Application Development for the Lightweight Client Framework

Release 3.2

July 31, 1997
Application Development for the Lightweight Client Framework
(July, 1997)

Copyright Notice

Copyright © 1997 by Tivoli Systems, an IBM Company, including this documentation and all software. All rights reserved. May only be used pursuant to a Tivoli Systems Software License Agreement or Addendum for Tivoli Products to IBM Customer or License Agreement. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without prior written permission of Tivoli Systems. The document is not intended for production and is furnished “as is” without warranty of any kind. All warranties on this document are hereby disclaimed including the warranties of merchantability and fitness for a particular purpose.

Note to U.S. Government Users—Documentation related to restricted rights—Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.

Trademarks

The following product names are trademarks of Tivoli Systems or IBM Corporation: AIX, IBM, OS/2, RISC System/6000, Tivoli Management Environment, and TME 10.

Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names mentioned in this document may be trademarks or servicemarks of others.

Notice

References in this publication to Tivoli Systems or IBM products, programs, or services do not imply that they will be available in all countries in which Tivoli Systems or IBM operates. Any reference to these products, programs, or services is not intended to imply that only Tivoli Systems or IBM products, programs, or services can be used. Subject to Tivoli System’s or IBM’s valid intellectual property or other legally protectable right, any functionally equivalent product, program, or service can be used instead of the referenced product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by Tivoli Systems or IBM, are the responsibility of the user.

Tivoli Systems or IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, New York 10594.

Patents may be pending.
Preface........................................................................................................................................... ix

Chapter 1—Introduction to LCF
Introduction..................................................................................................................................... 1-1
LCF Architecture ......................................................................................................................... 1-2

Chapter 2—LCF Application Design
Introduction.................................................................................................................................... 2-1
Methods in the Full Framework Environment ............................................................................. 2-2
Methods in the LCF Environment ................................................................................................. 2-3
LCF Objects ................................................................................................................................ 2-4
Endpoint Methods ....................................................................................................................... 2-5
Gateway Methods ........................................................................................................................ 2-6
Downcalls and Upcalls ................................................................................................................. 2-6
  Downcalls ................................................................................................................................... 2-7
  Upcalls ........................................................................................................................................ 2-8
    General Information about Upcalls ......................................................................................... 2-8
    The Sequence of an Upcall ...................................................................................................... 2-8
Upcall Architecture ..................................................................................................................... 2-10
  The Upcall Collector ................................................................................................................ 2-11
    Upcall Collector: An Abstract Object .................................................................................. 2-11
    Persistent Store ..................................................................................................................... 2-12
    IDL Definition ...................................................................................................................... 2-13
Upcall Authorization .................................................................................................................. 2-13
Upcalls and TME Principals ...................................................................................................... 2-14
  Single Upcall Authorization .................................................................................................... 2-14
  Principal Login ......................................................................................................................... 2-14
  Data Encryption ....................................................................................................................... 2-14
Using the Log File in lcfd and libmrt ....................................................... 4-12
Using the Log File in Methods................................................................. 4-12
  Summary of Logging Functions ...................................................... 4-13
  LogInit ............................................................................................. 4-13
  LogDeinit......................................................................................... 4-14
  LogMsg ........................................................................................... 4-15
  LogSetDefault.................................................................................. 4-16
  LogGetDefault ................................................................................. 4-17
  LogSetThreshold.............................................................................. 4-17
  LogGetThreshold ............................................................................. 4-18
  LogSetOutputStdout ........................................................................ 4-18
  LogGetOutputStdout ........................................................................ 4-19
  LogSetAppName ............................................................................. 4-19
  LogAppName ................................................................................... 4-20
  LogQ ................................................................................................ 4-20
  LogQueueAlloc................................................................................ 4-21
  LogQueueDealloc ............................................................................ 4-21
  LogQueueGetSize............................................................................ 4-21
  LogQGetBuffer................................................................................ 4-21
  LogData ........................................................................................... 4-22
  ADR Marshalling Functions ................................................................. 4-22
  C Data Types ............................................................................................ 4-23
  Creating Data Types Using adrgen .......................................................... 4-23
  Using ADR Types for Encoding and Decoding ....................................... 4-24
  Functions for Launching Processes .......................................................... 4-25
  Miscellaneous Functions ................................................................................... 4-27
  set_current_credentials ........................................................................ 4-28
  set_lang ............................................................................................. 4-28
  still_alive .................................................................................................. 4-28
  ep_stream_read.................................................................................... 4-29
Chapter 5—The Common Porting Layer

Introduction......................................................................................................... 5-1
Binary Tree Search Functions............................................................................. 5-2
cpl_correct_path.................................................................................................. 5-2
cpl_get_current_location .................................................................................... 5-3
cpl_fflush ............................................................................................................ 5-3
Directory Entry (dirent) Functions ..................................................................... 5-4
cpl_getcwd ......................................................................................................... 5-4
cpl_getenv and cpl_putenv......................................................................... 5-4
cpl_getopt................................................................................................... 5-5
cpl_getpass ................................................................................................. 5-5
cpl_gettimeofday........................................................................................ 5-5
cpl_gethostname......................................................................................... 5-6
cpl_ltoa................................................................................................................ 5-6
MSVC stat(2) Macros......................................................................................... 5-6
cpl_fprintf................................................................................................... 5-7
cpl_fclose ................................................................................................... 5-7
cpl_fopen.................................................................................................... 5-7
cpl_getc ...................................................................................................... 5-8
Callback Registration Functions ................................................................ 5-8
Temporary File Functions................................................................................. 5-10
cpl_tmpfile ............................................................................................... 5-10
cpl_tmpnam.............................................................................................. 5-10
cpl_tmpdir ................................................................................................ 5-11
Thread Yield (systhread).................................................................................. 5-11
UCB Compatibility Functions .......................................................................... 5-11
uname................................................................................................................ 5-11
wstat Macros ..................................................................................................... 5-12
Removing and Inserting Elements in a Queue.................................................. 5-13
Chapter 6—Dependencies

Specifying Dependencies................................................................. 6-2
Implementing Dependencies.............................................................. 6-2
Commands to Specify Dependencies.................................................. 6-2
   The wdepset Command................................................................. 6-3
      Creating a Dependency Set......................................................... 6-3
      Deleting a Dependency Set......................................................... 6-4
      Viewing a Dependency Set......................................................... 6-4
      Resolving a Dependency Set....................................................... 6-4
   The wchdep Command................................................................. 6-4
      Updating All Dependencies......................................................... 6-5
      Clearing Out-of-Date Dependencies......................................... 6-5
Out-of-Cache Dependencies............................................................. 6-5
Example .......................................................................................... 6-7

Chapter 7—CCMS for LCF Applications

Using CCMS for Endpoint Applications ........................................... 7-1
CCMS Functions Supported in LCF.................................................. 7-2
CCMS in the LCF Environment.......................................................... 7-3
   Profile Managers in Dataless and Database Mode....................... 7-4
      Database Mode ................................................................. 7-5
      Dataless Mode .................................................................. 7-7
IDL File.......................................................................................... 7-9
Header Files .................................................................................... 7-10
CCMS Library .................................................................................. 7-10

Chapter 8—Task Library

Using the Task Library Data Types..................................................... 8-1
Task Library.................................................................................... 8-2
IDL File.......................................................................................... 8-2
Header Files .................................................................................... 8-2
Chapter 9—LCF Sample Application

The Source Tree.................................................................................................. 9-2
Source Files......................................................................................................... 9-3
  Files in the Common Directory......................................................... 9-3
  Files in the Endpoint Directory.............................................. 9-4
  Files in the Platform Directory.................................................. 9-4
Sequence of Steps for Writing and Building an LCF Application............... 9-5
The IDL Files (Common) ............................................................ 9-7
  The IDL Definition File: Samp.idl........................................ 9-7
  The Implementation File: Samp.imp.................................. 9-10
  The Program File: Samp.prog........................................ 9-11
  The Installation File: Samp.ist .................................. 9-13
Endpoint Methods..................................................................................... 9-15
  ep_meth.c ................................................................. 9-15
  up.c ........................................................................... 9-17
Platform Files............................................................................................. 9-18
  samp.c................................................................. 9-18
  upserv.c ................................................................. 9-21
  my_t_MLM_meth.c .................................................. 9-24
  my_t_UpcallCollector_meth.c .................................. 9-24
Where Each Piece Executes.............................................................................. 9-28
Building the record_gen Sample Application................................................... 9-28
  Building the Application .................................................. 9-28
  The Export Tree .......................................................... 9-29
  Endpoint and Platform Makefiles ................................... 9-30
    Creating a Multi-Platform Environment .......................... 9-31

Appendix A—Reference

wchdep................................................................. A-3
wdepset ................................................................. A-4
wgetdeps ................................................................. A-7
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogData</td>
<td>A-8</td>
</tr>
<tr>
<td>LogDeinit</td>
<td>A-9</td>
</tr>
<tr>
<td>LogGetAppName</td>
<td>A-10</td>
</tr>
<tr>
<td>LogGetDefault</td>
<td>A-11</td>
</tr>
<tr>
<td>LogGetOutputStdout</td>
<td>A-12</td>
</tr>
<tr>
<td>LogGetThreshold</td>
<td>A-13</td>
</tr>
<tr>
<td>LogInit</td>
<td>A-15</td>
</tr>
<tr>
<td>LogMsg</td>
<td>A-17</td>
</tr>
<tr>
<td>LogQ</td>
<td>A-20</td>
</tr>
<tr>
<td>LogQueueAlloc</td>
<td>A-22</td>
</tr>
<tr>
<td>LogQueueDealloc</td>
<td>A-23</td>
</tr>
<tr>
<td>LogQueueGetSize</td>
<td>A-24</td>
</tr>
<tr>
<td>LogQGetBuffer</td>
<td>A-25</td>
</tr>
<tr>
<td>LogSetAppName</td>
<td>A-26</td>
</tr>
<tr>
<td>LogSetDefault</td>
<td>A-27</td>
</tr>
<tr>
<td>LogSetOutputStdout</td>
<td>A-28</td>
</tr>
<tr>
<td>LogSetThreshold</td>
<td>A-29</td>
</tr>
<tr>
<td>close_ex</td>
<td>A-31</td>
</tr>
<tr>
<td>copy_file_ex</td>
<td>A-32</td>
</tr>
<tr>
<td>does_file_exist</td>
<td>A-33</td>
</tr>
<tr>
<td>get_file_length_ex</td>
<td>A-34</td>
</tr>
<tr>
<td>get_open_file_length_ex</td>
<td>A-35</td>
</tr>
<tr>
<td>lseek_ex</td>
<td>A-36</td>
</tr>
<tr>
<td>make_path</td>
<td>A-38</td>
</tr>
<tr>
<td>mkdir_ex</td>
<td>A-39</td>
</tr>
<tr>
<td>open_ex</td>
<td>A-40</td>
</tr>
<tr>
<td>read_ex</td>
<td>A-42</td>
</tr>
<tr>
<td>write_ex</td>
<td>A-44</td>
</tr>
<tr>
<td>rename_file_ex</td>
<td>A-46</td>
</tr>
<tr>
<td>decrypt_data</td>
<td>A-47</td>
</tr>
<tr>
<td>encrypt_data</td>
<td>A-49</td>
</tr>
</tbody>
</table>
ep_stream_read ................................................................. A-51
ioch_recv ........................................................................... A-52
mrt_set_method_exit_mode ........................................ A-53
mrt_machine_id ............................................................... A-55
mrt_check_dependency .................................................. A-56
nw_echo_command_to_console ...................................... A-57
set_lang ............................................................................ A-58
still_alive ........................................................................ A-59
tiv_io_create .................................................................... A-60
tiv_io_destroy ................................................................. A-62
tiv_spawn .......................................................................... A-63
tiv_spawn_ui ................................................................. A-65
tiv_user_token_create .................................................. A-67
tiv_user_token_destroy ................................................ A-69
tiv_wait ........................................................................... A-70
Preface

*Application Development for the Lightweight Client Framework* provides information about how to develop applications to run on the Lightweight Client Framework (LCF). It introduces the LCF, describes the programming environment, discusses the special application library for use by LCF applications, and describes the components of an LCF application.

**Who Should Read This Guide**

This guide is intended for developers who plan to write Tivoli Management Environment 10 (TME 10) ADE applications to run on endpoints in an LCF environment. Readers of this manual should have a knowledge of the following:

- The TME 10 Advanced Development Environment (TME 10 ADE)
- The C programming language
- The UNIX operating system

**Prerequisite and Related Documents**

This document supplements the TME 10 ADE document set and assumes an understanding of TME 10 ADE tools, concepts, and services.

If you are new to the TME 10 and TME 10 ADE, you should familiarize yourself with TME 10 ADE before beginning this guide.

Related TME 10 Framework documentation includes:

- The *TME 10 Framework Planning and Installation Guide* provides introductory information about the LCF. It includes brief descriptions of the LCF and the resources it adds to the TME 10. It also includes instructions for installing and creating each LCF component.

Related ADE documentation includes:

- The *Introduction to TME 10 ADE* provides an overview of TME 10, as well as the Common Object Request Broker Architecture (CORBA) and application services it provides.
Preface

- The *Framework Services Manual* covers the TME 10 Framework, including CORBA and the Interface Definition Language (IDL), as well as Tivoli’s Extended IDL (TEIDL).

- The *Application Services Manual, Volumes 1 and 2*, discuss the TME 10 ADE application services, including transactions, exceptions, notification, scheduling, and the change control management system (CCMS).

- *Application Development with TME 10 ADE* provides a roadmap to application development by stepping through the design and implementation of a sample TME 10 ADE application. This application is called Tivoli/Phone. The source code to the Tivoli/Phone application is available, and you are welcome to modify and re-use this code.

- The *Desktop Services Manual* provides in-depth information about the TME 10 ADE desktop services. It discusses the services available to create user interfaces for an application (both graphical and non-graphical).

**What This Manual Contains**

*Application Development for the Lightweight Client Framework* contains these chapters:

- Chapter 1, “Introduction to LCF”
  Introduces LCF and the new components it adds to the TME 10 Framework architecture.

- Chapter 2, “LCF Application Design”
  Describes elements in the LCF application development environment, such as LCF objects, endpoint methods, and gateway methods. Also describes the components of an LCF application, such as endpoint, upcall collector, and server components.

- Chapter 3, “The LCF Programming Environment”
  Describes the programming tools, compiler, and debugging tools you use in developing LCF applications.

- Chapter 4, “The LCF Application Runtime Library”
  Describes the application programming interfaces (APIs)
available in the application runtime library (libmrt) for use in LCF applications.

- Chapter 5, “The Common Porting Layer”
  Describes libcpl, the common porting layer for LCF.

- Chapter 6, “Dependencies”
  Describes how to specify and manipulate method dependencies in LCF applications.

- Chapter 7, “CCMS for LCF Applications”
  Describes the CCMS functions available to your LCF application and how to include them in your code.

- Chapter 8, “Task Library”
  Describes how to include data types from the Task library in your LCF application.

- Chapter 9, “LCF Sample Application”
  Describes a sample LCF application: its structure, the build environment, and the steps in writing an LCF application.

- Appendix A, “Reference”
  Contains manual pages for libmrt functions.

Conventions Used in This Guide

The guide uses several typeface conventions for special terms and actions. These conventions have the following meaning:

**Bold**

Commands, keywords, file names, or other information that you must use literally appear in **Bold**. Names of windows, dialogs, and other controls also appear in **bold**.

