DCE for the HP e3000

HP e3000 MPE/iX Computer Systems

Edition 3

Manufacturing Part Number: B3821-90003
E0801

U.S.A. August, 2001

Notice

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

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

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

Restricted Rights Legend

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

Acknowledgments

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

Contents

Contents

Figures

Figures

Tables

Tables

Preface

This manual describes the DCE for the HP e3000, based on OSF DCE version 1.0.2 source code.

This manual is organized into the following chapters:

Chapter 1 , “General Information,” provides information on version identification and components and limitations.

Chapter 2 , “Configuring DCE Cells,” provides general information on using the DCE configurator tool and options.

Chapter 3 , “Threads Architecture on MPE/iX,” the section provides the architecture of threads on MPE/iX as well as building DCE programs.

Chapter 4 , “DCE 1.2.1 Features and Programming Notes,” describes the differences between DCE 1.0.2 and DCE 1.2.1.

Chapter 5 , “Programming with Kernel Threads,” provides basic thread creation and management routines

Chapter 6 , “Introduction to RPC,” provides the Remote Procedure Call component of the core services of OSF DCE.

Chapter 7 , “Programming with RPC 1.2.1 on MPE/iX,” provides examples of RPC application programming.

1 General Information

This version of DCE/3000 (version A.01.02) is based on OSF DCE version 1.0.2 source code. It provides the following OSF components for the core services:

- **Remote Procedure Calls (RPC)** — supports the development of distributed applications by making requests to remotely networked machines as if they were local. RPCs also implement network protocols used by clients and servers to communicate with each other.
- **Kernel Threads** — supports the interfaces defined in Draft 4 of the POSIX 1003.4a specification, with some exceptions as stated in this document.
- **Cell Directory Service (CDS)** — manages a database of information about the resources in a group of machines called a DCE cell. The database consists of the names of resources and associated attributes.
- **Distributed Time Service (DTS)** — provides synchronized time for the computers in a DCE cell.
- **DCE Security** — provides secure communications through the use of services such as authentication, which guarantees the identity of users, and authorization, which keeps track of user privileges.

In the DCE/3000 version A.01.12, the DCE application library is provided as both an archive library (libdce.a) and an executable library (DCEXL.HPDCE.SYS). The concept of an executable library is like the shared library on HP-UX. If you use the archive library, each application binary will contain its own copy of the DCE routines that it calls directly or indirectly. If you use shared library, all DCE applications can share the single copy of the DCEXL on a system.

Version Identification

Version information for the individual DCE/3000 components can be obtained by running the Version utility against the DCE program. You will find the product version (**B3821AA** A.01.02 for the domestic version, or **B3822AA** A.01.02 for the international version) and the program version control information at the beginning of the Version output. For example, the following is the output from the Version utility for an RPCD program:

```
:version idl.hpdce.sys
VERSION C.60.00 Copyright (C) Hewlett-Packard 1987. All Rights Resreved.

IDL.DCEPROGS.ROSEDCE

SOM #1
@(#) HP30315   A.05.10   95/02/08 NRT0 Startup routine
IDL1.2.1-002
DCEmpesrc-003
B0600001/SSICSOCN/$Revision: 1.2$

MAX STACK SIZE: 393216
MAX HEAP SIZE: 81920000
CAPABILITIES: BA,IA
UNSAT PROC NAME:
ENTRY NAME:
LIBRARY SEARCH LIST: CXL.LIB.ROSEDCE
```

DCE/3000 Components and Files

The DCE/3000 components, their corresponding files, the files size (in sectors), and a description of the files are listed in the following tables, Table 1-1 shows the CDS components.

Table 1-1 CDS Components

Filename	Description
/usr/bin/cdsd	shell script
cdsd.hpdce.sys	program
/usr/bin/cdscp	shell script
cdscp.hpdce.sys	program
cdsadv.pub.sys	command file
/usr/bin/cdsadv	shell script
cdsadv.hpdce.sys	program
cdsclerk.hpdce.sys	program

NOTE The file sizes list in these tables are for product **B3822AA**. The sizes may be different for product **B3821AA**.

The DTS components are shown in Table 1-2.

Table 1-2 DTS Components

Filename	Description
dtscp.pub.sys	command file
/usr/bin/dtscp	shell script
/usr/bin/dtsd	shell script
dtsd.hpdce.sys	program
dtsnullp.pub.sys	command file
/usr/bin/dts_null_provider	shell script
dtsnullp.hpdce.sys	program
dtsntpp.pub.sys	command file
/usr/bin/dts_ntp_provider	shell script
dtsntpp.hpdce.sys	program

The Security components are shown in Table 1-3.

Table 1-3 Security Components

Filename	Description
/usr/bin/sec_clientd	shell script
secclntd.hpdce.sys	program
seccrtdb.pub.sys	command file
/usr/bin/sec_create_db	shell script
seccrtdb.hpdce.sys	program
secadmin.pub.sys	command file
/usr/bin/sec_admin	shell script
secadmin.hpdce.sys	program
rgyedit.pub.sys	command file
/usr/bin/rgy_edit	shell script
rgyedit.hpdce.sys	program
acledit.pub.sys	command file
/usr/bin/acl_edit	shell script
acledit.hpdce.sys	program
dcelogin.pub.sys	command file
/usr/bin/dce_login	shell script
dcelogin.hpdce.sys	program
kinit.pub.sys	command file
/usr/bin/kinit	shell script
kinit.hpdce.sys	program
klist.pub.sys	command file
/usr/bin/klist	shell script
klist.hpdce.sys	program
destroy.pub.sys	command file
/usr/bin/kdestroy	shell script
kdestroy.hpdce.sys	program

The RPC components are shown in Table 1-4.

Table 1-4 RPC Components

Filename	Sector Size	Description
rpcd.pub.sys	32	command file
/usr/bin/rpcd	16	shell script
rpcd.hpdce.sys	547	program
rpccp.pub.sys	16	command file
/usr/bin/rpccp	16	shell script
rpccp.hpdce.sys	290	program
idl.pub.sys	16	command file
/usr/bin/idl	16	shell script
idl.hpdce.sys	2,166	program
uuidgen.pub.sys	16	command file
/usr/bin/uuidgen	16	shell script
uuidgen.hpdce.sys	125	program
dced.pub.sys		
/usr/bin/dced		
dced.hpdcde.sys		

The miscellaneous components are shown in Table 1-5.

Table 1-5 Miscellaneous Components

Filename	Description
/etc/dce_config/*	shell scripts for dce_config tool
/usr/lib/libdce.a	NMRL
DCEXL.HPDCE.SYS	DCE shared library
/usr/include/dce/*.h	header files
/usr/include/dce/*.idl	idl files
/opt/dce local/*	directories for DCE use
/usr/lib/libdce.sl	POSIX shared library.

Domestic and International Version

The DCE/3000 Security component of /usr/lib/libdce.a uses the Data Encryption Standard (DES) algorithm as its default encryption algorithm. Because the United States DOD restricts the export of DES software, DCE/3000 supports two binary versions:

The International version of the software disables the RPC data protection level *privacy*, disallowing users the ability to encrypt their data in RPCs. If an application specifies the *privacy* level of data protection while using the international version of /usr/lib/libdce.a, the application receives an rpc_s_unsupported_protect_level error. This restriction does not apply to the Domestic version.

2 Configuring DCE Cells

This section provides general information on using the DCE configurator to add your MPE/iX HP e3000 system into a cell. It is divided into two subsections:

- Using the DCE Configuration Tool — provides detailed steps to bring up the DCE Configuration main menu (these steps must be completed each time you change the DCE cell configuration).
- Using the DCE Configuration Options — provides detailed steps for each option in the DCE configuration main menu (basic familiarity with DCE terms and concepts are assumed) as described in the *Introduction to OSF DCE*.

Using the DCE Configuration Tool

The DCE configurator (called **dce_config**) is a shell-script-based configuration tool, this enables you to run **dce_config** from within the MPE/iX POSIX shell.

Check the following preliminary tasks before you enable the DCE configuration main menu:

- Ensure that the system network is running (RPC requires network sockets).
- Create an MPE/iX group named DCECONFG. At the system prompt, enter:

  ```
  NEWGROUP DCECONFIG
  ```

  You must be in an MPE/iX group (that is, your working directory must be an MPE/iX group not a POSIX directory) when you start the POSIX shell that runs **dce_config**.

Perform the following steps to obtain the DCE Main Menu for configuring cells:

1. Log on to the console as MANAGER.SYS,DCECONFG. At the system prompt, enter:

   ```
   HELLO MANAGER.SYS,DCECONFG
   ```

2. Enter the POSIX shell. At the system prompt, enter:

   ```
   sh.hpbin.sys -L
   ```

   The shell prompt is displayed (for example, shell/iX>).

3. Ensure that /usr/bin is in your shell command search path. At the shell prompt, enter:

   ```
   export PATH=/usr/bin:$PATH
   ```

4. Bring up the DCE cofiguration Main menu. At the shell prompt, enter:

   ```
   dce_config
   ```

   The DCE Main Menu as shown in Figure 2-1 is displayed on the console.

