6.2 Transceive, Send and, Receive Exchange Steps

The code for the transceive, send and receive exchanges is essentially the same; the differences being primarily in the number and types of arguments and forms records exchanged. The arrival of each exchange step is announced by the display of a dialog box. After the dialog is dismissed, instructions are displayed in the Messages text box. As was the case with the enable exchange, you are asked to choose whether or not you want to bind the exchange arguments by choosing "Bind" or "Continue" from the Exchange Args options box. Then you are instructed to press F1. Exchange arguments are displayed or not, depending on your choice. The API calls for retrieving the arguments are:

- acmsdi_bind_transceive_args

  The code can be found in the ShowTransceiveArgs procedure.

- acmsdi_bind_send_args

  The code can be found in the ShowSendArgs procedure.

- acmsdi_bind_receive_args

  The code can be found in the ShowReceiveArgs procedure.

After pressing F1, the exchange arguments are displayed in the Arguments text box if you chose to bind them. Whether you choose to bind the arguments or not, the next set of instructions requests that you press F2 to continue. If you chose to bind the arguments, the send records are displayed next, including the send control text. The TP Desktop Connector API call issued to retrieve the send control text and send forms records is acmsdi_bind_send_recs, the code can be found in the ShowSendRecs procedure.

In either case, pressing F2 eventually causes the Receive Control Text form (FRMRECVC.FRM) to be displayed. For this ACMS task, the receive control text is used to instruct the task to sleep for 5 seconds or to return immediately. If "Sleep" is chosen, the delay may be used to attempt to cancel the ACMS task using the Cancel Task button.

After the Receive Control Text form has been dismissed, the sample application completes the exchange step by calling acmsdi_bind_receive_recs to send receive control text and receive forms records back to ACMS and finally, by calling acmsdi_complete_pp. The code for these calls can be found in the CompleteExchange procedure.

7.7.7 Task Completion

After the last exchange step has completed, the acmsdi_poll service recognizes the task completion which arrives from the back end unless you decide to cancel the task (See Section 7.8.9). The CalltaskComplete procedure is given control. This procedure issues an acmsdi_complete_call API call to retrieve the final completion status and task argument workspaces from the back end. A dialog box is displayed showing the final completion status. After dismissing the dialog box, if the task completed normally, the task argument workspaces sent from the ACMS task are displayed in the Messages text box.

You can now either reexecute the task, perhaps trying some different options, by clicking the Call Task button or sign out by clicking the Sign Out button.

7.7.8 Signing Out

To sign out of TP Desktop Connector, click the button labeled "Sign Out". The application then issues the acmsdi_sign_out API call. The application waits until acmsdi_poll recognizes the sign-out completion. At that time, an acmsdi_complete_call API call is issued to retrieve the sign-out call status. The code for this call can be found in the SignoutComplete procedure.

A dialog box is displayed to inform you of the success or failure of the sign-out attempt. If the dialog box does not appear in a reasonable amount of time, it may be because polling is stopped. If this is the case, simply click the button labeled "Start Polling".

7.7.9 Cancelling the Task

After dismissing the Receive Control Text form (see Section 7.8.6.2), the current exchange step is completed and, if you chose to have the ACMS task "sleep" for 5 seconds, you can use that delay to cancel the task. Click the Cancel Task button. The application issues an acmsdi_cancel API call in the cmdCancelTask_Click procedure.

A dialog box is displayed to tell you that the task was or was not successfully cancelled. After dismissing that dialog box a second dialog box is displayed to show the final task completion. As in any API service completion the acmsdi_poll service recognizes both the cancel and the task completion.

Because the task cancellation is recognized as a task failure, the Messages
text box shows a set of task failure messages and the actual status message
received from ACMS:

```
Task was cancelled by task submitter
```

This represents a normal completion for a cancelled task. However, because
it represents a task failure, polling is automatically stopped. Therefore, to
proceed with another task call or a sign out call, start polling by clicking the
Start Polling button. This is not a recommended way to write your application.
As stated earlier, polling should be transparent to the end user. However, for
this sample application, this technique illustrates the importance of keeping
polling active in order to detect call completions and exchange steps.