**italics**

Variables and values that you must provide appear in *italics*.

**Bold Italic**

New terms appear in **bold italic** the first time they are used.

**Monospace**

Code examples appear in a **monospace** font.
### Platform-Specific Information

The following table identifies text used to indicate platform-specific information or procedures:

<table>
<thead>
<tr>
<th>Text</th>
<th>Supported Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIX 3.2.5</td>
<td>IBM RS/6000 series running AIX 3.2.5</td>
</tr>
<tr>
<td>AIX 4.x</td>
<td>IBM RS/6000 series running AIX 4.1 or 4.2</td>
</tr>
<tr>
<td>HP-UX 9.x</td>
<td>HP9000/700 and 800 series running HP-UX 9.00, HP-UX 9.01, HP-UX 9.03, and HP-UX 9.05</td>
</tr>
<tr>
<td>HP-UX 10.x</td>
<td>HP9000/700 and 800 series running HP-UX 10.01 or HP-UX 10.10</td>
</tr>
<tr>
<td>NetWare</td>
<td>IBM-compatible PCs 486 or higher running Novell NetWare 3.11, 3.12, or 4.0</td>
</tr>
<tr>
<td>OS/2</td>
<td>IBM-compatible PCs 486 or higher running IBM OS/2 Warp 3.0 or Warp 4.0.</td>
</tr>
<tr>
<td>Solaris</td>
<td>Sun SPARC series running Solaris 2.3, 2.4, 2.5, or 2.5.1</td>
</tr>
<tr>
<td>SunOS</td>
<td>Sun SPARC series running SunOS 4.1.2 or 4.1.3</td>
</tr>
<tr>
<td>Windows</td>
<td>IBM-compatible PC 486 or higher running Microsoft Windows 3.1, 3.11, or 3.11 for WorkGroups</td>
</tr>
<tr>
<td>Windows95</td>
<td>IBM-compatible PC 486 or higher running Microsoft Windows95</td>
</tr>
<tr>
<td>Windows NT</td>
<td>IBM-compatible PC 486 or higher running Microsoft Windows NT 3.51 or 4.0</td>
</tr>
</tbody>
</table>

### Contacting Customer Support

We are very interested in hearing from you about your experience with the products in the Tivoli Management Environment 10. We welcome your suggestions for improvements.

If you encounter difficulties with any TME 10 product, please contact your customer support representative. To assist you, the TME 10 Framework includes the `wsupport` command. This command prompts
you for problem information, which can be E-mailed to your support provider or saved to a text file. You can then print the saved file, and fax the resulting TME Problem Report form to your support provider. See the *TME 10 Framework Reference Manual* for additional information about the `wsupport` command.

If you have comments or suggestions about the TME 10 documentation, please send E-mail to pubs@tivoli.com.
Preface
Introduction to LCF

Introduction

The lightweight client framework (LCF) is an enhancement to the TME 10 Framework. The LCF rearchitects the client/server hierarchy created by the TME 10 Framework and extends the Framework’s scalability.

In versions 3.1 and earlier, you install the TME 10 Framework on each Tivoli Management Region (TMR) server and on each client system managed by the TMR server. Each TME 10 client also maintains its own database, which can be as large as 10 MB.

The LCF enables you to install a very small amount of code on client systems (known as endpoints) that are used for desktop operations, such as monitoring local operations or receiving management distributions from TME 10 profile managers. Endpoints do not maintain a client database or run the TME 10 desktop or command line interface (CLI) commands. The software installed on an endpoint requires only 1 to 2 MB of disk space, compared to the 10 to 20 MB required for a managed node or TMR server.

Endpoints provide an important advantage: you can gather management information from them and remotely manage them without the need for much disk space on each endpoint.

Today, a single TMR server can handle communications with up to approximately 200 managed nodes. To handle more than 200
In the LCF, communications with endpoints are handled by an **endpoint gateway** instead of the TMR server. Endpoint gateways are installed on the TMR server or on managed nodes, which run the full TME 10 Framework. A single endpoint gateway can handle the communications of thousands of endpoints.

The increase in scalability is easily seen when you consider a TMR server with 200 managed nodes, each of which has thousands of endpoints. A TMR can now include thousands of TME 10 clients.

### LCF Architecture

The LCF architecture adds the following new components to the TME 10 Framework:

- **endpoint**: An endpoint is a UNIX workstation or PC running the LCF. It is not used for the daily management operations in a network. It is managed from other TME 10 managed nodes and receives profile distributions, executes tasks, runs monitors for TME 10 Distributed Monitoring, and sends alerts to TME 10 Enterprise Console.

- **gateway**: An endpoint gateway (or gateway) runs on a managed node, which has the full TME 10 Framework installed. It provides communications between a group of endpoints and the rest of the TME 10 environment. The gateway also has the **MDist** repeater functionality built in, enabling it to act as the fan-out point for distributions to a very large number of endpoints.

- **endpoint manager**: The endpoint manager runs on the TMR server and assigns endpoints to gateways. It also maintains an endpoint list, which contains all information necessary to uniquely identify endpoints, including which gateway the endpoint is using.
These new components change the hierarchical look of a TME 10 installation. Instead of the two-tiered structure (TMR server and managed node) seen in previous releases, the LCF supports a three-tiered hierarchy, as shown in the following figure.

However, you are not required to have three tiers. In a smaller workgroup-size installation, you can create the endpoint gateway on the TMR server. The following figure illustrates a workgroup configuration:
In summary, the following are key features of the LCF:

- The three-tier structure of LCF—endpoint manager, gateway, and endpoint.
- The nature of the endpoints themselves—small, dataless clients that have no client database and that do not have the full TME 10 Framework installed.
- The scalability possible within the LCF environment—one gateway can manage thousands of endpoints.
- The secure, CORBA-based connection between the LCF endpoint and the rest of the TMR.

The following figure illustrates the new components of the LCF and shows their relationship to components in the existing TME 10 Framework. It shows a TMR server running the endpoint manager, a managed node running an endpoint gateway, and an LCF endpoint.
For information about installing, using, and administering the LCF, see the *TME 10 Framework User’s Guide* and the *TME 10 Framework Planning and Installation Guide*. 
Introduction

In the LCF environment, applications run within enterprises that may contain tens of thousands of endpoints. Enterprises this large contain mainly endpoints, and the TMR server and managed nodes serve as support for the large number of endpoints.

To run optimally, applications must be adapted to and reflect this three-tier structure. Specifically, applications need to be designed and written to work with endpoints and gateways.

The application development environment for LCF is very similar to that of the full framework (where the TME 10 Framework and a client database is installed on each client). However, there are some differences in how methods are executed for endpoints. These differences affect how LCF applications should be designed.

This chapter presents information about how to design and develop applications for an endpoint running the LCF. The chapter first introduces and describes the elements of the LCF, such as LCF objects, endpoint methods, and gateway methods. It then discusses the components of an LCF application. The following topics are covered:

- How methods run in the full framework and in the LCF environment
- LCF objects
- Endpoint methods
Methods in the Full Framework Environment

- Gateway methods
- Downcalls and upcalls
- Upcall architecture
- Components of an LCF application
- LCF application scenarios

This chapter, like this guide, assumes you are familiar with and have previous experience using the TME 10 Application Development Environment (ADE) and application development in the full framework environment. For more information about application development in the full framework, see the TME 10 ADE document set.

Methods in the Full Framework Environment

Before discussing how methods run in the LCF, it is helpful to briefly describe how they run in the full framework environment.

In the full framework, the enterprise consists of a TMR server and one or more managed nodes. All methods are executed in the same way, whether on the TMR server or on a managed node. From any system within the TMR, you can execute a method on any other system. To execute a method, you must have the object reference, which consists of the TMR region number, the dispatcher number, and the object number.

The calling application (a Perl script, a shell script, or a program) passes the object reference to a managed node. The TMR server resolves it through the object dispatcher, locates the methods for that object, retrieves the method header, and invokes the method. The method starts, the parameters are passed to it, and the results are passed back to the caller.

The key point is that, from any system in the TMR, you can run a method on any object on any other system in the TMR.
Methods run differently in the LCF environment than in the full framework environment. This section introduces and describes several key differences.

In the LCF, application code running on an endpoint can only invoke methods on objects on its gateway. To invoke methods on objects on other systems, the invoked gateway method in turn invokes methods on those other systems in the TMR.

Restricting method requests from an endpoint to its gateway allows gateways to share in the support requirements with the TMR server. This spreads the load across systems and increases scalability without affecting the performance of the TMR server, enabling highly scalable designs that can support the thousands of endpoints in an LCF enterprise. Endpoints, through the gateway, have access to the full TME 10 environment, so there is no restriction on the objects on which they invoke methods.

Because of these restrictions, methods are implemented differently in the LCF. In the full framework, all methods are implemented the same way, but in the LCF, calls differ according to whether they are upcalls (from the endpoint or gateway up) or downcalls (from the TMR server or gateway down). This difference introduces the following new terms and concepts:

- **upcall**: A method invocation from an endpoint or gateway “up” to the gateway.
- **downcall**: A method invocation from the TMR server or gateway “down” to an endpoint.
- **gateway method**: A method that runs on the managed node associated with the gateway servicing the endpoint referred to in the object reference. A gateway method runs as the result of an upcall.
- **endpoint method**: A method that runs on an endpoint. An endpoint method runs as the result of a downcall.
You implement gateway methods using the tools and services available in the TME 10 ADE. Endpoint methods use some of the tools and services of the TME 10 ADE, but they are implemented using a special application mini-runtime library (libmrt) specific to LCF. (For more information about libmrt, see Chapter 4, “The LCF Application Runtime Library.”)

The rest of this chapter discusses these LCF application development topics:

- LCF objects
- Endpoint methods
- Gateway methods
- Downcalls and upcalls
- Upcall architecture
- Components of an LCF application
- LCF application scenarios

**LCF Objects**

The object ID for LCF objects is formed using the usual triplet form (region.object_dispatcher.object_number) and appending +. Thus, the object ID for an LCF object has the form R.D.P+, where

- R is the TMR number
- D is the number of the object dispatcher
- P is the object number of the prototype object of the class the method is defined for
- + indicates an LCF object reference

The object dispatcher number uniquely identifies an endpoint. The TMR server maintains a mapping of endpoint dispatcher numbers to gateway objects so it can route a request to the appropriate gateway.

LCF objects are not created as instances in the object system. Instead, they are “born” when an appropriate prototype object and an appropriate endpoint dispatcher number are combined in this form.
There is no need to create an instance of each object as you do in the full framework.

LCF objects do not have a unique per-object state maintained for them by the object system (that is, per-object attributes). This means there is no need to notify the object system when a new endpoint object is created (or when the object is deleted).

To call a new object based on an existing prototype object, you generate a reference to it in the form above and then invoke it. The system locates the appropriate shared state (defined by the prototype object) and services the method request at the appropriate location.

Because there is no per-object data associated with an object by the system, applications have to assume responsibility for maintaining their own persistent store outside the context of the LCF base services.

**Endpoint Methods**

A subset of TME 10 operations, an endpoint method is one that runs directly on an endpoint. An endpoint method is implemented using the special application mini-runtime library, **libmrt**.

When an endpoint method is invoked on an endpoint, the endpoint uses the method executable stored in its method cache. If the method is not in the cache or is not the most current version, the gateway downloads the correct method to the endpoint, and it is added to the endpoint’s cache for future use.

Endpoint methods are based on TME 10 prototype objects and support many features of that model, including IDL bindings and **setid** methods. But endpoint methods differ from full framework methods in these ways:

- The LCF, unlike the full framework, is single-threaded; only per-method methods are supported for endpoint methods. That is, each method you run on an endpoint starts at invocation and completes.

- There is no transaction support for endpoints in the LCF environment. Gateways do not provide transaction management services to endpoint methods.
You need to specify dependencies for endpoint and gateway methods.

An endpoint method may be implemented across several files. Besides the executable binary, a method may require supporting files, such as a shared library or a message catalog, for example. These supporting files are called dependencies.

In the full framework, only the method body or implementation, the binary program or script that contains the method entry point, is stored in the method header. Any supporting files that the method requires are assumed to be present because all binaries, libraries, message catalogs, and so on are installed on every managed node and TMR server.

However, LCF endpoints do not have any methods or supporting files present on them when endpoints are initially installed. Method bodies are identified in the standard method header and are downloaded when needed. Because the supporting files (or dependencies) must also be downloaded when missing, the dependencies must be called out in the endpoint method so they can be present when needed.

For more information about endpoint method dependencies and how to specify them in LCF applications, see Chapter 6, “Dependencies.”

**Gateway Methods**

A gateway method runs on the gateway servicing the endpoint, the managed node where the gateway resides. It runs as the result of either an upcall request from an endpoint or a request from another managed node. After the gateway method executes, the results are passed back to the endpoint or to the managed node that made the call.

You implement gateway methods using the tools and services available in the TME 10 ADE.

**Downcalls and Upcalls**

This section discusses the sequence that occurs when your application makes an downcall or upcall.
Downcalls

A method request that originates on a managed node (or higher) and executes on an endpoint is termed a downcall. This occurs when the gateway (on the managed node) invokes a method on an endpoint. In a downcall, the gateway routes a stub call to the endpoint, and an endpoint method runs on the endpoint as a result. The downcall originates from an object call made from any managed node in the TMR or from the TMR server. The plus sign (+) in the object reference indicates that the object is an endpoint object, so the object call is interpreted as a downcall.

The endpoint receives the name of the program to execute, its arguments, and runs the program. After the endpoint method completes, the endpoint returns the results back through the gateway. The gateway formats the results back to the caller.

If the requested method does not already exist on the endpoint or does exist but is out of date (relative to the gateway), the method executable is downloaded to the endpoint. If the method has dependencies, such as libraries, they are also downloaded.

This brief sequence summarizes a downcall from a managed node to an endpoint:

- The + in the object reference distinguishes that the object runs on an endpoint.
- The TMR server maps the dispatcher in the object to a gateway object ID. The gateway object ID is used to determine which managed node the gateway is running on.
- The object dispatcher sends the request to the gateway servicing the endpoint.
- The gateway resolves the method on the corresponding behavior object and performs authorization of the invocation by the invoking TME principal.
- The method runs on the endpoint: the parameters are sent, the method runs, the MDist data is transmitted from the gateway if needed, and the results are passed back to the gateway. The gateway then returns the results to the caller.
Upcalls

Upcalls occur when endpoint applications initiate TME 10 operations in the LCF environment. This section describes the sequence that occurs when an application makes an upcall, and the following section describes the LCF upcall architecture.

General Information about Upcalls

Upcalls are method requests that originate at the endpoint. The endpoint can only invoke methods on the managed node associated with its gateway, not on any arbitrary object in the TMR. This design maintains scalability: upcalls are handled at the gateway without going to the TMR server each time, because information obtained from the TMR server is cached at the gateway. So TME 10 application code running on an endpoint is restricted to gateway methods on LCF objects. The gateway can authorize and resolve the method invocation so the TMR server does not have to be involved.

The upcall consists of a class name, the name of the method to be run, and the arguments for the method. The gateway then resolves the prototype object for the class name and constructs the object call to invoke the method.

The Sequence of an Upcall

Not all endpoint applications need to make upcalls. However, if an endpoint application does need to, the endpoint makes the upcall by calling an IDL compiler-generated stub. Unlike a method invocation in the full framework, an upcall always invokes a gateway method on the object dispatcher associated with the gateway. An application that supports upcalls must provide both the endpoint code and the gateway upcall method.