Figure 2-1 DCE Main Menu

```
DCE Main Menu

   1. CONFIGURE CLIENT      configure client and start DCE daemons
   2. START                 re-start DCE daemons
   3. STOP                  Stop DCE daemons
   4. REMOVE                 stop DCE daemons and remove data files created by DCE daemons

  99. EXIT

  selection:
```

From this menu you can configure your system as a DCE client or client system.

Configuring a DCE Client (Client-Only System)

A DCE client can not be configured without a functional DCE cell. In other words, when you configure your machine as a DCE client, the DCE cell that you are going to configure needs to be up and running. You need to know the name of the cell and the names of the systems that the DCE servers (Security, CDS and DTS) reside.

Before preceding with the DCE Client configuration, ensure that the HOSTS.NET.SYS file in your machine contains the IP addresses for the systems that are running as Servers. When complete, follow the description in the "Startup the DCE Configuration" menu to bring up the DCE main menu.

The following steps enable you to add your machine as a DCE client node:

1. Select "1. Configure Client" from the DCE Main Menu.
2. Respond to the questions as shown in Figure 2-2.

Figure 2-2 Security Client

```
What is the name of the Security Server for this cell you wish to
join? server1

   .
   .
   .
Enter the name of your cell (without /,,,/): n22cell

Enter Cell Administrator's principal name: cell_admin

Enter password:    password

   .
   .
   .
This machine is now a security client.
```

Two DCE daemon jobs (**rpcd**, **dced**) are streamed and are running. You are informed that your machine is now a Security client.

3. Respond to the questions shown in Figure 2-3 to add CDS client configuration to your system:

Figure 2-3 Add CDS Client

```
What is the name of a CDS server in this cell
(if there is more than one, enter the name of
the server to be cached if necessary)? <dcetst4>
```

4. Respond to the "...LAN profile..." question as shown in Figure 2-4.

Figure 2-4 Using LAN Profile Question

```
Create LAN profile so clients and servers can be divided into
profile groups for higher performance in a multi-lan cell? (n)<n>
```

One DCE daemon job (**cdsadv**) is now running and you are informed that this machine is now a CDS client.

5. To continue configuring the machine as a DTS clerk, DTS local server, or DTS global server, respond to the question shown in Figure 2-5.

Figure 2-5 Configuring Question

```
Should this machine be configured as a DTS Clerk, DTS Local Server, or
DTS Global Server? (Default is DTS Clerk)
(clerk, local, global, none)
```

Once DTS daemon job (**dtsd**) is running, you are informed that this machine is now a DTS clerk.

6. Select 99 to exit from dce_config.

WARNING **The password for the "cell_admin none none" user is a well-known default value. Since this is a security hole, it is recommended that the password be changed immediately after exiting this script by using "dce_login", then the "rgy_edit change" command.**

Removing the DCE Cell

To remove a cell, perform the following steps:

1. Bring up the DCE Main Menu (as described in "Using the DCE Configuration Tool" earlier in this section).
2. Select "4. REMOVE" from the DCE Main Menu. The **dce_config** tool displays the message as shown in Figure 2-6.

Figure 2-6 Remove Message

```
REMOVE will remove the node's ability to operate in the cell.
A reconfiguration of the node will be required. This node
should be unconfigured before a REMOVE is done. You may REMOVE
without unconfiguring if you are destroying the cell.
Do you wish to continue (y/n)? (n) <y>
```

A "Yes" response stops all running DCE daemons in that system, removes all remnants of previous DCE configuration and removes all remnants of previous DCE configuration for all components.

NOTE Existing user credentials will be invalid when DCE daemons are stopped and restarted.

3 Threads Architecture on MPE/iX

Threads library were the part of DCE product. From this release of RPC 1.2.1 threads will be delivered and maintained by the Process Management Group of CSY. Threads library is separated from the DCE library. This ensures that the threads programmers need not link their applications with the DCE library for the threads functionality.

The threads library on MPE/iX was supporting the POSIX 1003.4a Draft4. When the threads library was separated from the DCE product, the wrappers that are written over the OTHDXL.THREADS are ported to support the POSIX 1003.4c Draft10 APIs. The underlying library is still the Draft 4 version.

This section assumes that DCE application developers have some experienced in porting standard C applications to the MPE/iX POSIX environment. For application developers who are not familiar with the MPE/iX POSIX and C language interface, please read the *MPE/iX Developer's Kit* (36430A) first.

Threads Architecture

This section describes the architecture of threads on MPE/iX.

The following terminology is adopted throughout the remainder of this document. The term *process* refers to the MPE/iX operating system notion of process. The term *task* is defined as a multi-threaded application (depending on the implementation, a task can consist of a single process or multiple processes).

Threads on MPE/iX

A multi-threaded task on MPE/iX is implemented with multiple processes (one per thread). A task's threads are a cooperative processes in that they share some resources that are normally private to a process. All threads within a task share the same SR 5 space as the initial thread (a process created using run or createprocess). The heap and global variables are shared by all threads, along with loader information and system information regarding open files and sockets.

All other process resources are private to the thread. Each thread has its own NM stack, CM stack, pin number, PIB, PIBX, TCB, PCB, PCBX, process port, and so on. Fields within these data structures that are shared among threads (such as, file system information) are kept in a common location.

Process Management and Threads

An initial thread is a process created using **run** or **createprocess** (or **fork** and **exec** for POSIX). The threads of a task cannot exist independently of the initial thread. If the initial thread terminates or is killed, all of the task's threads are terminated. A secondary thread cannot be adopted by another task.

Each thread begins execution at an entry point specified at creation time. The entry point is an MPE/iX procedure with one parameter. This procedure resides in either the program file or the linked libraries of the task.

When a thread is created, the following attributes can be specified:

Stack size:	NM stack size for the thread
Inherit scheduling:	inherit the scheduling policies of the creating thread
Priority:	priority of the thread
Scheduling policy:	round robin, FIFO,...
Scheduling scope:	priority is global/local

These attributes are required in order to be POSIX compliant. POSIX also permits each implementation to add its own thread creation attributes. The following attribute was added for MPE/iX:

Debug: Enter debug before starting the thread

NOTE PH capability is required to create a thread.

From a process management point of view, thread creation is just an abbreviated form of process creation.

All threads are created as siblings. The threads of a task all have the same father task; namely, the father of the initial thread. If a thread creates a child using creatprocess, that child is the child of the task, not of the thread. From the tasks child-point-of-view, its father is the initial thread. When a thread exits, the children and the threads it created are not terminated.

Threads do not “own” the child processes they create. However, threads may find it necessary to wait for the termination of the offspring that they created. Therefore, a thread is permitted to wait for a specific child to terminate and is permitted to wait on the termination of any child. Refer to the **suspend** and **activate** intrinsics for more explanation.

While threads are implemented with multiple processes, to the end user threads should appear to coexist within a single process. Process management hides the MPE/iX implementation of threads from the programmer. The process handling intrinsics work on a task basis.

Development, Debugging, and Application Execution of Threads

This section discusses the development, debugging, and execution of applications that use threads on MPE/iX. It should be read before attempting to create or run an application that uses threads.

Debug has the following features to facilitate debugging in a threaded environment:

- Breakpoints
- Commands
- Environmental Variable

Breakpoints.

There are three types of breakpoints available when debugging a threaded program:

Breakpoint Type	Description
Task-Wide	Breakpoints that are recognized by any thread within a task.
Thread-Specific	Breakpoints that are identical to pin-specific breakpoints, but are thread-private, and are specified using an enhanced syntax.
Stop-All-Threads	Breakpoints with this option, when encountered by a thread within a threaded task suspend all other threads within the task until a `CONTINUE` command is issued.

The syntax for the address and pin parameters to breakpoint commands includes the specification:

```
logaddr [:pin|:@]
```

and the following for threads:

```
logaddr [:[[init_thread_pin].tin |.@][:@]]
```

where *tin* is the thread number returned by **pthread_create**. The pin number of the initial thread can be obtained using `SHOWPROC`. The syntax `[init_thread_pin].`*tin* specifies a thread, `[init_thread_pin].@` specifies a task-wide breakpoint, and `:@` following a `[init_thread_pin].`*tin* specification specifies a stop-all-threads breakpoint option.

For example:

Example Breakpoint	Description
`B thd_mtx:2e.2`	Sets a breakpoint at **thd_mtx** to be recognized by tin 2 of the task with initial thread 2e.
`B thd_mtx:.2`	Sets a breakpoint at **thd_mtx** to be recognized by tin 2 of the current task.
`B start_thread:2c.@`	Sets a task-wide breakpoint at **start_thread** to be recognized by all threads within the task with initial thread 2c.
`B start_thread:.@`	Sets a task-wide breakpoint at **start_thread** to be recognized by all threads within the current task.

`B HPFOPEN::@`	Sets a breakpoint at **HPFOPEN** for the current pin (tin) with the stop-all-threads option that is honored if the pin belongs to a threaded task.
`B HPFOPEN:.3:@`	Sets a breakpoint at **HPFOPEN** for tin 3 of the current task, and the breakpoint has the stop-all-threads option.
`B HPFOPEN:.@:@`	Sets a task-wide breakpoint at **HPFOPEN** for the current task, and the breakpoint has the stop-all-threads option.