A

Sample Application Code

This appendix describes where to find sample code that runs on an HP ACMS system with the TP Desktop Connector Gateway for ACMS and on supported desktop client systems. To locate the directories containing the code, use the logical names in Table A–1 and Table A–2.

The logical names in both tables are defined when the TP Desktop Connector gateway starts (see **HP TP Desktop Connector for ACMS** *Gateway Management Guide*). If they are not defined on your system, consult the system manager.

Table A–1 TP Desktop Connector API Directories

Logical Name	Directory Contents
ACMSDI$COMMON	Files common across platforms needed to compile and link the client services
ACMSDI$NT_I86	Windows TP Desktop client services libraries for building desktop client programs on Windows systems
ACMSDI$UNIX	Tru64 UNIX TP Desktop client services libraries for building Tru64 UNIX client programs
ACMSDI$VMS	OpenVMS TP Desktop client services libraries for building desktop client programs on OpenVMS systems

Table A–1 (Cont.)   TP Desktop Connector API Directories

Logical Name	Directory Contents
ACMSDI$VMS_ALPHA	OpenVMS Alpha TP Desktop client services libraries for building desktop client programs on OpenVMS Alpha systems
ACMSDI$VMS_I64	OpenVMS I64 client services libraries for building desktop client programs on OpenVMS I64 systems

---

Note

The logicals for OpenVMS clients point to `sys$common:[acmsdi.vms_clients]` directory. The logicals for Non-VMS clients point to `sys$common:[acmsdi.nonvms_clients]` directory.

Table A–2   TP Desktop Connector Directories

Logical Name	Directory Contents
ACMSDI_AVERTZ_DEFAULT	ACMS AVERTZ sources
ACMSDI$EXAMPLES	General tools
ACMSDI$EXAMPLES_ACMS	AVERTZ task definitions and database software
ACMSDI$EXAMPLES_MOTIF	Motif sample sources
ACMSDI$EXAMPLES_MOTIF_UNIX	Platform-specific files: makefile, executable, and so on, for the Tru64 UNIX Motif sample
ACMSDI$EXAMPLES_MOTIF_VMS	Platform-specific files: makefile, executable file, and so on, for the OpenVMS Motif sample
ACMSDI$EXAMPLES_MOTIF_VMS_ALPHA	Platform-specific files: makefile, executable file, and so on, for the OpenVMS Alpha Motif sample

Table A–2 (Cont.)   TP Desktop Connector Directories

Logical Name	Directory Contents
ACMSDI$EXAMPLES_MSWINDOWS_NT_I86	Program source and executable file for the Microsoft Windows nonblocking environment sample desktop client program
ACMSDI$EXAMPLES_MSWINDOWS_VB	Program source and executable file for the Microsoft Visual Basic sample desktop client programs
AVERTZ_TDB	ACMS AVERTZ sources

B
Tools

The ACMSDI$EXAMPLES directory contains a collection of general purpose tools, include files, and sample programs. Table B–1, Table B–2, and Table B–3 contain the following lists:

- Development tools and files

- Runtime tools

- General samples

Table B–1   Development Tools and Files

Tool	Description
FORMS.H	Include file that contains HP DECforms status codes. These status codes can be used by the client program's presentation procedures to return valid forms status values. (These values are used for the HP DECforms-style presentation procedures: acmsdi_enable, acmsdi_send, acmsdi_receive, acmsdi_transceive, acmsdi_disable)
MAKE_CBL.COM	Command procedure used by the tool MAKE_RECORDS.COM to generate COBOL include files for workspace definitions.
MAKE_FORMS_H.COM	Command procedure that generates an include file called forms.h. This header file contains all the status codes recognized by forms$manager.exe. It can be used by the client program's presentation procedures to return valid forms status values. (These values are used for the HP DECforms-style presentation procedures: acmsdi_enable, acmsdi_send, acmsdi_receive, acmsdi_transceive, acmsdi_disable.) For example:

```
$ @MAKE_FORMS_H
```