The gateway attempts to resolve the method and do the necessary authorization without making a call to the object dispatcher on the TMR server. The method header can be retrieved from the method header cache, as it is for downcalls. The access control list (ACL) check will use credentials that are sent up by the endpoint as a result of a principal login request. The object groups for an LCF object are determined on a per-endpoint basis; that is, the endpoint is placed in a
security group, and all objects on that endpoint are members of the same security group.

The gateway tells the object dispatcher to start a gateway method; the object dispatcher starts the daemon for the method; and the daemon then proceeds. Results are returned through the gateway daemon back to the endpoint.

The figure below illustrates the flow-of-control when an endpoint does an upcall, invoking a gateway method:

- The endpoint sends the request to the gateway (1).
- If the gateway can resolve the method header, it tells the object dispatcher on the managed node to invoke the method (2). If it cannot resolve the method header, it goes to the TMR server to get the header (3) and, when the header is returned to the gateway, the gateway tells the managed node to invoke the method (4).
- The method is invoked on the gateway (5).
- Results are passed back to the gateway (6, 7).
The gateway passes the results back to the endpoint (8).

Each time your application makes an upcall, it contacts lcfd, which encrypts the upcall data and sends it to the gateway. To contact lcfd, your application must determine lcfd’s communication settings.

Your application gets the port for lcfd from either the current directory or from the environment variable LCF_CFG_FILE. The file last.cfg contains the port lcfd uses. You either run the upcall from the directory with the file last.cfg (which contains the port number) or, if you are not in the directory that contains last.cfg, you set the environment variable LCF_CFG_FILE to the fully qualified pathname to last.cfg (for example, /tme10/lcf_app/last.cfg).

Upcall Architecture

If your application does upcalls, and particularly if it does upcalls from a number of endpoints, you need some way to collect this information.
For example, if many upcalls are occurring, it is a good idea, for performance reasons, to gather the upcall information at one point before processing or forwarding it. This prevents a situation where your application is making many separate attempts to process or forward information received from endpoints.

This section describes the upcall architecture and presents design considerations for applications that will make upcalls. It also provides information about how to implement upcalls in your application. The information in this section provides implementation detail for applications and supplements the general discussion of upcalls presented in earlier sections.

**The Upcall Collector**

LCF applications that make upcalls will need to have an *upcall collector* component, which is a daemon that implements all gateway methods for an application. The upcall collector is a store-and-forward router for application-specific upcalls. It collects your application’s upcall data, batches it together, and forwards it to the server component of your application (either to the TMR server or to an application Mid-level Manager (MLM)). The upcall collector should also attempt to compress data, if possible.

The application MLM then processes the data; depending on the application, it may possibly forward it to a Top-level Manager (TLM) for the application. Not all applications will need to have both a MLM and TLM; it depends on the application. For this discussion, the important concept is that an application may have a MLM to process and forward upcall data.

**Upcall Collector: An Abstract Object**

The upcall collector is implemented by an *abstract* object.

An abstract class provides a base class from which you can derive other classes. The abstract object is not instantiable; that is, it has no *create_instance* method to create an instance of the class, but it can be used for inheritance purposes. Abstract classes frequently describe the characteristics of other classes, and instantiable classes describe the characteristics of individual objects. To summarize, an abstract object
Upcall Architecture

has no state or data, exists only for the life of the transaction, and is never instantiated.

There is exactly one abstract object for each abstract class on each object dispatcher. An abstract object reference for a particular class is formed in the following way:

- Use the object reference for the prototype instance for the class
- Replace the dispatcher “1” in the object reference with the target oserv’s dispatcher
- Add a “-” to the end of the reference

For example, if 1234567890.1.555 is the object reference for a class’s prototype instance object, then 1234567890.16.555- is the abstract object reference for that class on dispatcher 16.

Application developers typically will not explicitly invoke methods on abstract objects. Most method invocations on abstract objects are implicit invocations of upcall collector methods.

Because abstract objects are not associated with an instance, they do not have any persistent store (that is, database attributes). The object dispatcher enforces this by using a different remote procedure call (RPC) server for abstract objects. The RPC server for abstract objects is modified to reject requests to add or modify attributes.

Persistent Store

Because the upcall collector is implemented by an abstract object, it has no persistent store associated with it. It is used solely for collecting and routing data from upcalls and does no processing of the data.

The upcall collector may use the configuration state to determine when to route data; for example, it may queue a certain number of records before routing them, or it may route all collected records after a certain amount of time. This type of configuration involves no processing of the data.

The upcall collector facilitates scalability. In very large enterprises, it provides an efficient way to route extremely large amounts of data to the application’s server component, whether that is on the TMR server or the MLM on a managed node.
**IDL Definition**

The upcall collector component is defined in a similar way to the endpoint component:

- It has its own IDL interface.
- Upcall collectors will never be instantiated, so they are defined to be abstract implementations. That is, the TEIDL keyword `abstract` appears in the implementation header in the `.imp` file.
- Dependencies for upcall collector methods are specified using the `wdepset` and `wchdep` commands.

**Upcall Authorization**

In the full framework, TME 10 uses the operating system user authentication and then maps operating system users to TME 10 principals. However, LCF endpoints may vary in their native functionality: they may not have a multiuser operating system, they may not be secure, and so on.

To make authentication easier for the application, LCF endpoints have the inherent capability to invoke methods that have `ANY_ACL` in their ACL. This is useful for applications that perform asynchronous upcalls; they can provide benign information to the application. (Examples of this type of application are TME 10 Distributed Monitoring and TME 10 Inventory.) Applications that choose to take advantage of this feature must carefully design their security:

- Methods called by LCF endpoints must have `ANY_ACL` in their ACL list, or endpoint upcall methods must perform a principal login.
- Methods called by LCF endpoints must have high enough roles to invoke methods as needed.

Application designers should ensure that cascaded object calls performed by upcall collectors are not security risks, since they can be indirectly invoked by anyone.
Upcall Architecture

Upcalls and TME Principals

If an LCF upcall must have real principal credentials behind it (for example, TME 10 UserLink), there are two principal authentication scenarios available to application developers. In both cases, a user name and password are submitted, and they must be a valid operating system user name and password on the TMR server.

Single Upcall Authorization

Short-lived methods that need to perform only one upcall can use per-upcall authorization. In this case, the upcall and principal login are combined into a single step to authorize a single upcall.

In your application, use the set_current_credentials function to set the user name and password. The user name and password are authenticated and mapped to a TME principal, and the upcall is authenticated based on the resulting capabilities.

Principal Login

Longer-running endpoint methods that require multiple upcalls use a principal login. The principal login returns credentials (using get_current_credentials) that may be used to authenticate subsequent upcalls. The gateway authenticates the user name and password, then returns a “sealed certificate.” The application should not store the certificate on disk or other persistent store where it may be used by others.

Data Encryption

Application data is not encrypted in the protocol. Your application is responsible for encrypting its own data. Use encrypt_data and decrypt_data to encrypt and decrypt your application’s data.

Mid-level Manager

To process data collected by an upcall, an application can either:

- Forward the data to the application’s server component, which typically resides on the TMR server.
Components of an LCF Application

- Forward the data to a Mid-level Manager (MLM), which can reside on a managed node.

An MLM is the server component of an application; it resides on a managed node and processes the upcall data received from the upcall collector. An MLM may do all of the information processing, or it may do some processing and then forward data to a Top-level Manager (TLM). An application may have one, many, or no MLMs. An application can have one TLM.

An MLM enables you to off-load the server component of an application from the TMR server to a managed node, which more evenly spreads the computing load across the enterprise. This is especially important if your TMR server is running several applications that are each handling data received from tens of thousands of endpoints. By distributing the application MLMs across managed nodes, you can create a model where data is collected and processed on managed nodes throughout the TMR, rather than concentrating all processing on the TMR server.

The figure on page 2-16 illustrates an MLM in relation to a TMR server.

Components of an LCF Application

This section discusses considerations to keep in mind when you design applications for the LCF environment and describes the components of an LCF application.

Design Considerations

When designing your LCF application, keep in mind:

- There is no database on endpoints.

- You must consider the functionality needed on both the client and server side of your application.

- The scalability of applications in the LCF. If an operation occurs on one endpoint, it will probably occur on hundreds or thousands of endpoints at the same time. You should design your application to use the scalability features that are part of LCF. For example, for upcalls, your application should use an upcall.
Components of an LCF Application

collector to collect and forward data. For downcalls, your application should use the MDist repeater capability the gateway provides.

The fact that endpoints are dataless affects application design. Except for the LCF daemon (lcfd) itself, there is no framework application database or other application-specific state kept on the endpoint. This is important for the scalability and maintenance of the LCF, and it also affects your application design.

LCF Application Components

Endpoint applications that need to perform TME operations must be written with both a client (endpoint) and a server component. Depending on the application, you may also need to write an optional gateway component.

In addition, depending on your application, it may also contain other components. The following lists the components you may need to create for your LCF application:

- Server component—Required; runs on the TMR server.
- Gateway component—The gateway component normally consists of an upcall collector. Required if an application makes upcalls; if it does not, this component is optional. The gateway component exists for scalability: it receives data from multiple endpoints and consolidates and forwards it.
- Endpoint (or client) component—Required; runs on the endpoint.
- Mid-level manager—Optional; if your application makes upcalls, you can use this component to process upcall data after it is received by the upcall collector.
Components of an LCF Application

The following figure illustrates these components in relation to each other.
LCF Application Scenarios

This section presents three different scenarios to illustrate the endpoint, gateway, and server components you may need to provide in your application.

In each of these scenarios:

- The client component runs on the endpoint. It can invoke operations on the gateway component, which runs as a regular TME method on the managed node associated with the gateway.
- The server component runs on any TMR server. The server component can invoke any other TME method for which it has sufficient privilege.
- If an endpoint method makes an upcall, there must be a gateway component; if an endpoint method does not make an upcall, there is no need for a gateway component. For a downcall, for example, there is no need to write a gateway component because of the MDist repeater capability that is inherent in the gateway.

The TME 10 Software Distribution Push Interface: a Downcall Example

To implement the TME 10 Software Distribution push interface in the LCF environment, for example, the application developer writes two pieces of code:

- Server-resident code that provides the files to the endpoint
- Endpoint-resident code that receives, unpackages, and installs the files

This example illustrates a downcall. The files are distributed through the gateway, but the gateway is uninvolved in the push. Because of this, there is no need to write a gateway component for the application. In the TME 10 Software Distribution application, this sequence would occur:

- The server initiates a file package distribution to an endpoint.
- The file package is distributed through the gateway, with no programmer involvement.
The method is downloaded automatically, if needed.
The endpoint unpackages and installs the file package.

The TME 10 UserLink Pull Interface: An Upcall Example

To implement the TME 10 Software Distribution pull interface in the LCF environment, for example, the application developer writes three pieces of code:

- Endpoint-resident code that initiates a file pull
- Gateway-resident code that handles the pull request from the endpoint
- Server-resident code that provides the files to the endpoint

In the TME 10 Software Distribution application, this sequence would occur:

- The endpoint initiates a file package distribution.
- The server receives and responds to the endpoint request.
- The file package is distributed through the gateway.
- The method is downloaded automatically, if needed.
- The endpoint unpackages and installs the file package.

TME 10 Inventory Reporting: An Upcall Collector Example

To illustrate application design that includes an upcall collector in the TME 10 Inventory environment, assume an application designed to collect inventory information from a machine each time it reboots. As part of this, an initialization method has been run on each endpoint to edit a script to run the inventory program on boot.

This example, involving an upcall to the gateway, illustrates the gateway using an upcall collector. The upcall collector gathers data from a number of endpoints and then forwards it to the database on the TMR server. This avoids the situation where possibly thousands of individual endpoints are trying to write data to the database on the TMR server.
To implement this in the LCF environment, the application developer writes three pieces of code:

- Endpoint-resident code that runs the inventory report
- Gateway-resident code that gathers the data from multiple endpoints
- Upcall collector
- Server-resident code that processes the inventory reports

When the next boot occurs, the application does the following:

- On the endpoint, runs a local inventory report to produce a data file of inventory results.
- Invokes the “inventory report” method on the gateway and sends the inventory report to the endpoint.
- The gateway waits until it has responses from many endpoints.
- The gateway invokes the TMR server component of “Inventory Report” and sends the collected inventory reports to the central inventory database object on the TMR server.
The LCF Programming Environment

The programming environment for writing endpoint methods is similar to the environment for writing methods for the full framework. Methods that run on the server or gateway are implemented using the current TME 10 ADE. Methods that run on endpoints are also written following the standard practices documented in the TME 10 ADE document set, but they are implemented using a special lightweight application runtime library (libmrt) specific to LCF.

However, the environment present in the full framework is not present in the LCF environment. The LCF ADE is a simple but functional subset of the full framework ADE.

ADE Tools

Programmers use the standard Tivoli Extended IDL (TEIDL) compiler to produce method stubs and skeletons. The stubs are used by TME methods to invoke endpoint methods. The skeletons are linked with endpoint methods and the LCF application runtime library to create executables. (For more information about TEIDL and the TEIDL compiler, see the Framework Services Manual.)

IDL Compiler-Generated Files

LCF methods are described using the standard four extended IDL files:
Linking Endpoint Methods

- **.idl**—Interface file. Specifies the IDL for the interface.
- **.imp**—Implementation file. Specifies how to implement the interface.
- **.prog**—Program file. Specifies execution characteristics for methods in a class.
- **.ist**—Installation file. Specifies how the class behaves when installed in a TME.

These four files are used to build the methods on the gateway and above on the server side and to build the endpoint-side methods.

**Compiling IDL**

To compile server-side and gateway methods, use the standard TEIDL compiler, which you invoke with the command `tidlc`. For more information about `tidlc`, see the Framework Services Manual or the manual page for `tidlc`.

To compile endpoint-side methods, use the special front-end script `ltid` to run the TEIDL compiler. `ltid` is a wrapper around `tidlc`. It has no command line arguments of its own; it passes all arguments you furnish to `tidlc` without modification. `ltid` runs `tidlc` and then modifies the standard TEIDL compiler-generated code to enable it to run on an endpoint.

For more information on `ltid`, see Chapter 9, “LCF Sample Application.”

**Generating Types**

If you are creating libraries, you must use `adrgen` to generate types on the endpoint side. See “ADR Marshalling Functions” on page 4-22 for more information about `adrgen`.

For information about how to structure and build an LCF application, see Chapter 9, “LCF Sample Application.”

**Linking Endpoint Methods**

Endpoint methods are linked with the following libraries:
Debugging

- **libmrt**—The application mini-runtime library for LCF
- **libcpl**—The common porting layer library
- **libdes**—The DES library

In addition, endpoint methods must be linked to any other libraries needed by the specific methods. For example:

- To use CCMS for LCF applications, you must link with the library **libccms_lcf**.
- To use the Task library, you must link with **libtask_lcf**. the Task library interface.
- For functions to assist in invoking processes, you must link with **libsubproc**.

The libraries are found at `$BINDIR/./lcf_bundle/lib/$INTERP`.

## Debugging

There are two ways to debug endpoint methods in the LCF environment:

- Use the ADE debugging tools from the full framework, such as the `wdebug` command. With these tools, you can debug one method at a time.
- Use the `-D debug_flags=1` option to `lcfd`. With this option, you can stop each endpoint method when it starts and attach a debugger. In this way, you can debug all methods on an endpoint.