Commands

The following commands aid in debugging threaded applications.

Command	Description
`TIN [init_thread_pin.]tin`	This command causes debug to switch to the environment of the specified tin. The default `init_thread_pin` is that of the current task. Privilege mode is required to switch to any tin in another task.
`SUSPEND`	This command suspends all other threads within the task of the tin being debugged. The suspended threads are not resumed automatically with the continue command.
`ACTIVATE`	This command resumes the threads that were suspended by the `SUSPEND` command. It should be issued from the same tin that issued the `SUSPEND` command.

Environmental Variables

There are two environment variables that simplify debugging applications:

Environment Variable	Description
`SS_TERM_KEEPLOCK`	When set to TRUE, a pin (tin) being debugged retains the terminal semaphore while single-stepping. This prevents any other pin (tin), that is waiting to enter debug, from obtaining the terminal semaphore and interfering with the debug session.

TERM_KEEPLOCK	Allows a process to retain the terminal semaphore under all conditions until the process terminates or the variable is reset to FALSE. However, this variable has the potential to create a deadlock. For example, a deadlock occurs if the process owning the terminal semaphore waits for another process that in turn is waiting for the debug terminal semaphore.

Limitations

The following are know limitations for the debug thread commands:

- The break command followed by an abort command hangs the task if the initial thread is waiting to enter debug (such as, another thread is currently in debug).
- The SUSPEND command has the potential to hang a task if the user does not issue an ACTIVATE command before doing the CONTINUE command.
- Each thread has its own debug environment. For example, loaded macros and environmental variables are not shared by threads within a task, and must be dealt with on an individual basis for each thread.

Building DCE Programs

Header Files

In addition to the standard POSIX libraries and HP C/XL functions, you may have to include the DCE header files, which can be found in the `/usr/include/dce` directory. If your C applications use **Try/Catch** for exception handling, you should include the following statement in the C programs:

```
#include <dce/pthread_exc.h>
```

There are no MPE/iX equivalent libraries for `/usr/lib/libbb.a` or `/usr/lib/libc_r.a`. The reentrant functions that are defined in MPE/iX and the thread-safe wrapper functions are in `/usr/lib/libdce.a`.

MPE/iX does not have the file `strings.h`. The HP-UX `strings.h` includes `string.h`, `sys/stdsyms.h` and some definitions that are strictly for C++ and HP-UX.

Compiler Flags

When compiling DCE applications using ANSI C under the MPE CI, set the following compiler switches:

```
-D_POSIX_SOURCE -D_MPEXL_SOURCE -D_SOCKET_SOURCE
-D_REENTRANT -Aa
```

When compiling under the MPE POSIX shell, you need the above flags except for the `-Aa` option. If `-Aa` is set, `/bin/c89` displays a large amount of error messages (by definition, the POSIX environment always uses the ANSI C compiler).

Unresolved Externals

When porting applications from a UNIX environment to MPE/iX, you may receive unresolved external errors during a compile, link, or run phase. It is likely that the unresolved externals are not part of the POSIX.1 standard. To find out if a function is defined in the POSIX environment, look at the manpage for that function on a UNIX system. At the bottom of the manpage, there is a section titled **STANDARD CONFORMANCE**, which lists the function name and the standard it conforms to. If the manpage does not have POSIX.1 listed as one of the standards then that function is not part of the MPE/iX POSIX Environment. To get around this porting issue, you may have to write a routine to emulate the functionality for the unresolved external.

4 DCE 1.2.1 Features and Programming Notes

This section describes the differences between DCE 1.0.2 and DCE 1.2.1.

- RPC changes
  - — Private Client Sockets
  - — Exception Handling
- IDL Compiler
  - — Out-of-Line Marshalling
  - — Enhanced IDL Data Types
  - — Support for IDL Encoding Services
  - — Support for User Defined Exceptions
  - — Support for Customized Binding Handles
- Control Programs and Daemons
- Transition of ACL Managers
- Removing DCE Credentials
- Serviceability Improvements
- Security Delegation
  - — Compiling Multithreaded Application

RPC Changes

Private Client Sockets

Previously a common pool of sockets was shared by concurrent RPC requests. Making this concurrency work requires a “helper” thread created to read from all of the open sockets, and passing received data onto the call thread for which it is intended. With “Private client sockets” there are a couple of sockets (2/3), which will only be used for individual requests (private to the request thread). This reduces the overhead of a “helper” thread, in case of small applications. When out of private sockets, socket sharing comes into effect.

Exception Handling

The new version of RPC 1.2.1 supports the exception-handling feature of RPC. The application developer can use the exception handling routines (**TRY**, **CATCH**, **CATCH-ALL** etc.).

IDL Compiler

Out-of-Line Marshalling

Out-of-line marshalling (library based marshalling) causes constructed data types such as unions, pipes or large structures to be marshalled or unmarshalled by auxiliary routines, thus reducing the stub size. The **out_of_line** attribute directs the IDL compiler to place the marshalling and unmarshalling code in IDL auxiliary stub files, rather than in the direct flow of the stub code.

Enhancing IDL Data Types

IDL support for arrays in the previous version was limited to:

- Arrays with a lower bound of zero.
- Arrays with conformance or varying dimensions only in the first (major) dimension.

Support for IDL Encoding Services

This extension to the IDL stub compiler will enable instances of one or more data types to be encoded into and decoded from a byte stream format suitable for persistent storage without invoking RPC Runtime.

Support for IDL Encoding Services

This extension to the IDL stub compiler will enable instances of one or more data types to be encoded into and decoded from a byte stream format suitable for persistent storage without invoking RPC Runtime.

Support for User Defined Exceptions

This extension to the IDL compiler will allow specification of a set of user-defined exceptions that may be generated by the server implementation of the interface. If an exception occurs during the execution of the server, it terminates the operation and the exception is propagated from server to client.

Support for Customized Binding Handles

This allows the application developer to add some information that the application wants to pass between the client and server. This can be used when application-specific data is appropriate to use for finding a server and the data is needed as a procedure parameter.

Control Programs and Daemons

The following control programs are delivered with DCE 1.2.1:

- **cdscp** — CDS control program
- **rpccp** — RPC control program
- **dtscp** — DTS control program
- **rgy_edit** — Registry Edit
- **acl_edit** — ACL edit

On OSF DCE 1.2.1 the above control programs are replaced by a single control program called DCECP. However, DCECP is not supported on DCE 1.2.1 on MPE/iX. The above programs are delivered and supported for MPE/iX.

The following daemons no longer exist:

- **sec_client**
- **rpcd**
- **cdsclerk**

DCED replaces **sec_clientd** and **rcpd**. The functionality of **cdsclerk** is part of **cdsadv.** Any scripts or programs that reference these non-existent daemons may need to be modified.

Transition of ACL Managers

OSD DCE 1.2.1 provides ACL management facilities within **libdce**. The **sec_acl_mgr** API is obsolete, and it is no longer necessary to write an ACL manager. Refer to the OSF DCE documentation to determine how to use the new **dce_acl** API to greatly reduce the amount of specialized ACL code that might have to be dealt with.

Application builders may want to try building their existing applications against DCE 1.2.1 before migrating their ACL management layer to the DCE supported **dce_acl** API. DCE 1.2.1 includes a backward-compatible set of header files that match the header files used by applications in previous DCE releases. Replace any instance of:

`#include <dce/daclmgr.h>` with `#include <dce/daclmgrv0.h>`

In makefiles and in application program, change all instances of:

`dalmgr` to `daclmgrv0`

These header files are provided as a transition aid only and should be used only until application is migrated to the **dce_acl** API.

Removing DCE Credentials

A user's DCE credentials (stored in the directory `/opt/dcelocal/var/security/creds`) are not automatically removed by exiting a shell or logging out. Unless any background processes require DCE credentials, the credentials can be removed before logging out by running **kdestroy** utility. This will make the system more secure by decreasing the opportunity for someone to maliciously gain access to your network credentials.

The `kdestroy` command has been modified to allow destruction of credentials older than a specified number of hours. **kdestroy** `-e` **exp-period** my be run manually to purge older credential files.

Serviceability Improvements

DCE 1.2.1 has an improved feature of Serviceability. This feature is helpful in debugging any problems under different sub components of DCE.

The default location for this file is `/opt/dcelocal/var/svc/routing`. The **DCE_SVC_ROUTING_FILE** environment variable can be used to name an alternate location for the file. The file is consulted if no switch is given on the command line or if no environment variable (**SVC_level** or **SVC_comp_DBG**) is found when a DCE process is started. Leading whitespace is ignored, as is any line whose first non-whitespace character is a #.