Table B–1 (Cont.)   Development Tools and Files

Tool	Description
MAKE_H.COM	Command procedure used by the tool MAKE_RECORDS.COM to generate C include files for workspace definitions.
MAKE_RECORDS.COM	Command procedure that generates C structure definitions or COBOL record definitions for the ACMS workspaces used by an application. This tool uses the record definitions in CDD to generate either C or COBOL include files. For example:
	$ **@MAKE_RECORDS** *cdd-path* **{ COBOL │ C }**
MAKE_TDMS_H.COM	Command procedure that generates an include file called tdms.h. This header file contains all the status codes recognized by tssshr.exe. It can be used by the client program's presentation procedures to return valid forms status values. (These values are used for the TDMS-style presentation procedure, acmsdi_request.) For example:
	$ **@MAKE_TDMS_H**
PPGEN.COM	Command procedure that generates presentation procedure code. This tool uses the ACMS application's .TDB file to generate code for the presentation procedures that does the following:
	• Validates the number and size of workspaces passed in to the presentation procedure.
	• Invokes the application-specific presentation procedures. (Names for the application-specific presentation procedures are based on the names used in the task definition.)
	Stubs for the application-specific presentation procedures are generated as well. For example:
	$ **@PPGEN** *tdb-file-name*
TDMS.H	Include file that contains TDMS status codes. These status codes can be used by the client program's presentation procedures to return valid forms status values. (These values are used for the TDMS-style presentation procedure, acmsdi_request.)

Table B–2   Runtime Tools

Tool	Description
ACMSDI$CANCEL.COM	Command procedure that automatically cancels TP Desktop Connector users who have been inactive for a specified period of time.
	See the ACMSDI$CANCEL.COM file for instructions on usage.
SHOW_DESKTOP_ USERS.EXE	Displays submitter information such as user name, client node name, network transport, and so forth, for all Desktop users that are currently signed in to the TP Desktop Connector gateway on this node. For example:
	$ **RUN SHOW_DESKTOP_USERS**

Table B–3   General Samples

Sample Build Procedure	Description
ACMSDI_GET_VERSION.COM	Command procedure that builds a sample version of ACMSDI$GET_VERSION.EXE. See also the ACMSDI_GET_VERSION sources.
BUILD_SHOW_DESKTOP_ USERS.COM	Command procedure that builds a sample, which demonstrates the use of the ACMSDI$GET_ SUBMITTER_INFO OpenVMS service. See also the SHOW_DESKTOP_USER sources.

Index

Task
   I/O (cont'd)
      comparison with NO I/O, 2–6
      converting to NO I/O, 2–8
      definition, 2–3
      procedure kinds, 4–1
   invocation
      expense, 2–6
      overhead, 2–7
   NO I/O
      definition, 2–3
      freedom, 2–6
      processor usage, 2–7
   processing phase, 1–8
   queued, 2–9
   reserve definition, 4–25
   sample definition, 3–3
   selection phase, 1–8
Task-call-task feature
   use in common applications, 2–7
Task definition
   I/O
      example, 2–5
   NO I/O
      example, 2–4
Task group
   form reference, 3–3
   sample definition, 3–3
Terminal
   mixing with desktop, 3–3
TP Desktop client service
   introduction to, 1–7
TRANSCEIVE
   applications using, 4–28

U

Unidirectional workspaces
   modifiable, 2–21
   read-only, 2–21
   write-only, 2–21
Usage
   processor, 2–7
User interface
   design, 2–15
   event-driven environment, 2–16

User interface (cont'd)
   software, 2–2

V

Version checking
   operations to implement, 3–5
   requesting, 4–20
   routine
      writing, 4–18
         Windows, 5–5
         X Windows, 6–5

W

Windows
   desktop client program
      building, 5–26
   development guidelines, 5–4
   sample, 5–5
Workspace
   coding, 4–8
   creating, 3–2
   design, 2–19
Write-only workspaces
   See Unidirectional workspaces

X

X Windows
   development guidelines, 6–4
   sample, 6–5