The `wdebug` command does two things: it resolves the method to find the behavior object it lives on, and it sets a breakpoint so that the next time the method runs, it will be suspended so you can attach a debugger. For more information about the `wdebug` command, refer to the manual page.

To debug all methods, enter the following command to start `lcfd`:

```
lcfd -D debug_flags=1
```

When each method starts, `lcfd` prints to the console the method name and the process it starts. It then suspends the method so you can attach a debugger, such as `gdb`. 
The LogQ function (in libmrt) is also useful for debugging methods. It sends messages to the console or log file and is useful for times when you want to debug by means other than setting a breakpoint.
The LCF Application Runtime Library

Introduction

The LCF application mini-runtime library (libmrt) contains functions you use to implement endpoint methods. This set of library functions is linked with every endpoint method executable.

libmrt provides the smallest subset of library functions needed to implement an LCF method executable for an endpoint. Using this special application runtime library keeps the endpoint portion of the application as small and simple as possible but still provides functionality that is robust and complete.

libmrt contains subsets of these functions available in the full framework:

- Memory management
- Distributed exceptions
- Sequence manipulations
- File system input/output
- Logging functions
- Abstract Data Representation (ADR) marshalling functions
- Functions for launching a process
- Miscellaneous functions
In general, the functions in libmrt are part of the full framework and function identically to those in the full framework (described in the TME 10 ADE Application Services Manual, Volumes 1 and 2), unless noted. For a full description of each function, see the TME 10 ADE Application Services Manuals or the Programmer’s Reference Manuals.

Some of the functions documented in this chapter are specific to LCF and are not documented in the TME 10 ADE manual set. Where that is the case, additional information is furnished here. For example, the logging functions are specific to LCF and are not part of the full framework.

Each of the following sections describes a set of functions available in libmrt.

### Memory Management

libmrt provides a subset of memory management functions that work with either local or global memory:

- Use local memory for temporary allocations that are automatically freed when they go out of scope. This scoping is normally done with the exception Try() frame and the ml_ex versions of the local memory functions.

- Use global memory for all allocations that must persist.

With both local and global functions, an out-of-memory condition throws an exception. For more information on memory management, see Chapter 3, “Memory Management,” of the Application Services Manual, Volume 1.

With the exception of mg_cleanup, all the memory management functions in libmrt are part of the full framework. They are documented in the Application Services Manual, Volume 1 and in the Programmer’s Reference Manuals. mg_cleanup is called on your behalf, so there is no need to call it from your application.
The following table summarizes the memory management functions available in **libmrt** for endpoint methods.

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mg_malloc</td>
<td>Allocates a block of uninitialized global heap memory.</td>
</tr>
<tr>
<td>mg_free</td>
<td>Frees global memory allocated by mg_malloc, mg_calloc, mg_strdup, or mg_realloc.</td>
</tr>
<tr>
<td>mg_calloc</td>
<td>Allocates a block of global memory, initialized to zero.</td>
</tr>
<tr>
<td>mg_realloc</td>
<td>Reallocates global memory; changes the size of an allocated block of memory.</td>
</tr>
<tr>
<td>mg_strdup</td>
<td>Copies a string into a new block of memory allocated with mg_malloc.</td>
</tr>
<tr>
<td>mg_cleanup</td>
<td>Frees all globally allocated memory (mg_*alloc) that has not been deallocated.</td>
</tr>
<tr>
<td>ml_create</td>
<td>Creates a new local memory heap.</td>
</tr>
<tr>
<td>ml_malloc</td>
<td>Allocates uninitialized local memory.</td>
</tr>
<tr>
<td>ml_free</td>
<td>Frees local memory allocated with the ml_*alloc functions or with mg_strdup.</td>
</tr>
<tr>
<td>ml_calloc</td>
<td>Allocates local memory, initialized to zero.</td>
</tr>
<tr>
<td>ml_realloc</td>
<td>Reallocates local memory.</td>
</tr>
<tr>
<td>ml_to_mg</td>
<td>Moves memory from local to global.</td>
</tr>
<tr>
<td>ml_strdup</td>
<td>Duplicates string using ml_malloc.</td>
</tr>
<tr>
<td>ml_destroy</td>
<td>Frees all memory in a local memory heap.</td>
</tr>
<tr>
<td>ml_ex_malloc</td>
<td>Allocates uninitialized local memory, in a Try() frame.</td>
</tr>
<tr>
<td>ml_ex_calloc</td>
<td>Allocates local memory, initialized to zero, in a Try() frame.</td>
</tr>
<tr>
<td>ml_ex_realloc</td>
<td>Reallocates local memory, in a Try() frame.</td>
</tr>
<tr>
<td>ml_ex_strdup</td>
<td>Duplicates string using ml_ex_malloc, in a Try() frame.</td>
</tr>
</tbody>
</table>
Distributed Exceptions

Exceptions are used to report fatal errors. You can use the standard Try/Catch frame macros or the variable argument exception functions to handle exceptions.

Try/Catch Frame Macros

Try/Catch macros are documented in Chapter 2, “Exceptions,” of the Application Services Manual, Volume I. The following table describes the macros you can use.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Try</td>
<td>Starts a new Try/Catch frame.</td>
</tr>
<tr>
<td>Catch</td>
<td>Catches an exception of the given type.</td>
</tr>
<tr>
<td>CatchAll</td>
<td>Catches exceptions of any type.</td>
</tr>
<tr>
<td>EndTry</td>
<td>Ends a Try/Catch frame.</td>
</tr>
<tr>
<td>Throw</td>
<td>Throws an exception.</td>
</tr>
<tr>
<td>ReThrow</td>
<td>Rethrows a caught exception.</td>
</tr>
<tr>
<td>ev_to_exception</td>
<td>Converts an environment to an exception.</td>
</tr>
<tr>
<td>exception_to_ev</td>
<td>Converts an exception to an environment.</td>
</tr>
</tbody>
</table>

Variable Argument Exceptions

In many cases, it is easier to use the variable argument (printf-style) exception functions than the Try/Catch macros.

The following table summarizes the variable argument exception functions you can use.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vaThrow</td>
<td>Throws an error message.</td>
</tr>
<tr>
<td>vaThrowDerived</td>
<td>Throws a type of error message.</td>
</tr>
<tr>
<td>vaMakeException</td>
<td>Returns a pointer to the exception.</td>
</tr>
</tbody>
</table>
Using vaThrow and vaAddMsg with Message Catalogs

On LCF endpoints, you can only bind the default message. To bind other messages, you must go back to the server side and bind the message there. This discussion of vaThrow, vaAddMsg, and message catalogs applies to the server, but is included here to describe the minimum needed to send a message back up to the server for binding.

vaThrow and vaAddMsg are two printf-style functions that make working with simple message catalogs easier. This section describes how to use them when working with message catalogs. These are included to be used with the MDist capability.

For more information about message catalogs, see Chapter 6, “Message Catalogs,” in the Application Services Manual, Volume 1.

Note: The functionality provided by vaAddMsg is incorporated into the logging functions (described in “Logging Functions” on page 4-10). You may find it easier to use the logging functions (LogMsg and LogQ, in particular).

The message catalog entries for vaThrow and vaAddMsg must meet these requirements:

- Only string and integer format types are used (currently they are string, decimal, unsigned decimal, hexadecimal, and octal).
- The default message references each argument exactly once and in order.
- Specific exception classes are not required. If you want exception classes, use vaThrowDerived.

Using vaThrow and vaAddMsg to work with message catalogs has several advantages:

- The code is more readable because it does not require searching a catalog to see the text of a message.

---

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vaAddMsg</td>
<td>Appends a new message to an X/Open message.</td>
</tr>
<tr>
<td>ThrowExErrorMsg</td>
<td>Throws a message as an exception.</td>
</tr>
</tbody>
</table>
Distributed Exceptions

- It is easy to create new messages; this encourages the use of specific, informative messages rather than the reuse of less helpful messages.

The following sample code uses `vaThrow` to throw an exception:

```c
vaThrow(CatSpec(file, BadLim),
   "Invalid limit setting (%$1d). Must be in range [%$2d, %$3d]",
   curval,
   minval,
   maxval);
```

The following sample code uses `vaAddMsg` to create a compound message:

```c
msg = vaAddMsg(NULL, CatSpec(file, NoSpace1),
   "No space left on device ‘%1$s’. Need %2$d bytes",
   devname,
   resid);
msg = vaAddMsg(msg, CatSpec(file, BadCode),
   "%$1s: Unexpected pipeline code 0x%2$x",
   pipeline,
   code);
```

Message Catalogs

The `vaAddMsg` and `vaThrow` functions require no changes to the existing message catalog system. They are an additional layer on top of the existing system.

CatSpec

All of the functions take a macro of this form as an argument:

```c
CatSpec(catalog, key)
```

where `catalog` is the name of the message catalog and `key` is the message catalog key. The key must be unique for the catalog. Again, the catalog is on the server side, not the endpoint.

During program development, the macro ignores the `catalog` and `key` values, supplying null values instead. A null catalog tells the standard message catalog functions to use the default string instead of searching an actual message catalog.
Because you do not need to produce actual message catalogs until later in the development process, it is easy to write message catalog-based code from the start. A `#define` (in the file `msg_supp.h`) tells the `CatSpec` macro to start generating true file and key values.

**Format/Default Message**

The format string specifies both the `printf` format and the default message catalog entry. Because it specifies both, there are some restrictions on format types and argument order. The format string is scanned for entries preceded by `%`. Each entry results in the next variable argument being processed. Note the sequential use of 1$, 2$, ..., between the ‘%’ and format type.

The variable argument functions take special action when they encounter a string format type with the argument `$errno$. In this case, the string is converted to the corresponding `sys_errlist[]` entry (or `errno=<n>`, if out of range).

**vaAddMsg**

The `vaAddMsg` function adds a variable argument message. Its C signature is shown below:

```c
    tmf_msg_t *vaAddMsg(msg, CatSpec(file, key), format [ , arg... ])

    tmf_msg_t *msg;
    char *format;
```

A `tmf_msg_t` is a sequence of X/Open messages. `vaAddMsg` creates a new message, and either creates a new sequence (if `msg` is NULL) or appends to an existing sequence. A new message is always created. If an error occurs (for example, a format string problem), a default error message is used.

The main user of `vaAddMsg` is expected to be code like `filepack`, which accumulates a long log of errors, warnings, and informative messages during a single method invocation. Writing this code with traditional message catalog calls would be very difficult and would produce unreadable code.
Sequence Manipulations

vaThrow

The vaThrow function calls vaAddMsg to create a new message. It then throws a generic exception with the new message as context. The exception has the single format “%7$M.” Its C signature is shown below:

```c
void vaThrow(CatSpec(file, key), format [, arg... ])
    char *format;
```

Because exceptions are just messages, their arguments are 1-based, not 7-based.

vaThrow does not work for programs that need to access the first seven phantom arguments. Use vaThrowDerived to throw a derived instance of an exception thrown by vaThrow. This enables you to handle classes of messages.

Sequence Manipulations

IDL data types do not include variable length arrays. Instead, you must use sequences. A sequence consists of a pointer to an array of a given data type and a count of the number of elements in the array.

The LCF environment supports a limited subset of the sequence functions, and their use is slightly different from the full framework. In the full framework, the sequence APIs are all implemented as function calls in libtas. You must typecast all references of user-defined sequences from the native data type to the sequence_t type. For example, you would do this as follows:

```c
TMF_Types_StringList myseq;
    .
    .
    seq_get((sequence_t *) &myseq, index, sizeof(item));
```

In the LCF, the sequence APIs are lightweight macros defined in seq.h. In the LCF, it is no longer necessary to cast from the user-defined type to the sequence_t type. In fact, you want to avoid casting, because the LCF sequence macros rely on the sequence’s native data type to resolve the size of an element in the sequence. For example, the code
Sequence Manipulations

shown above (for the full framework) would be written as follows for
the LCF:

```c
TMF_Types_StringList myseq;
.
.
*seq_get(myseq, index);
```

Note that the sample code for the LCF does not cast to the `sequence_t`
type; the sequence’s native data type is used to resolve the size of an
element in the sequence. All the LCF sequence macros allow you to
use the sequence without having to cast values to and from `sequence_t`.

The following table shows the supported sequence macros, where
`<type>` is a typed `sequence_t`.

<table>
<thead>
<tr>
<th>Sequence Macros</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq_new(size_t size)</td>
<td>Creates a sequence of elements of the size specified in memory.</td>
</tr>
<tr>
<td>Seq_zero(&lt;type&gt; *seq)</td>
<td>Clears the sequence.</td>
</tr>
<tr>
<td>Seq_len(&lt;type&gt; *seq)</td>
<td>Returns the number of elements in the sequence specified.</td>
</tr>
<tr>
<td>Seq_get(&lt;type&gt; *seq, int index)</td>
<td>Returns a pointer to the data item in the sequence for the index specified.</td>
</tr>
<tr>
<td>Seq_add(&lt;type&gt; *seq, &lt;type&gt; item)</td>
<td>Adds a data item to the end of a sequence.</td>
</tr>
<tr>
<td>Seq_remove(&lt;type&gt; *seq, int index)</td>
<td>Removes a data item from a sequence.</td>
</tr>
<tr>
<td>Seq_free_buffer(&lt;type&gt; *seq)</td>
<td>Frees the memory allocated for the buffer portion of the sequence specified.</td>
</tr>
</tbody>
</table>

Application Development for the Lightweight Client Framework  4–9
The LCF supports a set of functions for file system input and output, which are summarized in the following table. For more information about each of these functions, see the manual pages in the appendix of this guide.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>open_ex</td>
<td>Opens a file. Throws an exception on error.</td>
</tr>
<tr>
<td>read_ex</td>
<td>Reads a file. Throws an exception on error.</td>
</tr>
<tr>
<td>write_ex</td>
<td>Writes a file. Throws an exception on error.</td>
</tr>
<tr>
<td>close_ex</td>
<td>Closes a file. Throws an exception on error.</td>
</tr>
<tr>
<td>makedir_ex</td>
<td>Makes a directory. Throws an exception on error.</td>
</tr>
<tr>
<td>make_path</td>
<td>Checks for and builds every component of the path.</td>
</tr>
<tr>
<td>does_file_exist</td>
<td>Returns true or false.</td>
</tr>
<tr>
<td>get_file_length_ex</td>
<td>Returns the number of bytes in a file.</td>
</tr>
<tr>
<td>get_open_file_length_ex</td>
<td>Returns (in bytes) the size of a file opened with open_ex.</td>
</tr>
<tr>
<td>lseek_ex</td>
<td>Moves around in a file opened with open_ex.</td>
</tr>
<tr>
<td>rename_file_ex</td>
<td>Renames a file. Throws an exception on error.</td>
</tr>
<tr>
<td>copy_file_ex</td>
<td>Copies source path to destination path and returns the number of bytes copied. Throws an exception on error.</td>
</tr>
<tr>
<td>ep_stream_read</td>
<td>Reads an MDist stream.</td>
</tr>
</tbody>
</table>

**Logging Functions**

A logging utility has been added for LCF and is part of libmrt. This section describes the logging utility, discusses its use in lcfd and libmrt and for methods, and briefly describes each of the functions it offers. This utility is for use only with endpoint methods.
Logging Functions

Overview

The logging utility provides functions that enable you to create multiple logs to produce `printf`-style messages to a console and to a file. The module is contained in `log.c` and `log.h`.