Production messages are parsed as:

`<level>:<where>:<parameter>`

`<level>` is FATAL ERROR WARNING NOTICE NOTICE_VERBOSE or * (meaning all)

`<where>` is STDERR STDOUT FILE (or TEXTFILE) BINFILE DISCARD

`<parameter>` is the filename, where “%ld” becomes the process-id

Send all messages to the console:

```
*. FILE:/dev/console
```

If FILE or BINFILE ends with “.n.m”, then at most “n” files and at most “m” messages for each file will be written, where “.n” will be appended to each generation of the file. To keep the last 1000 NOTICE messages for all programs, with 100 messages in each of 10 files:

```
NOTICE: FILE.10.100:/var/log/syslog
```

Multiple routings for the same severity level can be specified by simply adding the additional desired routings to form a semicolon-separated list of `<where>:<parameter>` pairs.

Debug messages are parsed as:

`<comp>:<level>:<where>:<parameter>`

`<comp>` is the component (rpc, sec, cds, dts, dhd, ...)

`<level>` is a comma-separated list of sub-component levels for each component.

`<where>` and parameter are as above.

Each component can have its own entry. Each subcomponent level has the form “<subcomp>.n”, where “n” is 1 to 9; these are parsed in order, so put subcomponent wildcard entries first.

For example, to enable tracing for different components at different levels.

```
dts:*.9:FILE:/tmp/logs/%ld.dts
rpc:*.3:FILE:/tmp/logs/%ld.rpc
sec:*.4:FILE:/tmp/logs/%ld.sec
dhd:*.7:FILE:/tmp/logs/%ld.dhd
cds:*.9:FILE:/tmp/logs/%ld.cds
```

Security Delegation

Intermediary servers can operate on behalf of the initiating client while preserving identities and ACLs.

Compiling Multithreaded Application

There are some new preprocessing directives that has been introduced to compile with latest **Pthread** implementation.

_POSIX_SOURCE: This needs to be used when compiling any DCE application. It is not necessary when compiling threads only application.

_MPE_THREADS: This is used when compiling any DCE or multithreaded application. This directive is used set the thread specific errno.

Applications should ensure that they link `/lib/libpthread.sl` before `/lib/libc.sl` to ensure that they invoke the thread-safe versions of these C runtime. Applications should not use the c89 linking feature directly since it bind the C routines to `/lib/libc.a` before `/lib/libpthread.sl`. Also since we are binding to `/lib/libc.sl`, we need to explicitly link the program object file(s) with the loader “start” routine.

The following makefile will provide an example of the linking process.

```
DEBUG = -g

INCENV = -I. -I/usr/include

ANSI_FLAGS = -D_POSIX_SOURCE -D_POSIX_D10_THREADS

MPE_FLAGS = -D_MPEXL_SOURCE -D_MPE_THREADS -DMPEXL

CFLAGS = ${ANSI_FLAGS} ${DEBUG} ${MPE_FLAGS} ${INCENV}

#LIBS = -lsocket -lsvipc -lm

XL = /lib/libdce.sl, /lib/libpthread.sl, /lib/libc.sl

PROGRAMS = server client

server_OFILES = manager.o server.o sleeper_sstub.o

client_OFILES = sleeper_cstub.o client.o

IDLFLAGS = -keep c_source ${INCENV}

IDLFILES = sleeper.idl

IDLGEN = sleeper.h sleeper_*stub.c

IDL = /SYS/HPBIN/SH /usr/bin/idl

#IDL = /SYS/PUB/IDL

all: objects ${PROGRAMS}
```

```
objects: ${server_OFILES} ${client_OFILES}
fresh: clean all
clean:
          @-rm $(IDLGEN) $(server_OFILES) $(client_OFILES) $(PROGRAMS)
server: ${server_OFILES}
      ar -rc server.obj ${server_OFILES}
      callci linkedit \" 'link from=/SYS/PUB/STARTO, /Speedware/sleeper/server.obj;\
      to=/Speedware/sleeper/server;xl=${XL};posix;share'\"
client: ${client_OFILES}
      ar -rc client.obj ${client_OFILES}
      callci linkedit \" 'link from=/SYS/PUB/STARTO, /Speedware/sleeper/client.obj;\
      to=/Speedware/sleeper/client;xl=${XL};posix;share'\"
sleeper.h: ${IDLFILES}
          $(IDL) ${IDLFLAGS} ${IDLFILES}
sleeper_cstub.o sleeper_sstub.o manager.o server.o client.o: sleeper.h
```

5 Programming with Kernel Threads

Programming with threads is useful for structuring programs, performance enhancement through concurrency and overlapping I/O, and making client/server interaction more efficient (it increases programming complexity). Some things that need to be addressed when programming with threads are:

- Creation and management of threads.
- Threads synchronization and communication.
- Threads scheduling.
- Error handling

A traditional non-threaded process has a single thread of control, started and terminated with the process. Multi-threaded programs require that threads be created and terminated explicitly.

The HP e3000 Kernel Threads provide basic thread creation and management routines.

Threads Synchronization and Communication

All threads in a process execute within a single address space and share resources. When threads share resources in an unsynchronized way, incorrect output can result from race conditions or thread scheduling anomalies. The Kernal Threads provide the following facilities and routines to synchronize thread access to shared resources.

Mutexes (Mutual Exclusion Objects)

Mutexes are used to synchronize access by multiple threads to a shared resource, allowing access by only one thread at a time. Routines for creating and managing mutexes are:

- `pthread_mutex_init(mutex,attr)`
- `pthread_mutex_destroy(mutex)`
- `pthread_mutex_lock(mutex)`
- `pthread_mutex_trylock(mutex)`
- `pthread_mutex_unlock(mutex)`

Condition Variables

Condition variables provide an explicit communication vehicle between threads. A condition variable is a shared resource, and requires a mutex to protect it. A condition variable is used to block one or more threads until a condition becomes true, then any or all of the blocked threads can be unblocked. Routines for creating and managing condition variables are:

- `pthread_cond_init(cond,attr)`
- `pthread_cond_broadcast(cond)`
- `pthread_cond_signal(cond)`
- `pthread_cond_wait(cond,mutex)`
- `pthread_cond_destroy(cond)`

Join Facility

The join facility is the simplest means of synchronizing threads, and uses neither shared resources or mutexes. The join facility causes the calling thread to wait until the specified thread finishes and returns a status value to the calling thread. Routines for joining and detaching threads are:

- `pthread_join(thread,status)`
- `pthread_detach(thread)`

Threads Scheduling

HP e3000 Kernel Threads scheduling is handled through the dispatcher, therefore each thread is visible to and known by the kernel. Altering the scheduling of one or more threads in a task is accomplished with the same tools and methods used to alter the scheduling of any non-threaded task.

NOTE

The HP e3000 Kernel Threads is a POSIX 1003.1 Draft 10 implementation. Individual threads created within a given task may use the same processor at any given time; the threads are independently scheduled by the kernel. Therefore, a multi-threaded process can take advantage of the increased concurrency available on a multi-CPU machine.

Writing Threaded Applications

Useful hints on writing multi-threaded DCE applications:

- All DCE applications are multi-threaded — DCE runtime software is multi-threaded and all DCE applications are multi-threaded; even if the application code itself does not explicitly create threads.
- Using non-thread-safe libraries — When making calls to libraries which are not known to be specifically thread-safe, a locking scheme needs to be provided by the application, For example, non-thread-safe routines 1 and 2 make a call to routine A (also non-thread-safe), if routines 1 and 2 use different mutexes to lock their calls to routine A, then routines 1 and 2 can both get into routine A at the same time (violating the programmer's attempt to make the calls thread-safe).
- Using `fork()` in a threaded application — `fork()` is not allowed from a threaded task.
- **environ** is a process-wide resource — Programmers must coordinate threads that use the `putenv()` and `getenv()` interfaces to change and read **environ**.
- Signal mask: A thread-specific resource — If one thread manipulates the signal mask, it only affects signals that a specific thread is interested in.
- Handling synchronous terminating signals — The default behavior of OSF DCE 1.0.2 is to translate synchronous terminating signals into exceptions. If the exception is not caught, the thread that caused the exception is terminated. Any thread that goes through the terminate code causes the entire task to be terminated.
- Establish synchronous signal handlers using `sigaction()` — It is used to establish handlers for synchronous signals on an individual thread basis only.
- Asynchronous signals — There is no supported mechanism for establishing signal handlers for asynchronous signals on MPE/iX.
- Cancelling threads blocked on a system call — The HP e3000 Kernel Threads provide a cancellation facility that enables one thread to terminate another. The canceled thread normally terminates at a well-defined point. Terminating a thread that is blocked while executing system code is not possible on MPE/iX; only threads executing non-system code may be canceled.
- Using `waitpid()` — The `waitpid()` routine allows the parent thread to specify which child it cares about by specifying its PID. This call only works for the initial thread; because children created by any thread within the task are considered children of the whole

task.

- Using `setjmp` and `longjmp` — Do not use calls to `setjmp` and `longjmp`, these routines save and restore the signal mask and could inadvertently cause a signal that another thread is waiting on to be masked. Instead, use `_setjmp` and `_longjmp`; these routines do not manipulate the signal mask.

  When executing `_longjmp` be aware of the following:

  — Ensure you are returning to a state saved within the context of the same thread.

  — If you `_longjmp` over a **TRY** clause, an exception could try to `_longjmp` to a stack frame that no longer exists; and vice versa.