The module defines a structure to manage resources related to the log, such as file descriptor, path, and output to console. The structure is private to `log.c`; it returns a pointer (of type `lh_p_t`) to the caller during initialization. The caller passes the pointer in subsequent calls to log functions. The caller is responsible for calling the deinitialization function (`LogDeinit`) to free resources.

On UNIX platforms, the log file is opened when the log is initialized and closed when it is deinitialized. On PC platforms, the log file is opened, written to, and closed for each message.

Log functions that use file system input/output functions always catch exceptions in the log function and handle the problem without rethrowing these exceptions.

The following sample illustrates the format of the log file. It includes the date and time, the level of the message, the application generating it, and the message itself:

```
Oct 11 15:24:48 1 lcfd New connection
    from '146.84.26.55+32866'
Oct 11 15:24:48 2 lcfd recv: len='243'
    (code='14', session='91')
Oct 11 15:24:48 2 lcfd write login file complete
Oct 11 15:24:48 1 lcfd Login to gateway complete.
Oct 11 15:24:48 1 lcfd Ready. Waiting for
    requests (0.0.0.0+9495).
Oct 11 15:24:48 Q lcfd Entering '../../src/prog/lcfd.c':165
Oct 11 15:24:48 2 lcfd New connection from
    146.84.26.55+32868
Oct 11 15:24:48 1 lcfd recv: len=86: 0x5a038
    (code=0, sess=1485535081)
Oct 11 15:24:48 1 lcfd Spawning: ./cache/tst, session id:
    588b7769, ConfigFile:./cfgout.dat
Oct 11 15:24:48 Q lcfd Entering '../../src/prog/lcfd.c':165
Oct 11 15:24:49 Q tst Entering '../../src/ecp/mrt_run.c':85
Oct 11 15:24:49 Q tst argv: session_id=588b7769 cntl_file=0
Oct 11 15:24:49 Q tst Entering
    '../../src/ecp/mrt_run.c':153
Oct 11 15:24:49 Q tst waiting for input args
```
Logging Functions

Messages with a level of Q are output from the LogQ function. LogQ hardcodes its level to 2, so the more informative Q is used in the log file to distinguish those messages from other level 2 messages.

Using the Log File in lcfd and libmrt

The LCF logging utility creates a default log for lcfd and libmrt output. The name of the default log is ./lcfd.log. You can specify a different name for the log file by means of the configuration file and command line. You can also create additional logs, but all output from libmrt goes to the default log.

The default log is created by calling LogInit, then calling LogSetDefault to establish the log as the default. Calling LogSetDefault allows you to use LOGDEF as the log handle without having to make the lh_p_t returned from LogInit a global.

To create additional logs, call LogInit, and use the returned lh_p_t in the desired log module functions.

The default display_threshold for a log is 0, or no output. The following valid values for display_threshold can be set by the user in the configuration file or on the command line:

<table>
<thead>
<tr>
<th>Value</th>
<th>Log Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No logging</td>
</tr>
<tr>
<td>1</td>
<td>Minimal output</td>
</tr>
<tr>
<td>2</td>
<td>Tracing and moderate output</td>
</tr>
<tr>
<td>3</td>
<td>Data buffers and tight loops</td>
</tr>
</tbody>
</table>

To set the display_threshold on the command line, use the -d option of the lcfd command, and supply a value from 0 to 3.

Using the Log File in Methods

Messages from libmrt calls go to the default log. When a method is spawned from lcfd, a log resource is initialized for the method, and its output is appended to lcfd’s log. Use LOGDEF as the log handle for
the logging functions. For additional logs, call LogInit, and use the lh_p_t it returns in the desired log module functions.

Upcall logging is independent of lcfd logging. An upcall will get libmrt log messages only if it initializes the log and sets the appropriate log level.

Summary of Logging Functions

The logging utility includes the following functions. Each is discussed in the sections following the list.

- LogInit
- LogDeinit
- LogMsg
- LogSetDefault
- LogGetDefault
- LogSetThreshold
- LogGetThreshold
- LogSetOutputStdout
- LogGetOutputStdout
- LogSetAppName
- LogGetAppName
- LogQ
- LogQueueAlloc
- LogQueueDealloc
- LogQueueGetSize
- LogQGetBuffer
- LogData

LogInit

The LogInit function creates a new log file. It also backs up the old log file and allocates resources needed by the log module. Each time LogInit is called, the resources for a log file are allocated. The caller
Logging Functions

is responsible for calling the `LogDeinit` function to deallocate resources. The function’s C signature is:

```c
lh_p_t LogInit(char *path,
               char *appname,
               int display_threshold,
               bool_t output_to_stdout
               long max_size)
```

The parameters are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>path</code></td>
<td>A null-terminated character array containing the absolute path for the log file and output messages.</td>
</tr>
<tr>
<td><code>appname</code></td>
<td>A null-terminated character array containing the name of the calling application.</td>
</tr>
<tr>
<td><code>display_threshold</code></td>
<td>An integer that defines the ceiling of display level to output. Any messages with a display level greater than <code>display_threshold</code> will not be output.</td>
</tr>
<tr>
<td><code>output_to_stdout</code></td>
<td>A boolean that allows messages to be displayed on the user’s console.</td>
</tr>
<tr>
<td><code>max_size</code></td>
<td>The maximum size of the log file.</td>
</tr>
</tbody>
</table>

Upon successful completion, `LogInit` returns `lh_p_t`, which is a void pointer to the log structure. NULL is returned on error. However, NULL (LOGDEF) is a valid value to be passed to the other log modules.

**LogDeinit**

The `LogDeinit` function deallocates resources set by a call to `LogInit`. The function’s C signature is:

```c
void LogDeinit(lh_p_t logHndl)
```

`logHndl` is a pointer (of type `lh_p_t`) to the log structure. There are no output parameters and no return value.
LogMsg

The LogMsg function uses the Tivoli National Language Support (NLS) to format an internationalized message and then outputs it to console and log file. The function’s C signature is as follows:

```c
void LogMsg(int display_level,
            lh_p_t logHndl,
            char *catalog,
            long key,
            char *msg,
            ...)
```

The parameters are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>display_level</td>
<td>An integer indicating the level for an individual log message; used for comparison against the log's display_threshold.</td>
</tr>
<tr>
<td>logHndl</td>
<td>A pointer (of type lh_p_t) to the log structure.</td>
</tr>
<tr>
<td>catalog, key</td>
<td>Output from the CatSpec, genKey and nullKey macros identifying the NLS internationalized message.</td>
</tr>
<tr>
<td>msg</td>
<td>A null-terminated character array containing the format of the message.</td>
</tr>
<tr>
<td>...</td>
<td>The input arguments to the format string.</td>
</tr>
</tbody>
</table>

The message catalog entries must meet these requirements:

- Only string and integer format types are used (currently string, decimal, unsigned decimal, hexadecimal, and octal).
- The default message references each argument exactly once and in order.
- Specific exception classes are not required.

The following example illustrates the use of LogMsg:

```c
LogMsg(1, LOGDEF, genKey(UsingMethCacheDir), "Using method cache directory '%1$s/%2$s'", gcs->cache_loc, gcs->current_interp);
```
In this example:

- LOGDEF is the log initialized for you at startup.
- genKey(key) is a short-hand macro around `CatSpec`. It incorporates the catalog name. The key must be unique for the catalog.
- During program development, the macro ignores the catalog and key values, supplying null values instead. A null catalog tells the standard message catalog functions to use the default string rather than search an actual message catalog. A `#define` (HAVE_CATALOG in the `make` file) tells the `CatSpec` macro to start generating true file and key values.
- The format string plays the dual role of `printf` format and default message catalog entry; there are restrictions on format types and argument order. The format string is scanned for `%` entries. Each entry results in the next variable argument being processed. Note the sequential use of 1$, 2$, ... between the `%` and format type.

The LogMsg functions take special action when they encounter a string format type with the argument `$errno`. In this case, the string is converted to the corresponding `sys_errlist[]` entry (or `errno=<n>`, if out of range).

LogMsg has no output parameters and no return value.

**LogSetDefault**

The LogSetDefault function maintains a static (private) pointer to the default log structure. Once set, all log functions using LOGDEF as the log handle will use this log. The function’s C signature is as follows:

```c
lh_p_t LogSetDefault(lh_p_t lh)
```
The parameters are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lh</td>
<td>A pointer (of type lh_p_t) to the log structure.</td>
</tr>
</tbody>
</table>

Upon successful completion, LogSetDefault returns lh_p_t, which is a pointer to the previous default log value.

**LogGetDefault**

The LogGetDefault function returns a pointer to the default log structure. The function’s C signature is as follows:

```
lh_p_t LogGetDefault(void)
```

Upon successful completion, LogGetDefault returns lh_p_t, which is a pointer to the default log value.

**LogSetThreshold**

The LogSetThreshold function sets the output threshold of the display (debug) level of the requested log. It is usually called by LogInit, but it may be called directly by the user to dynamically override the initial value of display_threshold. It returns the value of the previous threshold.

The function’s C signature is as follows:

```
int LogSetThreshold(lh_p_t logHndl, int new_display_threshold)
```

The input parameters are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>logHndl</td>
<td>A pointer (of type lh_p_t) to the log structure.</td>
</tr>
<tr>
<td>new_display_threshold</td>
<td>An integer that defines the ceiling of display level to output. Any messages with a display (debug) level greater than the display threshold will not be output.</td>
</tr>
</tbody>
</table>
The function returns an integer that was the previous \textit{display\_threshold} of the requested log.

\textbf{LogGetThreshold}

The \texttt{LogGetThreshold} function returns the value of the \textit{display\_threshold} for that log. The function’s C signature is as follows:

\begin{verbatim}
int LogGetThreshold(lh_p_t logHndl)
\end{verbatim}

The parameters are as follows:

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Parameter & Description \\
\hline
\textit{logHndl} & A pointer (of type \texttt{lh\_p\_t}) to the log structure. \\
\hline
\end{tabular}
\end{table}

The function returns an integer that is the current \textit{display\_threshold} of the requested log.

\textbf{LogSetOutputStdout}

The \texttt{LogSetOutputStdout} function sets the boolean to output messages to stdout. It allows the user to set the value at times other than initialization. The function’s C signature is as follows:

\begin{verbatim}
bool_t LogSetOutputStdout(lh_p_t logHndl, 
                         bool_t new_output_stdout)
\end{verbatim}

The parameters are as follows:

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Parameter & Description \\
\hline
\textit{logHndl} & A pointer (of type \texttt{lh\_p\_t}) to the log structure. \\
\textit{new\_output\_stdout} & A boolean set to TRUE for output to display on stdout as well as to log file. Set to FALSE to output only to log file. \\
\hline
\end{tabular}
\end{table}

It returns the boolean that was the previous \textit{output\_stdout} of the requested log.
LogGetOutputStdout

The LogGetOutputStdout function returns the value of the output_stdout for that log. The function’s C signature is as follows:

\[
\text{bool}_t \text{ LogGetOutputStdout}(\text{lh}_p_t \text{ logHndl})
\]

The parameters are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>logHndl</td>
<td>A pointer (of type \text{lh}_p_t) to the log structure.</td>
</tr>
</tbody>
</table>

It returns a boolean that is the current output_stdout for the requested log.

LogSetAppName

The LogSetAppName function sets the identifier to be used in logging messages. It allows the user to set the value at times other than initialization. Several applications may have output going to one log file, and LogSetAppName provides an identifier to indicate which application generated the message. The function’s C signature is as follows:

\[
\text{void LogSetAppName} (\text{lh}_p_t \text{ logHndl}, \\
\text{char *appName})
\]

The parameters are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>logHndl</td>
<td>A pointer (of type \text{lh}_p_t) to the log structure.</td>
</tr>
<tr>
<td>appName</td>
<td>A null-terminated character array containing the calling application’s identifier or name.</td>
</tr>
</tbody>
</table>

There is no return value.
Logging Functions

**LogGetAppName**

The **LogGetAppName** function returns the value of the application name for the requested log. The function’s C signature is as follows:

```c
char * LogGetAppName(lh_p_t logHndl)
```

The parameters are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>logHndl</code></td>
<td>A pointer (of type <code>lh_p_t</code>) to the log structure.</td>
</tr>
</tbody>
</table>

**LogGetAppName** returns the `appName` set by **LogSetAppName** or by the parameter `appName` in **LogInit**. The return value is a null-terminated character array containing the requested log’s `appName`.

**LogQ**

The **LogQ** function is a wrapper function around **LogMsg**; it implements a circular queue in memory of the last `n` messages, to be included in exceptions. It also outputs messages to the console and/or log file by means of **LogMsg**. **LogQ** does not support NLS or multiple log files. It does not use message catalogs; it uses the `printf` format string.

Each message is forwarded to the default **LogMsg**. The storage is allocated and deallocated by the log module. The `display_level` for LogQ is hardcoded to 2. This function is useful for debugging.

The function’s C signature is as follows:

```c
void LogQ(char *msg, ...)
```

The parameters are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>msg</code></td>
<td>A null-terminated character array containing the format string of the message.</td>
</tr>
<tr>
<td>...</td>
<td>The input arguments to the format string.</td>
</tr>
</tbody>
</table>
Logging Functions

The LCF Application Runtime Library

It has no return value.

**LogQueueAlloc**

The `LogQueueAlloc` function allocates the size of the buffer `LogQ` messages. You must call `LogQueueAlloc` before any `LogQ` messages.

The function’s C signature is as follows:

```c
long LogQueueAlloc(long newBufSz)
```

The parameters are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>newBufSz</code></td>
<td>The new size of the buffer in bytes.</td>
</tr>
</tbody>
</table>

`LogQueueAlloc` returns the previous size of the `LogQ` buffer.

**LogQueueDealloc**

The `LogQueueDealloc` function shuts down and deallocates memory used for the `LogQ` buffer. The function’s C signature is as follows:

```c
void LogQueueDealloc(void)
```

It has no return value.

**LogQueueGetSize**

The `LogQueueGetSize` function queries the size of the buffer for `LogQ` messages. The function’s C signature is as follows:

```c
long LogQueueGetSize(void)
```

`LogQueueGetSize` returns the size of the `LogQ` buffer in bytes.

**LogQGetBuffer**

The `LogQGetBuffer` function returns a character array containing the circular queue of `LogQ` messages. The function’s C signature is as follows:

```c
char * LogQGetBuffer(void)
```
ADR Marshalling Functions

There are no input parameters. It returns a null-terminated character array containing the circular queue of LogQ.

**LogData**

The LogData function formats and logs binary data. It takes in binary data and passes HEX values to LogMsg. Use this function to output binary data to the log file, such as to receive or send buffers. The function’s C signature is as follows:

```c
void LogData(int display_level,
             lh_p_t *logHndl,
             char *label,
             void *dataIn,
             long length)
```

The parameters are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>display_level</td>
<td>An integer indicating the message level for comparison against the log’s display_threshold.</td>
</tr>
<tr>
<td>logHndl</td>
<td>A pointer (of type lh_p_t) to the log structure.</td>
</tr>
<tr>
<td>label</td>
<td>A null-terminated character array containing a title for the output.</td>
</tr>
<tr>
<td>dataIn</td>
<td>A void* to binary data.</td>
</tr>
<tr>
<td>length</td>
<td>A long int describing the length of dataIn.</td>
</tr>
</tbody>
</table>

It has no return value.

**ADR Marshalling Functions**

The abstract data representation (ADR) functions are identical to the full framework versions with one exception: there is no interface repository. Because there is no dynamic type lookup, all types used must be present at compile time. If you need IDL types on an endpoint, you must compile them in.
This section discusses the C data types built in to libmrt, how to use adrgen to create data types you need on endpoints, and how to use ADR types for encoding and decoding.