  — Do not `_longjmp` out of a signal handler.

Writing Thread-Safe Code

The standard C/XL library is not completely thread-safe on the HP e3000. Hewlett-Packard has provided a set of wrapper functions to intercept calls to the C library and make them thread-safe. The threads library takes care of this intercept library implementation.

6 Introduction to RPC

This version of DCE/3000 is based on the OSF DCE version 1.2.1 source code. It provides the Remote Procedure Call component of the core services of OSF DCE.

Remote Procedure Call: supports the development of distributed applications by making requests to remotely networked machines as if they were local. RPCs also implement network protocols used by clients and servers to communicate with each other.

The different components of RPC product are:

RPC Runtime Library:

This is the shared library, which provides the functionality of different RPC APIs. It maintains and manages memory for the RPC application. Runtime basically is the transparent layer, which manages the communication with a remote machine in the network. Along with the help of IDL, runtime makes the RPC protocol possible on a heterogeneous network. The data sent across the network are “Marshalled” before being sent and “Unmarshalls” the incoming data. The functions to do these activities are present in the runtime library.

RPCD:

This is the RPC endpoint mapper daemon. RPCD is a process that provides services for the local host, and is also the server used by remote applications to access these host services. The endpoint mapper service maintains a database called the local endpoint map, which allows DCE clients to find servers, individual services provided by servers, and objects managed by services on the host. The endpoint mapper service maps interfaces, object UUIDs, and protocol sequence registrations to server ports (endpoints). Servers register their bindings with the local endpoint mapper, and the endpoint mapper service on each host uses the local endpoint map to locate a compatible server for clients that do not already know the endpoint of a compatible server.

IDL Compiler:

Interface definition language compiler. IDL compiler to converts an interface definition, written in IDL, into output files. The output files include a header file, server stub file, client stub file, and auxiliary files. The compiler constructs the names of the output files by keeping the basename of the interface definition source file but replacing the filename extension with the new extension (or suffix and extension) appropriate to the newly generated type of output file. For example, math.idl could produce `math_sstub.c` or `math_sstub.o` for the server stub. IDL compiler generates the "stubs" which, when linked with the appropriate modules of the application, makes the RPC communication simple.

In DCE/3000 RPC 1.2.1 version, the DCE library is provided as Shared library (`libdce.sl`). This library contains only the RPC functionality of DCE product.

Table 6-1 indicates the different files in different formats present in the MPE/iX environment as part of the DCE/3000 product.

Table 6-1 RPC Components

Filename	Description
`rpcp.pub.sys`	Command Script
`/usr/bin/rpcp`	Shell Script
`rpcd.hpdce.sys`	Program
`rpccp.pub.sys`	Command Script
`/usr/bin/rpccp`	Shell Script
`rpccp.hpdce.sys`	Program
`idl.pub.sys`	Command Script
`/usr/bin/idl`	Shell Script
`idl.hpdce.sys`	Program

Table 6-2 shows various components.

Table 6-2 Miscellaneous Components

Filename	Description
`/usr/lib/libdce.sl`	Shared library for DCE
`/usr/include/*.h`	Header files for DCE applicaton development
`/usr/include/*.idl`	IDL files for DCE application development

Runtime Library

Private Client Sockets

Previously a common pool of sockets was shared by concurrent RPC requests. Making this concurrency work requires that there be a "helper" thread created to read from all of the open sockets, passing received data onto the call thread for which it is intended. Now with "Private client sockets" there are a couple of sockets (2/3) which will be used only for individual requests (private to the request thread). This reduces the overhead of "helper" thread in case of small applications. However, when we run out of private sockets, the sharing of sockets comes into effect.

Serviceability

This is another new feature, which has been added to RPC runtime. This feature logs messages during the runtime to a specified log file. The level of the messages and the components can be configured using the routing file (`/opt/dcelocal/var/svc/routing`).

This feature will be helpful during analysis of a problem.

By default the routing file is picked from "`/opt/dcelocal/var/svc/routing`". The **DCE_SVC_ROUTING_FILE** environment variable can be used to name an alternate location for the file.

The various switches that can be used are as below:

"general" "mutex" "xmit" "recv" "dg_state" "cancel" "orphan" "cn_state" "cn_pkt" "pkt_quotas" "auth" "source" "stats" "mem" "mem_type" "dg_pktlog" "thread_id" "timestamp" "cn_errors" "conv_thread" "pid" "atfork" "inherit" "dg_sockets" "timer" "threads" "server_call" "nsi" "dg_pkt" "libidl".

The level of messaging ranges from 0-9, where level 9 is the highest level and gives the maximum details. The file to which the logs should be redirected can also be configured. For example: for the RPC, if we want to generate log files with "general" and "cn_pkt" switch enabled at level 9 and the logs to be written to a file named after the process-id of the process, the line would be something like:

```
rpc:general.9,cn_pkt,9:FILE:/tmp/%ld.log
```

Exception Handling

The new version RPC 1.2.1 supports the exception-handling feature of RPC. Now, the application developer can use the exception handling routines (**TRY**, **CATCH**, **CATCH-ALL**, etc.).

DCE-IDL Compiler for RPC 1.2.1

Out-of-Line Marshalling

Out-of-line marshalling (library-based marshalling) causes constructed data types such as unions, pipes or large structures to be marshalled or unmarshalled by auxiliary routines, thus reducing the stub size. The **out_of_line** attribute directs the IDL compiler to place the marshalling and unmarshalling code in IDL auxiliary stub files, rather than in the direct flow of the stub code.

In-line/Out_of_line: The **in_line** and **out_of_line** attributes affect the stub code generated for marshaling and unmarshalling non-scalar parameters (Non-scalar types include int, float, char and pointers in C). Normally IDL compiler generates marshalling and unmarshalling code for all parameters in line. This means that if the same data type is used repeatedly, the identical code will appear in multiple places. If **out_of_line** is specified, the marshaling and unmarshalling code will be provided as a subroutine, which is called from wherever it is needed.

Enhancing IDL Data Types

IDL support for arrays in previous version was limited to:

- Arrays with a lower bound of zero.
- Arrays with conformance or varying dimensions only in the first (major) dimension.

The current version of IDL will remove these restrictions by supporting fully general arrays as described in the IDL functional specification. The following example includes declarations that were not supported in previous version that, but are allowed now:

- long c1[][4];
- long c2[][0..3]; /* Same array shape as c1 */
- long c3[0..*][4]; /* Same array shape as c1 */
- long c4[0..*][0..3]; /* Same array shape as c1 */
- float d1[1..10]; /* Equivalent to FORTRAN REAL D1(10) */
- float d2[*..10]; /* Lower bound is determined at run time */
- float d3[*..*]; /* Both bounds determined at run time */

The <attr_var>s are in one-to-one correspondence with the dimensions of the array, starting at the first. If there are fewer <attr_var>s than the array has dimensions, the missing <attr_var>s are assumed to be null. An <attr_var> will be non-null if and only if the lower bound of the

corresponding dimension is determined at runtime. Not all <attr_var>s in a min_is clause can be null. Below are examples of the syntax. Assume values of variables are as follows: long a = -10; long b = -20; long c = -30; long d = 15; long e = 25.

- min_is(a)] long g1[*..10]; /* g1[-10..10] */
- [min_is(a)] long g2[*..10][4]; /* g2[-10..10][0..3] */
- [min_is(a,b)] long g3[*..10][*..20]; /* g3[-10..10][-20..20] */
- [min_is(,b)] long g4[2][*..20]; /* g4[0..1][-20..20] */
- [min_is(a,,c)] long g5[*..7][2..9][*..8]; /* g5[-10..7][2..9][-30..8] */
- [min_is(a,b,)] long g6[*..10][*..20][3..8]; /* g6[-10..10][-20..20][3..8] */
- [max_is(,,e),min_is(a)] long g7[*..1][2..9][3..*]; /* g7[-10..1][2..9][3..25] */
- [min_is(a,,c),max_is(,d,e)] long g8[*..1][2..*][*..*];/* g8[-10..1][2..15][-30..25] */

Support for IDL Encoding Services

This extension to the IDL stub compiler will enable instances of one or more data types to be encoded into and decoded from a byte stream format suitable for persistent storage without invoking RPC Runtime.

The encode and decode attributes are used in conjunction with IDL Encoding service routines (idl_es*) to enable RPC applications to encode datatypes in input parameters into a byte stream and decode datatypes in output parameters from a byte stream without invoking the RPC runtime. Encoding and decoding operations are analogous to marshaling and unmarshalling, except that the data is stored locally and is not transmitted over the network.

Support for User Defined Exceptions

This extension to the IDL compiler will allow specification of a set of user-defined exceptions that may be generated by the server implementation of the interface. If an exception occurs during the execution of the server, it terminates the operation and the exception is propagated from server to client.

Support for Customized Binding Handles

This allows the application developer to add some information that the application wants to pass between the client and server. This can be used when application-specific data is appropriate to use for finding a server and the data is needed as a procedure parameter.