**C Data Types**

For argument passing, the following C data types are built in to libmrt:

- any
- boolean
- char
- double
- float
- long
- octet
- short
- string
- ulong
- ushort

All other data types must be written as CORBA IDL type definitions. Complex types are created with the `struct`, `array`, and `sequence` keywords. Use the following function to register defined types:

```c
void adr_type_init(type_repository **types);
```

You can make multiple calls to `adr_type_init` to initialize `type_repository` lists from different modules. If you use only built-in types, you must still make the following call:

```c
adr_type_init(NULL);
```

**Creating Data Types Using adrgen**

The IDL compiler produces a large number of files, even when run with simple input. Many times, if you only need the types for marshalling and unmarshalling in your application, you can follow the simpler approach of using the script `adrgen`. 
**ADR Marshalling Functions**

The `adrgen` command runs the IDL compiler, then post-processes the files to produce just two files: `module_adr.c` and `module_adr.h`.

Input to `adrgen` is a file with IDL `typedef` declarations or structures as shown in the following:

```c
typedef sequence<octet> vdata;
struct dstr {long count; string dstr};
struct atst {any a_any; vdata a_v long a_array[10];};
```

No module, interface, `#include` or other lines are needed. As an example, assume the input above is in a file called `vdata.idl`. When you run it through `adrgen`, this input produces the files `vdata_adr.c` and `vdata_adr.h`. Type codes for the individual types will have `extern` declarations in `vdata_adr.h` similar to the following:

```c
extern _TypeCode TC_dstr[];
```

One of the last lines in `vdata_adr.h` will be:

```c
extern type_repository vdata_type_repository[];
```

This `type_repository` is then passed to `adr_type_init` to initialize the type repository so that the `vdata` types are available for encoding and decoding.

```c
adr_type_init(vdata_type_repository);
```

**Using ADR Types for Encoding and Decoding**

You can use several special functions that make it easy to use the `TypeCode` interface for general purpose encoding and decoding: `tmf_encode`, `tmf_decode`, and `tmf_free_generic`.

The C signature for `tmf_encode` is as follows:

```c
int tmf_encode(TypeCode tc, void *param, char **datap, int *datalenp);
```
Functions for Launching Processes

\texttt{tmf\_encode} encodes the pointer \textit{param}, of type \textit{tc}, and returns the resulting buffer and buffer length in \textit{datap} and \textit{datalen}. The storage for \textit{datap} is allocated with \texttt{mg\_malloc} and must be freed with \texttt{mg\_free}. It returns zero upon success or an ADR error code upon failure. \texttt{tmf\_encode} throws exceptions.

The C signature for \texttt{tmf\_decode} is as follows:

\begin{verbatim}
int tmf_decode(TypeCode tc, char *data, int datalen, void *param);
\end{verbatim}

\texttt{tmf\_decode} decodes the previously encoded buffer in \textit{data}, of length \textit{datalen}, and type \textit{tc}. The resulting data structure is returned in \textit{param}. The storage for the top-level of \textit{param} must be allocated by the caller; the deep fields are allocated with \texttt{mg\_malloc} and must be freed by the caller. It returns zero upon success or an ADR error code upon failure. \texttt{tmf\_decode} throws exceptions.

The C signature for \texttt{tmf\_free\_generic} is as follows:

\begin{verbatim}
void tmf_free_generic(void *storage, TypeCode tc);
\end{verbatim}

\texttt{tmf\_free\_generic} performs a deep free of the data structure \textit{storage}, of type \textit{tc}. It returns zero upon success or an ADR error code upon failure.

\textbf{Functions for Launching Processes}

Several functions have been added to launch processes on UNIX and PC platforms. They include functions to launch a process, wait for a process, and so on. These functions simplify process launch and give a unified API across all platforms for performing these types of functions.

For more detailed information on each, see the manual pages in Appendix A, “Reference.”
Functions for Launching Processes

The following sample code illustrates how you use several of these functions:

```c
launch(char** args, char** env) {
    void *io_ptr;
    int pid, status;

    io_ptr = tiv_io_create(IO_DEFAULT, NULL, IO_DEFAULT, NULL, NULL, NULL);

    pid = tiv_spawn(NULL, args, env, io_ptr, NULL);
    if (pid != -1 && tiv_wait(pid, &status) == pid) {
        exit_status = WEXITSTATUS(status);
    } else {
        /* error */
    }
}
```

The following functions are available:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tiv_spawn</strong></td>
<td>Spawn a process.</td>
</tr>
<tr>
<td><strong>tiv_wait</strong></td>
<td>Wait for an asynchronously spawned process to return.</td>
</tr>
<tr>
<td><strong>tiv_io_create</strong></td>
<td>Creates an array to pass to <strong>tiv_spawn</strong> as the argument representing the file handles for stdin, stdout, and stderr.</td>
</tr>
<tr>
<td><strong>tiv_io_destroy</strong></td>
<td>Free a pointer allocated by a call to <strong>tiv_io_create</strong>.</td>
</tr>
<tr>
<td><strong>tiv_user_token_create</strong></td>
<td>Allocates a pointer to token that contains information used by <strong>tiv_spawn</strong> for launching processes in the context of another user.</td>
</tr>
<tr>
<td><strong>tiv_user_token_destroy</strong></td>
<td>DEALLOCATES THE TOKEN CREATED BY CALLING <strong>tiv_user_token_create</strong>.</td>
</tr>
<tr>
<td><strong>tiv_spawn_ui</strong></td>
<td>Launch a GUI-based command on a desktop other than the default desktop of the current process.</td>
</tr>
</tbody>
</table>
## Miscellaneous Functions

This table shows miscellaneous functions provided for endpoint applications. For more information on each, see the manual pages in Appendix A.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>decrypt_data</td>
<td>Decrypt application data.</td>
</tr>
<tr>
<td>encrypt_data</td>
<td>Encrypt application data.</td>
</tr>
<tr>
<td>set_current_credentials</td>
<td>Sets the user and password to obtain a principal login during an upcall method.</td>
</tr>
<tr>
<td>set_lang</td>
<td>Sets the language to use to bind messages on the gateway.</td>
</tr>
<tr>
<td>still_alive</td>
<td>Reinitializes the timer on the gateway to keep the gateway from timing out during an upcall or downcall.</td>
</tr>
<tr>
<td>ep_stream_read</td>
<td>Reads an MDist stream.</td>
</tr>
<tr>
<td>mrt_set_method_exit_mode</td>
<td>Sets an attribute to create a condition (restart or reboot) when a method exits.</td>
</tr>
<tr>
<td>mrt_machine_id</td>
<td>Returns the machine ID.</td>
</tr>
<tr>
<td>nw_echo_command_to_console</td>
<td>NetWare only; sends a command line to the console.</td>
</tr>
<tr>
<td>mrt_test_dependency</td>
<td>Determines if a dependency to a method has been updated. For use by long-running methods such as daemons.</td>
</tr>
<tr>
<td>ioch_recv</td>
<td>Receive bytes from the IO channel.</td>
</tr>
</tbody>
</table>
**set_current_credentials**

The `set_current_credentials` function sets the user and password to obtain a principal login during an upcall method, for the per-upcall principal authorization. The function’s C signature is as follows:

```c
void set_current_credentials(char *user, char *password)
```

The parameters are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>user</td>
<td>The user ID the method should run as.</td>
</tr>
<tr>
<td>password</td>
<td>The password for the user ID.</td>
</tr>
</tbody>
</table>

It has no return value.

**set_lang**

The `set_lang` function sets the language the gateway uses to bind messages. The function’s C signature is as follows:

```c
void set_lang(char *lang)
```

The parameters are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lang</td>
<td>Language to use.</td>
</tr>
</tbody>
</table>

**still_alive**

The `still_alive` function reinitializes the timer on the gateway to keep the gateway from timing out the session. When the endpoint gateway is waiting to receive data (for example, the results from a downcall to an endpoint), by default it times out after 10 minutes. `still_alive` sends a packet to the gateway to reinitialize the timer on the gateway.

The function’s C signature is as follows:

```c
void still_alive(void *ipc, unsigned char status)
```
The parameters are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{ipc}</td>
<td>IPC handle.</td>
</tr>
<tr>
<td>\textit{status}</td>
<td>Status field (for application use).</td>
</tr>
</tbody>
</table>

\textbf{ep\_stream\_read}

The \texttt{ep\_stream\_read} function reads an MDist stream and returns the number of bytes in the stream. The function’s C signature is as follows:

\begin{verbatim}
long ep_stream_read(char **datap);
\end{verbatim}

For more information, see the manual page in Appendix A, “Reference.”
The Common Porting Layer

Introduction

This chapter describes libcpl, the common porting layer (CPL) for use with endpoint applications. libcpl implements functions that either exist on some but not all platforms or that exist but behave differently on a particular platform. For example, getopt is commonly used on UNIX platforms to parse program command lines, but does not exist on Windows or NetWare platforms; hence, it is included in libcpl.

For each libcpl function, this chapter describes the functionality it provides, by what name, and how to include it in your code. It includes the following sections:

- Binary tree search functions
- cpl_correct_path
- Directory entry functions
- get functions
- ltoa function
- Microsoft Visual C++ (MSVC) stat functions
- printf functions
- Temporary file functions
- Thread yield functions
Binary Tree Search Functions

- University of California Berkeley (UCB) compatibility functions
- `uname` functions

**Binary Tree Search Functions**

Use the `cpl_tdelete`, `cpl_tfind`, `cpl_tsearch`, and `cpl_twalk` functions to manage binary tree searches. To use these in your code, specify the following:

```c
#include <cpl/config.h>
#include <cpl/search.h>

typedef enum { preorder, postorder, endorder, leaf } VISIT;

void *cpl_tdelete(const void *, void **, int (*)(const void *, const void *));
void *cpl_tfind(const void *, void *const *, int (*)(const void *, const void *));
void *cpl_tsearch(const void *, void **, int (*)(const void *, const void *));
void cpl_twalk(const void *, void (*)(const void *, VISIT, int));
```

For more information on the `tdelete`, `tfind`, `tsearch`, and `twalk` functions, refer to the section 3C (C Library Functions) UNIX manual pages.

**cpl_correct_path**

The `cpl_correct_path` function corrects the path by replacing slashes with the appropriate path delimiter for the particular operating system. You indicate the mode by specifying one or more values from the list below.

The function’s C signature is as follows:

```c
#include <cpl/config.h>
#include <cpl/dir.h>

char *cpl_correct_path(char *path, unsigned int mode);
```

The following modes are supported:

- PATH_CORRECT_SLASHES—Corrects the orientation of slashes in the path string.
**PATH_CORRECT_EXTENSION**—For PCs, use this mode to add an extension if the path does not already contain one. This corrects to `.exe` for Win95, WIN3x, DOS and OS/2. Corrections to `.com` or `.bat` are not supported. On NT and UNIX platforms, this mode does nothing.

**PATH_REMOVE_MULTI_SLASHES**—Removes multiple slashes in a path string. Some operating systems do not allow multiple slashes. It removes the beginning double slashes on uniform naming convention (UNC) paths in NT and Windows 95. (UNC names begin with `\`, indicating that the resource exists on a remote machine.)

**PATH_COMPRESS_NAMES**—For Win3x and NetWare, this mode compresses long names to names that comply with the standard file allocation table (FAT) 8.3 file naming convention. This function currently truncates the name.

---

**cpl_get_current_location**

The `cpl_get_current_location` function finds the path of the file loaded to create your process.

```c
char *cpl_get_current_location(char *pathBuff)
```

**cpl_fflush**

The `cpl_fflush` function flushes a stream. It causes any buffered data to be written; any unread data buffered in the stream is discarded. The stream remains open.

```c
int cpl_fflush(FILE *stream);
```

For more information on the `wstat` macros, refer to the section 5 (Header, Tables, and Macros) UNIX manual page for `wstat(5)`. 
Directory Entry (dirent) Functions

The **dirent** functions enable the caller to use these APIs for all platforms without having to be concerned with platform-specific implementation details. The following APIs are supported:

```c
DIR* cpl_opendir(const char *);
struct dirent* cpl_readdir(DIR *);
void cpl_rewinddir(DIR *);
void cpl_closedir(DIR *);
int cpl_telldir(DIR *);
void cpl_seekdir(DIR *, long);
```

There is CPL support for all platforms and CPL implementations for WIN32 platforms.

**get Functions**

There are a variety of UNIX system calls that are not provided on other platforms. This section describes CPL support for the UNIX **get** functions.

**cpl_getcwd**

The **cpl_getcwd** function returns the calling process’s current working directory, if it can be determined. It is analogous to the UNIX function **getcwd**. Its C signature is as follows:

```c
#include <cpl/config.h>
#include <cpl/dir.h>

cpl_getcwd(char *buf, size_t size);
```

There is CPL support for this function for all platforms and CPL implementation for NetWare.

**cpl_getenv and cpl_putenv**

NetWare’s non-writeable environment requires special **getenv** and **putenv** functions. To include these in your code, specify the following:

```c
#include <cpl/config.h>
#include <cpl/env.h>
```
**cpl_getopt**

Use the `cpl_getopt` function to extract command line switches and their arguments from the command line:

```c
#include <cpl/config.h>
#include <cpl/getopt.h>

int cpl_getopt(int argc, char *const argv[], const char *optstring);
```

**cpl_getpass**

The `cpl_getpass` function queries users for a (password) string from standard input. Characters entered by the user are not echoed. Its C signature is as follows:

```c
#include <cpl/config.h>
#include <cpl/getpass.h>

char *getpass(const char *prompt);
```

There is CPL support for WIN32 and UNIX platforms and CPL implementation for WIN32.

**cpl_gettimeofday**

Because the `gettimeofday` function returns the time zone on some platforms but not on others, `libcpl` includes the `cpl_gettimeofday` function in the common porting layer. To use the `cpl_gettimeofday` function, include the following in your code:

```c
#include <cpl/config.h>
#include <cpl/gettimeofday.h>

int cpl_gettimeofday(struct timeval *tp, void *tzp);
```

You must declare the structure `timeval` before you include `<cpl/gettimeofday.h>`. It is usually declared in `<sys/time.h>`. It is also declared in `<winsock.h>`.
cpl_gethostname

To use the **cpl_gethostname** function, include the following in your code:

```c
#include <cpl/config.h>
#include <cpl/gethostname.h>

int cpl_gethostname(char *name, int namelen);
```

Use **cpl_gethostname** instead of **gethostname** when **CPL_USE_WINSOCK** is defined. It differs from the WinSock **gethostname** function. It does not require a prior **WSAStartup()** call, and it sets **errno**.

cpl_ltoa

The **cpl_ltoa** function converts a long integer from a binary representation to a string representation. It also returns a pointer to the string. Its C signature is as follows:

```c
#include <cpl/config.h>
#include <cpl/ltoa.h>

char *cpl_ltoa(long n);
```

There is CPL support and implementation for WIN32 (Microsoft Visual C compiler runtime, also known as MSVC runtime) platforms.

MSVC stat(2) Macros

The file **msvc_stat_macros.h** defines several common macros that are useful when using the **stat** function to get information about a file. The MSVC runtime does not provide an implementation for these macros, so they are defined in CPL. The following macros are defined:

- **S_ISDIR()** Is the file a directory?
- **S_ISCHR()** Is the file a character special device?
- **S_ISFIFO()** Is the file a FIFO?
- **S_ISREG()** Is the file a regular file?
MSVC-based platforms should get these macros by including:

```
#include <cpl/config.h>
#include <cpl/msvc_stat_macros.h>
```

UNIX (and possibly other platform types) should get these macros by including:

```
#include <sys/stat.h>
```

or possibly:

```
#include <sys/mode.h>
```

There is CPL support and implementation for MSVC-based platforms.

**printf, fclose, fopen, getc**

Because `printf.h`, `fclose`, `fopen`, and `getc` do not exist on the Windows and NetWare platforms, `libcpl` furnishes wrappers for these functions.

**cpl_fprintf**

The wrapper for `printf.h` is `cpl_fprintf`:

```
#include <cpl/config.h>
#include <cpl/printf.h>
void cpl_fprintf(FILE *fd, char *format, ...);
```

**cpl_fclose**

The wrapper for `fclose` is `cpl_fclose`:

```
#include <cpl/config.h>
#include <cpl/dir.h>
cpl_fclose(FILE *stream);
```

**cpl_fopen**

The wrapper for `fopen` is `cpl_fopen`:

```
#include <cpl/config.h>
#include <cpl/dir.h>
cpl_fopen(const char *filename, const char *type);
```
printf, fclose, fopen, getc

cpl_getc

The wrapper for getc is cpl_getc:

```c
#include <cpl/config.h>
#include <cpl/dir.h>
cpl_getc(FILE *stream);
```

Callback Registration Functions

Two registration callback functions, cpl_register_print_callback() and cpl_register_thread_yield_callback(), are provided as part of the common porting layer.

The cpl_register_print_callback function registers a function for handling LCF output messages generated inside libmrt. By default, all libmrt messages are printed to stdout. However, on platforms such as Windows and NetWare, it is more useful to have an application-specific way of displaying these messages. The cpl_register_print_callback function provides this.

To use the cpl_register_print_callback function, include the following in your code:

```c
#include <cpl/printf.h>

void cpl_register_print_callback(void (*cb)(char *));
```

The cb parameter is the callback.

The following sample code illustrates how to have the myprintf() function handle all libmrt display messages, rather than sending them to stdout:

```c
1 #include <cpl/printf.h>
2
3 void myprintf( char *msg )
4 {
5    display_in_debug_window(windowHandle,msg);
6 }
7
8 int main ( int argc, char *argv[] )
9 {
10    ...
11   /* Have myprintf() handle all libmrt */
12   /* display messages */
```
printf, fclose, fopen, getc

The Common Porting Layer

```c
12    cpl_register_print_callback(myprintf);
13    ...
14 }
15
Use the `cpl_register_thread_yield_callback` function to allow an application to register a callback function that will be called periodically during long `libmrt` operations. For example, some LCF operations, such as waiting for a task to complete or for a communication event to occur, may require that a program wait inside a `libmrt` function for an extended period of time. This can be a problem in single-threaded applications that need to handle other events during the operation. For example, Windows applications need to handle OS events for moving and resizing a window.

To use the `cpl_register_thread_yield_callback` function, include the following in your code:

```c
#include <cpl/systhread.h>
void cpl_register_thread_yield_callback(void (*cb)(int));
```

The following sample code illustrates how to handle window events on Windows 3.x platforms during critical `libmrt` calls:

```c
#include <cpl/systhread.h>

void threadYield(int s)
{
    MSG msg;

    /* Throttle messages from OS */
    while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        DispatchMessage(&msg);
    }

    return;
}
```

int WINAPI
Temporary File Functions

Most runtimes provide support for `tmpfile()` and `tmpnam()`. However, not all implementations provide sufficient functionality. For example, the MSVC compiler defaults to the root of the current drive as the temporary directory. This is a problem if drive permissions are set so that the caller does not have permission to open or write to files on that drive. These APIs work around problems in the native versions of `tmpfile()` and `tmpnam()`.

There is CPL support and implementations for all platforms and runtimes.

**cpl_tmpfile**

The `cpl_tmpfile` function is analogous to the ANSI/C function `tmpfile`. Its C signature is as follows:

```c
FILE *cpl_tmpfile(void);
```

**cpl_tmpnam**

The `cpl_tmpnam` function is analogous to the ANSI/C function `tmpnam`. Its C signature is as follows:

```c
char *cpl_tmpnam(char *);
```
cpl_tmpdir

An additional function, `cpl_tmpdir`, enables applications to query for the location of the system temporary directory on a particular platform (for example, `/tmp` on UNIX). Its C signature is as follows:

```c
char *cpl_tmpdir(char *);
```

cpl_tmpdir works much the same as `cpl_tmpnam` but it returns only a directory name. Applications can use the returned path to create their own transient files. The directory is not checked to be sure it exists.

Thread Yield (systhread)

The `cpl_THREADyield` function yields a timeslice for the Windows, Windows 95, or NetWare platforms. To include it in your code, specify the following:

```c
#include <cpl/config.h>
#include <cpl/systhread.h>
void cpl_THREADyield(int s);
```

UCB Compatibility Functions

You should avoid using the UCB compatibility functions `bcopy`, `bcmp`, and `bzero`. Instead, use `memmove` (rather than `bcopy`), `memcmp` (rather than `bcmp`), and `memset` (rather than `bzero`).

You can also use `memcpy` in place of `bcopy`, but it will not handle overlapping copies, as does `memmove`.

uname

The `uname` function enables callers to query for basic information about the running platform. The information returned is in a structure `utsname`. The structure is as follows:

```c
#define _SYS_NMLN 257
struct utsname
{
    char machine [_SYS_NMLN];
    char release [_SYS_NMLN];
    char sysname [_SYS_NMLN];
};
```
wstat Macros

    char version [_SYS_NMLN];
    char nodename [_SYS_NMLN];
};

To use the `uname` function, include the following in your code:

```c
#include <cpl/config.h>
#include <cpl/sys/utsname.h>

int uname (struct utsname *results_p);
```

There is CPL support for all platforms and CPL implementations for WIN 32 runtimes.

wstat Macros

These functions provide an implementation of the `wstat(5)` macros for use on PC platforms where they are not defined in a system file. The following macros are supported:

**WIFSTOPPED**(status)
This macro evaluates to a non-zero value if the status was returned for a child process that is currently stopped.

**WSTOPSIG**(status)
If the value of WIFSTOPPED(status) is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

**WIFSIGNALED**(status)
This macro evaluates to a non-zero value if status was returned for a child process that terminated because it received a signal.

**WTERMSIG**(status)
If the value of WIFSIGNALED(status) is non-zero, this macro evaluates to the number of the signal that caused the child process to terminate.

**WCOREDUMP**(status)
If the value of WIFSIGNALED(status) is non-zero, this macro evaluates to a non-zero value if a core image of the terminated child process was created.
Removing and Inserting Elements in a Queue

**WIFEXITED**\((\text{status})\)
This macro evaluates to a non-zero value if the status was returned for a child process that terminated normally.

**WEXITSTATUS**\((\text{status})\)
If the value of WIFEXITED\((\text{status})\) is non-zero, this macro evaluates to the exit code the child process passed to \_exit() or the value the child process returned from the main function.

These macros are of limited use on platforms that do not support the same \text{wait}() and/or \text{waitpid()} semantics as UNIX. Use with care.

For more information on the \text{wstat} macros, refer to the section 5 (Header, Tables, and Macros) UNIX manual page for \text{wstat}(5).

Removing and Inserting Elements in a Queue

Use the \text{insque} and \text{remque} functions to insert or remove an element in a queue. They are included in \text{libcpl} to be used by platforms that do not provide these functions. To insert an element in the queue:

```
#include <cpl/config.h>
#include <cpl/insque.h>

void cpl_insque(struct qelem * elem, struct qelem * pred);
```

To remove an element from the queue:

```
void cpl_remque(struct qelem * elem);
```
An LCF method may be implemented across several files. Besides the executable binary, the method may also require a shared library, message catalogs, or other files. In addition, an LCF method may require other software, such as a perl interpreter or the Java Virtual Machine (VM).

In the full framework, only the method body, the binary program or script that contains the method entry point, is stored in the method header. Any other files that the method requires at run-time are assumed to be present. This is true in the full framework, because all binaries, libraries, message catalogs, and so on are installed on every managed node.

However, LCF endpoints do not have any methods or supporting files present on them when endpoints are initially installed. Method bodies are identified in the standard method header and are downloaded on demand. This chapter discusses how the files that support an endpoint method are identified and downloaded.

When an LCF application is installed, the method bodies and supporting files (called dependencies) are copied to each gateway and are stored in the gateway repository. When the gateway is called on to download a method to an endpoint, it searches its repository for the method bodies and dependency files.
Specifying Dependencies

You specify dependencies in the .ist file using the wdepset command. For more information on the wdepset command, see “The wdepset Command” on page 6-3.

To make application development easier, you specify dependencies in a general way, and the paths to the actual files are resolved at run-time. In the wdepset command, you identify the dependency type with a resolve tag. The resolve tag can be one of the following:

- **bin** A binary program. The dependency file resides in the bin directory of the gateway repository. The path to the file is resolved at run-time based on the target endpoint’s interpreter type.
- **lib** A shared library. The dependency file resides in the lib directory of the gateway repository. The path to the file and the shared library suffix are resolved at run-time based on the target endpoint’s interpreter type.
- **$INTERP** The file is downloaded only for endpoints of the type $INTERP. No path resolution is performed.
- **generic** The dependency file is downloaded for each endpoint. No path resolution is performed.
- **depset** The dependency is another dependency object.

Implementing Dependencies

The dependencies are supplied on a DependencyMgr object, specified by the Depends::Mgr IDL interface. You specify this interface when you use the wchdep command to associate a method with its dependency. You can create a new object that will hold your dependencies, or your class can inherit from Depends::Mgr.

Commands to Specify Dependencies

You use two commands to create and manipulate dependencies:

- First, use the wdepset -c command to create a dependency set.
Then, use the wchdep command to associate the dependency set with a particular method header.

The wdepset Command

The wdepset command has four options: create, delete, view, or resolve a dependency set.

Creating a Dependency Set

To create a dependency set, use this command:

```bash
wdepset -c label [-C class] -a tag path [ +flags ]
```

The wdepset -c command takes the following parameters:

- **label**: The name of the dependency set.
- **class**: The name of the class (if not DependencyMgr).
- **tag**: One of the resolve tags: bin, lib, $INTERP, generic, or depset.
- **path**: The path to the file.
- **flags**: Two values are supported:
  - **+d**: Specify the directory in the endpoint cache where the dependency will be put.
  - **+p**: The path (in the dependency name) being passed down is an absolute path, not a path relative to the cache.
  - **+x**: Signal to the LCF endpoint to attempt to execute the dependency before receiving any remaining dependencies and before running the method.

The following example creates a dependency set named hello_depset. The example specifies that the dependency set is a library (-a lib) named libfoo:

```bash
wdepset -c hello_depset -a lib libfoo
```
The following example creates a dependency set named `foo` and uses the `+d` flag to tell the gateway to send the dependency (`sh.exe`) down to the endpoint as `bin/w32ix-86/sh.exe`:

```
wdepset -c foo w32-ix86 ../w32-ix86/tools/sh.exe +d bin/w32-ix86
```

**Deleting a Dependency Set**

To delete a dependency set, use this command:

```
wdepset -d objspec
```

The `wdepset -d` command takes the following parameters:

- `objspec` The name of the dependency set.

**Viewing a Dependency Set**

To view a dependency set, use this command:

```
wdepset -v objspec
```

The `wdepset -v` command takes the following parameters:

- `objspec` The name of the dependency set.

**Resolving a Dependency Set**

To resolve a dependency set, use this command:

```
wdepset -r objspec
```

The `wdepset -r` command takes the following parameters:

- `objspec` The name of the dependency set.

**The wchdep Command**

The `wchdep` command associates a dependency set with a particular method header. You must first use the `wdepset -c` command to create the dependency set, then use the `wchdep` command to specify which method header to associate the dependency set with.

The `wchdep` command has the following form:

```
wchdep @Classes:class-name @DependencyMgr:depset method-name
```
where *class-name* is the name of the class, *depset* is the label specified in the *wdepset* command, and *method-name* is the name of the method the dependency set supports.

### Updating All Dependencies

The *wgetdeps* command updates all dependencies for a class or method. It has the following form, where the *scoped-name* is a path to a class or method in the interface repository:

```
wgetdeps scoped-name
```

### Clearing Out-of-Date Dependencies

Over time, dependency and method header information may accumulate. To clear out-of-date dependencies and method headers, use this command:

```
wgateway gateway-label dbcheck
```

### Out-of-Cache Dependencies

In addition to dependencies that are stored in the gateway’s method cache, the LCF dependency mechanism allows you to specify out-of-cache dependencies. An out-of-cache dependency is one that is not contained in the endpoint’s method cache. Rather, it is in persistent store on the endpoint. By default, *lcfd* interprets the dependency as relative to the endpoint’s cache. But with an out-of-cache dependency, you specify the absolute path to the dependency.

This mechanism means you can write dependencies anywhere on the local machine, instead of *lcfd* putting the files in the cache and then your having to run a method or script to move them to their ultimate location. This is useful if an application has a dependency which is, for example, a permanently deployed product.

To specify an out-of-cache dependency, use the *wdepset* command with following syntax:

```
wdepset -c dependency_label -a interp dependency +p \dependency_base
```
For out-of-cache dependencies, you must always include the +p flag to indicate that the path to the dependency is interpreted as an absolute path, not relative to the cache.

For example, assume your dependency is an application called sentry.nlm that runs on NetWare. The dependency location is sentry/interp for both NetWare interpreter types (nw3 and nw4).

First, to create the dependency set and specify its out-of-cache location in sys\tivoli, use the wdepset command with the following syntax:

```
wdepset -c my_depends_label -a nw3 sentry/nw3/sentry.nlm \
    +p sys:/tivoli -a nw4 sentry/nw4/sentry.nlm +p sys:/tivoli
```

The dependency will be placed in the directory `sys:\tivoli\sentry\interp` for both interpreter types, depending on the interpreter of the endpoint machine when the method is invoked.

After you create the dependency set, use the wchdep command to associate the dependency object with your method.

```
wchdep @Classes:my_method_class \n    @DependencyMgr:my_depends_label method_name
```

When `method_name` is invoked, LCF moves the dependency `sentry.nlm` down to the endpoint to the location you specified.

In the wdepset command, you can also use the +x flag to signal the LCF endpoint to execute a dependency before receiving any remaining dependencies and before running the method. This is useful when you need to launch a dependency as a stand-alone utility that installs or launches an application prior to running the method. Note that a dependency executed with this flag must be a relatively short-lived process; receiving other dependencies and running the method is blocked until the dependency process completes.

The method or stand-alone executable (if linked with libmrt) can use the mrt_check_dependency() API, which tells the caller whether a dependency has been updated. This is useful for applications that need to kill and then re-launch a long-running or daemon-like application.
Example

You place the `wdepset` and `wchdep` invocations in the `.ist` file, so the resulting commands will be invoked from the `.cfg` file, which is generated by the TEIDL compiler.

The following example, `initialize`, creates a dependency set named `hello_depset` that contains a single shared library dependency. The `wchdep` command associates `hello_depset` with the `Hello::Hello_meth` method.

```plaintext
initialize {
   "wdepset -c hello_depset -a lib libfoo";
   "wchdep @Classes:Hello @DependencyMgr:hello_depset hello_meth";
}
```

The following example `.ist` file illustrates setting a dependency for a downcall that requires `libmrt`.