7 Programming with RPC 1.2.1 on MPE/iX

This chapter explains the RPC application programming with a small example. The example consists of server and client components. The client makes a RPC request to the server and asks the server to sleep for a specified amount of time. The server serves this request from the client by going to sleep for the time given by the client.

```
/*                                      client.c
 ******************************************************************************
 * This is the sleeper client program.It takes two arguments, a hostname to contact
 * for the server and a number of seconds to sleep. The client locates the server
 * using the hostname provided and the endpoint mapper on the server's host -- the
 * client does not contact the name service for server location information. The
 * client uses the explicit binding method, so it uses th hostname argument to
 * construct a binding handle (rpc_binding_handle_t).The client passes this binding
 * handle to the invocation of the remote procedure.
******************************************************************************
 */

/*
 * (c) Copyright 1992, 1993, 1994 Hewlett-Packard Co.
 */
/*
 * @(#)HP DCE/3000 @(#)Module: client.c
*/

#include          <stdlib.h>                  /* Standard POSIX defines */
#include          <strings.h>                 /* str*() routines */
#include          <stdio.h>                   /* Standard IO library */
#include          <dce/dce_error.h>           /* DCE error facility */
#include          <pthread.h>                 /* DCE Pthread facility */
#include          "common.h"                  /* Common defs for this app */
#include          "sleeper.h"                 /* Output from sleeper.idl */

#ifdef    TRACING
tr_handle_t       *tr_handle = NULL;          /* Initialize for client */
#endif                                        /* TRACING */
```

```
void main(int argc, char *argv[])
{
    rpc_binding_handle_tbh;                    /* "points" to the server */
    error_status_t          st, _ignore;       /* returned by DCE calls */
    dce_error_string_t      dce_err_string;    /* text describing error code */
    ndr_char                *string_binding;   /* used to create binding */
    unsigned long           sleep_time;        /* seconds server will sleep */
    unsigned_char_t         *netaddr;          /* network address of server */

#ifdef    TRACING
    /* tr_init() --
     *
     * The tr_init call initializes the trace facility.  The first parameter
     * is the name of an environment variable to consult to determine the
     * values for the selector levels, output filename, etc.  These values
     * can have defaults assigned in the second and third parameters, but
     * this sample application does not choose to do this.  The trace_name
     * parameter is a prefix string that will appear on each line of output
     * to distinguish tracing from this application from other applications.
     */
    if (tr_handle == NULL) {
        tr_handle = tr_init("TR_SLEEPER",      /* environment variable name */
                            NULL,              /* selector level defaults */
                            NULL,              /* filename for output */
                            trace_name);       /* prefix string in output */
   if (tr_handle == NULL) {
    /*
     * Still NULL -- unable to initialize tracing.  This may cause
     * the following tr_printmsg calls (via PRINT_FUNC) to fail.
     */
    fprintf(stderr, "Unable to initialize tracing interface!\n");
   }
 }
#endif /* TRACING */
if (argc != 3) {
         ifprintf(stderr, "Usage: %s hostname sleep_time\n", argv[0]);
         exit(1);
    } else {
```

```
     netaddr = (unsigned_char_t *)argv[1];
     sleep_time = atoi(argv[2]);
}

/*    rpc_string_binding_compose() --
 *
 * Create a string binding using the command line hostname parameter.  A
 * string binding must be converted into a binding handle, required by
 * the DCE runtime, before it can be used.
 *
 * The first parameter is an optional object UUID.  This application does
 * not use multiple object UUIDs, so none is supplied.  The second
 * parameter is the protocol sequence to use to establish a connection;
 * the "ip" parameter selects the UDP/IP protocol.  The third parameter is
 * the network address of the server; this was specified on the command
 * line either as a hostname or as an IP address.
 *
 * The fourth parameter is an endpoint value (IP port number) to use; you
 * should only specify this when creating a string binding if the
 * endpoint is well-known.  Most servers use a dynamic endpoint, chosen
 * when the server starts up; so specify a value of NULL to cause the
 * RPC runtime to determine the value during the RPC setup.  The fifth
 * parameter is for network options.
 *
 * The sixth parameter is the return argument where the string binding
 * will be stored.  New memory will be allocated for this return value;
 * it must be freed later by this application.  The final parameter is a
 * DCE return status which will be checked for errors.
 */
rpc_string_binding_compose(NULL,           /* no object UUID */
                       (unsigned_char_t *)"ip",  /* protocol to use */
                        netaddr,          /* network addr of server */
                        NULL,             /* use a dynamic endpoint */
                        NULL,             /* misc. network options */
                        &string_binding,  /* returned string binding */
                        &st);             /* error status for this call */
if (st != rpc_s_ok) {
       /*       dce_error_inq_text() --
```

```
             *
             * Inquire about the error status returned by the previous DCE call.
             *
             * The first parameter to this call is a DCE error_status_t presumed
             * to have been returned by a preceeding DCE call.  The second
             * parameter is a string long enough to hold the longest possible
             * DCE error string -- the data type dce_error_string_t is defined
             * to be a character array of this length.  The third parameter is
             * another dce error status; this call is unlikely to fail so its
             * status is ignored.
             */
            dce_error_inq_text(st, dce_err_string, (int *)&_ignore);
            PRINT_FUNC(PRINT_HANDLE, "Cannot compose string binding: %s\n",
                  dce_err_string);
            exit(1);
    }

    /*    rpc_binding_from_string_binding() --
     * Create a binding handle structure from the string binding.  The
     * client stub function needs a binding handle; it cannot use the string
     * binding form created above.
     *
     * The first parameter to this call is the string binding generated
     * earlier.  The second parameter is an RPC binding handle structure;
     * a new binding handle will be allocated and stored here -- this
     * application must free the storage when it is done with it.  The third
     * parameter is the DCE return status.
     */
    rpc_binding_from_string_binding(string_binding,     /* created above */
                                    &bh,                /* allocated and returned */
                                    &st);               /* error status for this call
*/
    if (st != rpc_s_ok) {
    dce_error_inq_text(st, dce_err_string, (int *)&_ignore);
    PRINT_FUNC(PRINT_HANDLE, "Cannot get a binding handle: %s\n",
           dce_err_string);
    exit(1);
}
```

```
/*
     * At this point the application is not actually connected to any
     * server, but it has all the information needed to establish a
     * connection to the remote server.  Connection establishment happens in
     * the client stub function called below.
     */
    PRINT_FUNC(PRINT_HANDLE, "Bound to %s\n", string_binding);

    /*    rpc_string_free() --
     *
     * Free a string allocated by the RPC runtime.  The first parameter is
     * the address of a string which was previously allocated (or is NULL).
     * It will be free()d, and the space returned to the system for use in
     * the future.  The second parameter is the DCE return status.
     */
    rpc_string_free(&string_binding,                  /* DCE string to free */
                    &_ignore);                        /* DCE return status */

    PRINT_FUNC(PRINT_HANDLE, "Calling remote_sleep(%d)\n", sleep_time);

    /*    TRY --
     *
     * The macro TRY is used to wrap a call which may result in a DCE
     * exception being raised.  Any call to an RPC client stub can result in
     * an exception being raised if something goes wrong.  Examples of what
     * can go wrong include: there is no server listening on the remote
     * host; there is a data error in a client or server stub; the server
     * raises an exception while executing the procedure call.  If an
     * exception is raised and there is no TRY/CATCH block surrounding the
     * call, the exception will cause the process to abort and dump core.
     * Since this is typically not very helpful, we prefer to catch the
     * exception.  In the string_conv and later sample applications the
     * client will do something intelligent with the exception.
     */
    TRY {
        /*
         * Call the remote procedure passing in the number of seconds to sleep,
         * as defined in the .idl file.  A binding handle parameter is required
```

```
        * since this client uses the explicit binding method.
        */

        remote_sleep(bh, sleep_time);

    /*   CATCH_ALL --
     *
     * The CATCH_ALL macro denotes the end of a TRY block.  If an exception
     * occurs in any of the calls within the TRY block, control will pass to
     * the CATCH_ALL block where the exception is dealt with.  This client
     * will simply inform you that something went wrong; in the string_conv
     * and later sample applications the client will do something
     * intelligent with the exception.
     */
    } CATCH_ALL {
        /*
         * We caught an exception in the client stub code.  Inform the user.
         */
         PRINT_FUNC(PRINT_HANDLE, "Caught an exception!\n");
         exit(1);
    }

    /*   ENDTRY --
     *
     * The ENDTRY macro is required by the exception implementation to
     * terminate a TRY block.
     */
    ENDTRY;

    /*
     * No status information was passed back.  If the call failed, the RPC
     * runtime will have raised an exception and caused an exit.
     */
    PRINT_FUNC(PRINT_HANDLE, "Returned from remote_sleep(%d)\n", sleep_time);

    exit(0);
}
```

```
/*                              manager.c
 ****************************************************************************
 * This is the server-side RPC manager function; this is the function that
 * actually implements the remote procedure defined in the .idl file.  The
 * server stub (called by the RPC runtime) calls this function when an RPC
 * request comes in for this interface.
 *
 * The manager function takes the arguments defined in the .idl file,
 * performs its function and returns results as defined in the .idl file.
 * This particular manager function does not return any results (it does not
 * have any [out] parameters, nor a return value).
 ****************************************************************************
 */

/*
 * (c) Copyright 1992, 1993, 1994 Hewlett-Packard Co.
 */
/*
 * @(#)HP DCE/3000
* @(#)Module: manager.c
*/

#include         <stdlib.h>                /* Standard POSIX defines */
#include         <stdio.h>                 /* Standard IO library */
#include         "common.h"                /* Common defs for this app */
#include         "sleeper.h"               /* Output from sleeper.idl */


/*
 * This particular manager function simply sleeps for the number of seconds
 * specified by its argument.  Since the .idl file speficies use of explicit
 * binding, the manager must take a binding handle as its first argument.
 *
 * Note: the code in this manager function must be (and is) reentrant since it
 * may be running simultaneously in multiple server threads.
 */
```

```
void remote_sleep
    (
     /* [in] */   handle_t       h,        /* Use explicit binding */
     /* [in] */   ndr_long_int   time      /* Seconds to sleep */
     )
{
    PRINT_FUNC(PRINT_HANDLE, "Enter remote_sleep(%d) manager\n", time);
    /*
     * This is a mind-numbingly simple manager ...
     */
    (void) sleep (time);

    PRINT_FUNC(PRINT_HANDLE, "Return from remote_sleep(%d) manager\n", time);
    return;
}
/*                              server.c
 ************************************************************************
 * This is the server program for the basic sleeper sample application.  It
 * will register the interface named "sleeper" with the local RPC runtime
 * and with the endpoint mapper daemon (rpcd) on the local host.  It then
 * listens for incoming requests and serves each request in a separate
 * thread.  The manager function (see manager.c) is invoked to serve the
 * requests after the inbound arguments are unmarshalled.
 ************************************************************************
 */

/*
 * (c) Copyright 1992, 1993, 1994 Hewlett-Packard Co.
 */
/*
 * @(#)HP DCE/3000
 * @(#)Module: server.c
 */

#include          <pthread.h>               /* POSIX threads facility */
#include          <stdlib.h>                /* Standard POSIX defines */
#include          <strings.h>               /* str*() routines */
#include          <stdio.h>                 /* Standard IO library */
```

```
#include          <dce/dce_error.h>          /* DCE error facility */
#include          "common.h"                 /* Common defs for this app */
#include          "sleeper.h"                /* Output from sleeper.idl */

#ifdef   TRACING
tr_handle_t *   tr_handle = NULL;            /* Initialize for server */
#endif                                       /* TRACING */


void main(int argc, char *argv[])
{
    rpc_binding_vector_t  *bvec;             /* used to register w/runtime */
    error_status_t        st, _ignore;       /* returned by DCE calls */
    dce_error_string_t    dce_err_string;    /* text describing error code */
    ndr_char              *string_binding;   /* printable rep of binding */
    int                           i;         /* index into bvec */

#ifdef    TRACING
    /*
     * Initialize tracing.
     */
    if (tr_handle == NULL) {
           char    trace_name_buf[40];
           /*
            * Construct the tracing prefix string from the trace_name constant
            * and the current process id.  This allows multiple servers on the
            * same host to differentiate themselves from each other.
            */
           sprintf(trace_name_buf, "%s-%d", trace_name, getpid());
        tr_handle = tr_init("TR_SLEEPER",        /* environment variable name */
                            NULL,                /* selector level defaults */
                            NULL,                /* filename for output */
                            trace_name_buf);     /* prefix string in output */
           if (tr_handle == NULL) {
              /*
               * Still NULL -- unable to initialize tracing.  This may cause
               * the following tr_printmsg calls (via PRINT_FUNC) to fail.
               */
```

```
            fprintf(stderr, "Unable to initialize tracing interface!\n");
        }
    }
#endif     /* TRACING */

    /*     rpc_server_use_protseq() --
     *
     * Specify the protocol sequences that the RPC runtime should use when
     * creating endpoints.  The first parameter is a string representation
     * of a protocol sequence to use.  The second parameter is the maximum
     * number of concurrent remote procedure call requests that the server
     * will accept.  In the first version of DCE, the second parameter is
     * always replaced by a default value.  The third parameter is the DCE
     * return status.
     *
     * This server uses only the UDP/IP protocol sequence for efficiency
     * reasons: the UDP transport is more efficient for procedures that are
     * idempotent and expected to return only small amounts of data.  The
     * reason why we don't simply listen on all protocols and let the client
     * choose is because it consumes more system resources to listen on
     * multiple protocol sequences.
     */
    rpc_server_use_protseq((unsigned char *)"ip", /* prot seq to listen on */
                           rpc_c_protseq_max_calls_default,
                           &st);            /* error status for this call */
    if (st != rpc_s_ok) {
            dce_error_inq_text(st, dce_err_string, (int *)&_ignore);
        PRINT_FUNC(PRINT_HANDLE, "Cannot use protocol sequence ip: %s\n",
                  dce_err_string);
         exit(1);
    }

    /*    rpc_server_register_if() --
     *
     * Register the interface definition and manager entry point vector with
     * the RPC runtime.  The first parameter is the interface specification
     * generated by the IDL compiler; it is declared in the "sleeper.h" file
     * generated by idl.  The second parameter is the manager type UUID to
```

```
 * associate with the third parameter.  This application does not use
 * type UUIDs (an advanced feature).  The third parameter is the manager
 * entry point vector, the array of functions used as implementations
 * for incoming remote procedure calls.  A value of NULL indicates that
 * the runtime should use the default manager EPV generated by the IDL
 * compiler.  The fourth parameter is the DCE error status.
 */
rpc_server_register_if(sleeper_v1_0_s_ifspec, /* generated interface spec */
                       NULL,                  /* No type UUIDs */
                       NULL,                  /* Use supplied epv */
                       &st);                  /* error status for this call */
if (st != rpc_s_ok) {
       dce_error_inq_text(st, dce_err_string, (int *)&_ignore);
    PRINT_FUNC(PRINT_HANDLE,"Cannot register interface with runtime: %s\n",
            dce_err_string);
     exit(1);
}

/*    rpc_server_inq_bindings() --
 *
 * Inquire from the RPC runtime about the bindings that were created in
 * the registration call above.
 *
 * The first parameter is the address of a binding vector data type.
 * Memory for a new binding vector will be allocated and returned.  The
 * application must later free this memory.  The second parameter is the
 * DCE error status.
 *
 * The binding information is required for registration with the
 * endpoint mapper below.  We print it out simply for debugging
 * purposes.
 */
rpc_server_inq_bindings(&bvec,            /* runtime's binding vector */
                        &st);             /* error status for this call */
if (st != rpc_s_ok) {
        dce_error_inq_text(st, dce_err_string, (int *)&_ignore);
    PRINT_FUNC(PRINT_HANDLE, "Cannot get bindings: %s\n", dce_err_string);
        exit(1);
```

```
} else
    PRINT_FUNC(PRINT_HANDLE, "Bindings:\n");

/*
 * Print out the bindings obtained from the RPC runtime.  This info is
 * only for debugging purposes -- it shows what protocol sequence and
 * ports have been grabbed by the runtime for this server.
 */
for (i = 0; i < bvec->count; i++) {
       /*       rpc_binding_to_string_binding() --
        *
        * Convert a binding handle to a string binding for printing.  The
        * first parameter is a binding handle.  (In a binding vector there
        * are bvec->count binding handles).  The second parameter is a
        * pointer to a dce string data type; memory will be allocated and
        * the value returned in it.  The application must free this memory.
        * The third parameter is the DCE error status.
        */
    rpc_binding_to_string_binding(bvec->binding_h[i], /* a binding handle */
                            &string_binding,  /* returned string form */
                            &st);             /* error status for this call */
    if (st != rpc_s_ok) {
           dce_error_inq_text(st, dce_err_string, (int *)&_ignore);
        PRINT_FUNC(PRINT_HANDLE, "Cannot get string binding: %s\n",
                 dce_err_string);
        } else
          PRINT_FUNC(PRINT_HANDLE, "  %s\n", string_binding);

       /*
        * Free the memory allocated in rpc_binding_to_string_binding().
        */
       rpc_string_free(&string_binding, &_ignore);
}

/*    rpc_ep_register() --
 *
 * Register the interface with the endpoint mapper.  The first parameter
 * is the interface specification generated by the IDL compiler.  The
```

```
 * second parameter is the binding vector returned by the RPC runtime
 * describing the endpoints (IP ports) on which this server is listening
 * for RPC requests.  The third parameter is a vector of object UUIDs
 * that the server offers; this server does not implement multiple
 * objects so it specifies NULL.  The fourth parameter is an annotation
 * used for informational purposes only.  The RPC runtime does not use
 * this string to determine which server instance a client communicates
 * with, or for enumerating endpoint map elements.  The last parameter
 * is the DCE error status.
 *
 * When this call completes the bindings we established with the RPC,
 * runtime will be associated with this interface.  This allows a client
 * to look up a server by interface without specifying an endpoint
 * (port): instead, by contacting the endpoint mapper, a client is able
 * to locate servers registered using dynamic (system-chosen) endpoints.
 */
rpc_ep_register(sleeper_v1_0_s_ifspec,  /* generated interface spec */
                bvec,                   /* runtime's binding vector */
                NULL,                   /* no objects supported */
                (unsigned_char_t *) sleeper_description,
                &st);                   /* error status for this call */
if (st != rpc_s_ok) {
       dce_error_inq_text(st, dce_err_string, (int *)&_ignore);
       PRINT_FUNC(PRINT_HANDLE, "Cannot register with endpoint map: %s\n",
              dce_err_string);
       exit(1);
}

PRINT_FUNC(PRINT_HANDLE, "Listening...\n");
/*    rpc_server_listen() --
 *
 * Listen and handle incoming RPC requests.  This call typically does
 * not return; instead incoming RPC requests will be dispatched to the
 * manager function(s), each in its own thread.
 *
 * The first parameter is the maximum number of concurrently executing
 * remote procedure calls to allow.  The second parameter is the DCE
 * error status.
```

```
 */
rpc_server_listen(rpc_c_listen_max_calls_default,
                    &st);                /* error status for this call */

if (st != rpc_s_ok) {
        dce_error_inq_text(st, dce_err_string, (int *)&_ignore);
        PRINT_FUNC(PRINT_HANDLE, "Listen returned with error: %s\n",
               dce_err_string);
} else
        PRINT_FUNC(PRINT_HANDLE, "Stopped listening...\n");

/*********************************************************************
 * IMPORTANT NOTE: We will probably never reach here.  If you interrupt
 * the server with an asynchronous signal, such as a ^C (or SIGINT) from
 * the keyboard or a "kill <PID>" (a SIGTERM signal), it will cause the
 * process to exit; it will not reach here.  See the lookup sample
 * application for code that is able to properly clean up after the
 * listen call.
 *********************************************************************/

PRINT_FUNC(PRINT_HANDLE, "Unregistering endpoints and interface...\n");

/*    rpc_ep_unregister() --
 *
 * Unregister the interface and endpoints with the RPC runtime.  The
 * first parameter is the interface specification from the IDL compiler.
 * The second parameter is the binding vector registered with this
 * interface.  The third parameter is the object UUID vector (NULL since
 * this application does not support multiple objects).  The final
 * parameter is the DCE error status.
 */
rpc_ep_unregister(sleeper_v1_0_s_ifspec,      /* IDL-generated ifspec */
                bvec,                         /* this server's bindings */
                NULL,                         /* no object UUIDs supported */
                &_ignore);                    /* ignore any errors */

/*    rpc_binding_vector_free() --
 * Free a binding vector that is no longer needed.  Since it was
```

```
     * allocated by the runtime, the application should remember to free it.
     * The first parameter is the binding vector to free; the second
     * parameter is the DCE error status, which is ignored.
     */
    rpc_binding_vector_free(&bvec, &_ignore);

    /*    rpc_server_unregister_if() --
     *
     * Unregister this server from the RPC runtime.  This is unnecessary
     * since this process is about to exit, but is here to demonstrate good
     * programming style.  The first parameter is the interface
     * specification; the second is the manager type UUID (which is NULL
     * since this application does not support multiple types).  The last
     * parameter is the DCE error status, which is ignored.
     */
    rpc_server_unregister_if(sleeper_v1_0_s_ifspec,      /* IDL-generated ifspec */
                             NULL,                       /* No object UUID */
                             &_ignore);                  /* ignore any errors */

    exit(0);
}
/*                              common.h
 ************************************************************************
 * This file contains definitions common between the client and server.
 ************************************************************************
 * (c) Copyright 1992, 1993, 1994 Hewlett-Packard Co.
 */
/*
 * @(#)HP DCE/3000 1.5
 * @(#)Module: common.h
 */
/*
 * This string will be registered with the RPC runtime as an annotation
 * describing the endpoint entry.
 */
# define    sleeper_description       "sleeper"

#ifdef     TRACING
```

```
/*
 * If you want to use the building blocks tracing facility then define the TRACING
 * flag in your compile (put -DTRACING in the Makefile).  For this to compile and
 * link, you will need the building blocks library installed on your system.
 */
#include      <dce/trace_log.h>              /* Building blocks tracing */

extern   tr_handle_t *      tr_handle;       /* used by client, server */

/*
 * These print functions use the trace/log facility instead of stdio. All print
 * statements in this file use these macros so it's easy to replace use of stdio with
 * the trace/log facility. The NULL after tr_handle signifies the use of the 500
 * byte, default buffer for trace output.
 */
# define     RINT_FUNC     tr_printmsg
# define     PRINT_HANDLE  tr_handle, NULL

/*
 * The trace_name string is registered with the trace/log facility as the
 * name of this application.  It will appear in any tracing output.
 */
# define     trace_name    sleeper_description

#else   /* TRACING */

/*
 * These print functions use stdio instead of the trace/log facility.  They
 * turn off the tracing macros by replacing them with standard IO routines.
 */
# define     PRINT_FUNC        fprintf
# define     PRINT_HANDLE      stdout

#endif    /* TRACING */
```

```
/*                                sleeper.idl
 ************************************************************************
 * This .idl file declares an interface with a set of remotely-callable
 * procedures.  This file is compiled by the idl compiler into a C interface
 * declaration (.h) and a C client stub (_cstub.c) and server stub
 * (_sstub.c) that interface with the RPC runtime.  You must write a manager
 * for this procedure (see manager.c) and the client and server main()
 * functions (see client.c and server.c).
 ************************************************************************/
/*
 * (c) Copyright 1992, 1993, 1994 Hewlett-Packard Co.
 */
/*
 * @(#)HP DCE/3000
 * @(#)Module: sleeper.idl
 */

/*
 * This definition declares the interface for this application and
 * associates it with a globally (universally) unique identifier, or UUID.
 * The RPC runtime uses the UUID to identify this interface.  If you
 * leverage this code, BE SURE TO CHANGE THE UUID!  Do this by running the
 * program "uuidgen" and putting the uuidgen output in place of the one
 * supplied.  Failure to do this may cause bizarre results.
 */
[uuid(D0FCDD70-7DCB-11CB-BDDD-08000920E4CC),      /* NOTE: CHANGE THIS!!! */
 version(1.0)]
interface sleeper
{
    void remote_sleep
         (
         [in] handle_t    h,          /* Use explicit binding */
         [in] long        time        /* Seconds to sleep */
         );
}
```

Compiling Multithreaded Application

There are some new preprocessing directives that has been introduced to compile with latest Pthread implementation.

_POSIX_SOURCE: This needs to be used when compiling any DCE application. It is not necessary when compiling threads-only application.

_MPE_THREADS: This is used when compiling any DCE or multithreaded application. This directive is used to set the thread specific errno.

Applications should ensure that they link `/lib/libpthread.sl` before `/lib/libc.sl` to ensure that they invoke the thread-safe versions of these C runtime. Applications should not use the c89 linking feature directly since it binds the C routines to `/lib/libc.a` before `/lib/libpthread.sl`. Since we are binding to `/lib/libc.sl`, we need to explicitly link the program object file(s) with the loader “start” routine.

The following makefile will provide an example of the linking process.

```
DEBUG = -gINCENV = -I. -I/usr/include

ANSI_FLAGS = -D_POSIX_SOURCE -D_POSIX_D10_THREADS

MPE_FLAGS = -D_MPEXL_SOURCE -D_MPE_THREADS -DMPEXL

CFLAGS = ${ANSI_FLAGS} ${DEBUG} ${MPE_FLAGS} ${INCENV}

#LIBS = -lsocket -lsvipc -lm

XL = /lib/libdce.sl, /lib/libpthread.sl, /lib/libc.sl

PROGRAMS = server client

server_OFILES = manager.o server.o sleeper_sstub.o

client_OFILES = sleeper_cstub.o client.o

IDLFLAGS = -keep c_source ${INCENV}

IDLFILES = sleeper.idl
```

```
IDLGEN = sleeper.h sleeper_*stub.c
IDL = /SYS/HPBIN/SH /usr/bin/idl

#IDL = /SYS/PUB/IDL

all: objects ${PROGRAMS}

objects: ${server_OFILES} ${client_OFILES}

fresh: clean all

clean:

          @-rm $(IDLGEN) $(server_OFILES) $(client_OFILES) $(PROGRAMS)

server: ${server_OFILES}

      ar -rc server.obj ${server_OFILES}

      callci linkedit \" 'link from=/SYS/PUB/STARTO, /Speedware/sleeper/server.obj;\

      to=/Speedware/sleeper/server;xl=${XL};posix;share'\"

client: ${client_OFILES}

      ar -rc client.obj ${client_OFILES}

      callci linkedit \" 'link from=/SYS/PUB/STARTO, /Speedware/sleeper/client.obj;\

      to=/Speedware/sleeper/client;xl=${XL};posix;share'\"

sleeper.h: ${IDLFILES}

          $(IDL) ${IDLFLAGS} ${IDLFILES}

sleeper_cstub.o sleeper_sstub.o manager.o server.o client.o: sleeper.h
```

Index

Index