```plaintext
module ist_Hi {
   installation Hello_World for imp_Hi::Hello_World
      with prog_Hi::Hello_main {
   acldefault { "any" };
   external path {
      ( echo ) = { "default", "/TMF/LCF/Hello_main"; };
   }
   initialize {
      // // Set the name of our class in the name registry
      // "set_friendly_name $imp_Hi_Hello_World_CO Hello";
      // // Create a dependency set for libmrt, add this
      // dependency set to the requirements for the
      // echo method
   }
}
```

Dependencies
Example

24        //
25      "wdepset -c libmrt -a lib libmrt";
26
27      "wchdep @Classes:Hello @DependencyMgr:libmrt
          echo";
28      
29    
30  
31
32  #endif
CCMS for LCF Applications

Using CCMS for Endpoint Applications

The Configuration and Change Management System (CCMS) provides a model for configuration management that many applications use. CCMS enables you to set up profiles, which are sets of related parameters that form a description of a resource’s configuration. These profiles can be organized into collections called profile managers and then subscribed to by profile endpoints. The TME 10 administrator manipulates a system’s configuration by editing the profiles in a profile manager and distributing the manager’s profiles to its subscribers. For more information about CCMS, see the chapter titled “Configuration and Change Management System” in the Application Services Manual, Volume 2. In addition, Application Development with TME 10 ADE describes an application, Tivoli/Phone, that uses CCMS. The Tivoli/Phone application supports and works with LCF endpoint clients.

The LCF provides support for CCMS. This chapter provides information you’ll need to know for endpoint applications that use CCMS:

- CCMS functions supported in LCF
- CCMS in the LCF environment
- The IDL file to include
- The header files to include
- The libraries to link to
CCMS Functions Supported in LCF

The following CCMS convenience functions, which operate identically in both the full framework and LCF, are supported for LCF:

- `ccms_find_record_item()`
- `ccms_read_next_record()`
- `ccms_read_pushed_records()`
- `ccms_read_some_records()`
- `ccms_read_update_parameters()`

The table below provides a brief description of each of the CCMS convenience functions implemented for the LCF. These functions help process the data sent to the endpoint method through an MDist channel.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ccms_read_update_parameters</code></td>
<td>Opens the MDist channel and reads the <code>update_params</code> structure. You must call this function before calling any of the other convenience functions.</td>
</tr>
<tr>
<td><code>ccms_find_record_item</code></td>
<td>Search for and return a record in the sequence returned by <code>ccms_read_pushed_records()</code> or <code>ccms_read_some_records()</code> that matches the key value specified. Return NULL if the key is not found.</td>
</tr>
<tr>
<td><code>ccms_read_next_record</code></td>
<td>Reads the stream of records sent to the endpoint, and returns the next record in the MDist channel.</td>
</tr>
<tr>
<td><code>ccms_read_pushed_records</code></td>
<td>Reads the stream of records sent to the endpoint, and returns a sequence that contains all the records in the MDist channel.</td>
</tr>
</tbody>
</table>
CCMS in the LCF Environment

The LCF environment implements the same CCMS model as the full framework: profile managers contain profiles, and endpoints are subscribers to those profile managers. When you perform a distribute operation, data is sent to endpoints in the same way as it is in the full framework. LCF provides CCMS functionality so that endpoint applications can continue to read their datastreams.

This results in seamless integration of all types of endpoints, whether they are managed nodes or endpoints. When you run a CCMS-based application, endpoints and managed nodes are, from the user or administrator perspective, the same:

- Both managed nodes and endpoints subscribe to one or more profile managers.
- You can optionally tune or configure MDist to push the data out to the endpoints.
- The distribution goes to both managed nodes and endpoints.

But there are a few differences to be aware of. There is no database on an endpoint, as there is on a managed node. Data is sent to the application on the endpoints, but it is not written to a database on the endpoint. If your application depends on reading from or writing to an endpoint database, it will need to change to run on an endpoint.

There is endpoint support for populating a profile. An LCF application can provide the capability for an endpoint to send information about

---

**Function** | **Description**
---|---
ccms_read_some_records | Reads the stream of records sent to the endpoint, and returns a sequence that contains the requested number of records from the MDist channel.

For more information about these CCMS convenience functions, see the chapter titled Configuration and Change Management System in the Application Services Manual, Volume 2.
itself up the hierarchy of profile managers. The information about the endpoint is then stored by the profile manager.

Two database-dependent CCMS functions are not supported on endpoints: the synchronize functions `ccms_sync_endpt_ex` and `ccms_synchronize_endpoint`. These synchronize functions both make use of an endpoint database.

Also, AEF actions are not supported on LCF endpoints, so no TME 10 AEF actions execute at the endpoint. AEF actions can only be shell scripts, and the majority of the platforms for LCF endpoints do not support a Bourne shell interpreter.

Applications that need to do generic scripting actions can use the Task Library outside of CCMS. While this is not quite the same as using actions, it may serve in situations where the script to be executed is not tightly bound to the data in the distribution.

**Profile Managers in Dataless and Database Mode**

CCMS for endpoint applications uses the same profile manager as all other CCMS applications do. However, to accommodate dataless endpoints (no database on endpoints), there is an additional attribute you can set so that the profile manager runs in a dataless mode.

During a distribution, a profile manager normally writes the data first to the database, then to the application. When you set the attribute so that a profile manager runs in a dataless mode, the profile manager writes the data only to the application.

It is important to note that the profile manager is the same for endpoint applications and all other applications. The only difference is whether or not this one attribute is set—when it is not, the profile managers are exactly the same.

When a profile manager is in dataless mode, the endpoints it distributes to are dataless, by virtue of the fact that no data is written to a database. The following two sections provide more details about the dataless and database modes.
Database Mode

The database mode is the default mode for a profile manager. This mode is the traditional way profile managers have operated in TME 10. The following is an example of an icon for a profile manager running in database mode:

Profile managers running in the database mode can distribute only to managed nodes, PC managed nodes, NIS domain objects, and other profile managers. They cannot distribute to endpoint clients.

When you create a profile in a profile manager, it is stored in a platform-independent profile database. When you distribute the profile, the entire profile database is distributed to the subscribers. You control how far the profile is distributed by choosing the level of distribution. You can distribute to only the next level of subscribers or to all levels of subscribers.
The following diagram shows a profile manager with two levels of subscribers, labelled Level 1 and Level 2:

When you distribute the contents of Profile Manager A to the next level of subscribers, profiles P1, P2, and P3 are distributed to Profile Manager B and to managed node Taki. A copy of the profile database on profile manager A is created in profile manager B and in the TME 10 client database of managed node Taki.

After this distribution, the changes included in the profiles are available through (stored in) the TME 10 client database, but the changes have not been applied to any system files on Taki. TME 10-based applications that look for information in the client database can use the profile information. However, applications (TME 10-based or otherwise) that look for information in the system files will not be able to access the newly distributed information (because those files have not yet been updated). To update the system files after a distribution to Level 1 subscribers, you must perform a distribute operation from the profile endpoint (in this case, Taki).
When you distribute to all levels of subscribers, the system files are automatically updated for all profile endpoints receiving the distribution. For example, if you distribute the contents of profile manager A to Level 1 and Level 2 subscribers, the three profiles are distributed not only to profile manager B and managed node Taki, but also through profile manager B to managed nodes Ming and Raul. The system files on managed nodes Taki, Ming, and Raul are automatically updated.

**Dataless Mode**

The following icon represents a profile manager operating in dataless mode:

![Profile Manager Icon](Image)

Profile managers running in dataless mode can distribute only to profile endpoints: managed nodes, PC managed nodes, and endpoints. They cannot distribute to other profile managers.

In addition, distributions in the dataless mode cannot have multiple subscription levels. Finally, endpoint clients can only subscribe to profile managers running in the dataless mode; they cannot subscribe to profile managers running in the database mode.
The following diagram shows Profile Manager A, running in dataless mode:

Profile Manager A distributes profiles P1 and P2 to three profile endpoints: PC managed node Red, managed node Oak, and endpoint Ruby. Note that Profile Manager A does not distribute to any profile managers, only profile endpoints, and has only one subscription level. Because Profile Manager A is distributing to an endpoint, Ruby, it must run in dataless mode.

When a dataless profile manager distributes to its subscribers, all profile changes are made directly to the system files. No information is written to client databases on subscribing managed nodes or PC managed nodes.

In summary, keep the following key points in mind:

- Endpoints must subscribe to a profile manager that is enabled for dataless mode.
- Managed nodes can subscribe to a profile manager in either regular or dataless mode.
- A profile manager in database mode cannot subscribe to a profile manager in dataless mode.
- However, profile managers in dataless mode can subscribe to profile managers in database mode.
Note: For managed nodes, you have the ability to bypass the database on the endpoint. You can choose to have a dataless endpoint or to store information in the database. You select this based on the type of profile manager you subscribe the endpoint to.

The following figure illustrates a possible hierarchy for profile managers, profile managers in dataless mode, managed nodes, and endpoints.

IDL File

When you are writing endpoint IDL, your application must include the file `CCMS_types.idl`. It contains the public IDL types exported by CCMS. To include it, add the following to your code:

```cpp
#pragma generate False
#include <mrt/stypes.idl>
#include <ccms/CCMS_types.idl>
#pragma generate True
```
Header Files

You must include the following header files in your CCMS endpoint application:

- **CCMS_types_adr.h**—Contains IDL-generated type definitions used by CCMS and CCMS clients.
- **ccms_util.h**—Contains function prototypes for the CCMS application endpoint interfaces.

CCMS_types_adr.h and ccms_util.h are located in the directory include/$(INTERP)/ccms. Include them in your code as follows:

```c
#include <ccms/CCMS_types_adr.h>
#include <ccms/ccms_util.h>
```

CCMS Library

The CCMS library interface is libccms_lcf.a. This library contains the compiled IDL type information and the CCMS functions prototyped in ccms_util.h. Applications that use libccms_lcf.a must create a dependency on the shared library (libccms_lcf.dll) using the wdepset command.
Task Library

This chapter describes how to include data types from the Task library in your endpoint application. There are no Task library functions or methods to be exported for use by applications, only data types.

In much the same way as you do for CCMS, you need to include or use files that are exported for use with ADE. For more information about tasks and the Task library, see Chapter 6, “Task Library,” in the Application Services Manual, Volume 2.

Using the Task Library Data Types

To use the ADR data types from the Task library, your application must make a call to `adr_type_init()` to initialize the type repository. Because the Task library has no utility functions, it is unable to auto-initialize the type repository.

Make the following sequence of calls somewhere in the start-up of your application code. You should only make this call once.

```c
#include <task/task_adr.h>

type_repository* t[2];

t[0] = task_type_repository;
t[1] = type_repository_null;
adr_type_init(t);
```
Task Library

The Task library interface is libtask_lcf.a. This library contains the compiled IDL type information. There are no functions or methods to be exported for use by applications, only data types, so there is no exported DLL.

IDL File

When you are writing endpoint IDL, your application must include the file <task/task.idl>. It contains the public IDL types exported by the Task library. To include it, add the following to your code:

```c
#pragma generate False
#include <task/task.idl>
#pragma generate True
```

Header Files

You must include the following header file in your endpoint application:

- `<task/task_adr.h>`—Contains IDL-generated type definitions used by the Task library.

Include it in your code as follows:

```c
#include <task/task_adr.h>
```
This chapter describes a sample LCF application called `record_gen`. It discusses the source files and their layout in the source tree. The source code for `record_gen` is furnished with the LCF.

The sample application illustrates:

- How to structure an LCF application into endpoint code, platform code, and common code (shared by the endpoint and platform).
- How to write an upcall collector.
- The types of code you might write to move a file from one system to another (in this case, distribute it from gateway to endpoint), do an upcall that is collected by an upcall collector and then batched to a Mid-level Manager (MLM).

The `record_gen` application takes a file similar to the `/etc/hosts` file, parses it into records, and sends the records from the gateway to endpoints. The endpoints also do upcalls, and the upcall data is collected by an upcall collector.

This sample application is intended to illustrate upcalls, downcalls, and an upcall collector, rather than to be a useful working example. For example, this sample has a Mid-level Manager (MLM) that accepts the batched upcalls, but it does no meaningful processing of them.
The Source Tree

This section discusses the directory and file structure for the sample application; use a similar structure for an application that has both platform and endpoint components. The diagram below shows the structure of the source tree for the `record_gen` sample application:

Within the source tree, there are three directories:

- **Endpoint**—Contains files unique to the endpoint side. The makefile builds an LCF method.
- **Platform**—Contains files unique to the server side. The makefile builds a TME 10 command line program.
- **Common**—Contains files common to both endpoint and server. Note that no builds take place in this directory; there are no makefiles. The common files are referenced by relative path from the endpoint and platform directories.

Maintaining this directory layout when you structure your application has several important advantages:

- As you develop your application, it enables you to build and test the endpoint and server sides independently of each other.
- It makes porting your application easier; that is, if you are porting your application to a new platform, you only have to port the endpoint-side code.

If necessary, you can use `#ifdef ENDPOINT_BUILD` in your source files to distinguish between the TME 10 server-side and the endpoint-side code.
The following table lists the files for the `record_gen` application by directory (common, endpoint, and platform).

<table>
<thead>
<tr>
<th>Directory</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common (IDL files)</td>
<td>common/Samp.idl common/Samp.imp common/Samp.ist</td>
</tr>
<tr>
<td></td>
<td>common/Samp.prog</td>
</tr>
<tr>
<td>Common (other files)</td>
<td>common/par.c common/rsprintf.c common/rsprintf.h</td>
</tr>
<tr>
<td></td>
<td>common/seq.h</td>
</tr>
<tr>
<td>Endpoint (client-side)</td>
<td>endpoint/Makemfile endpoint/configure.mk endpoint/src/Makefile</td>
</tr>
<tr>
<td></td>
<td>endpoint/src/ep_meth.c endpoint/src/up.c</td>
</tr>
<tr>
<td>Platform (server-side)</td>
<td>platform/Makemfile platform/configure.mk platform/src/Makefile</td>
</tr>
<tr>
<td></td>
<td>platform/src/samp.c platform/src/upserv.c</td>
</tr>
<tr>
<td></td>
<td>platform/src/my_t_MLM_meth.c platform/src/my_t_UpcallCollector_meth.c</td>
</tr>
<tr>
<td></td>
<td>platform/src/global.h</td>
</tr>
</tbody>
</table>

**Source Files**

This section briefly describes the contents of each file in the sample application, by directory.

**Files in the Common Directory**

The following files are contained in the common directory.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samp.idl</td>
<td>The interface definition (IDL) for the sample application.</td>
</tr>
</tbody>
</table>
## Source Files

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samp.imp</td>
<td>The implementation file for the sample application.</td>
</tr>
<tr>
<td>Samp.prog</td>
<td>The program file for the sample application.</td>
</tr>
<tr>
<td>Samp.ist</td>
<td>The installation file for the sample application.</td>
</tr>
<tr>
<td>par.c</td>
<td>File parsing functions to turn the file into records.</td>
</tr>
<tr>
<td>rsprintf.c</td>
<td></td>
</tr>
<tr>
<td>rsprintf.h</td>
<td></td>
</tr>
<tr>
<td>seq.h</td>
<td></td>
</tr>
</tbody>
</table>

## Files in the Endpoint Directory

The following files are contained in the endpoint directory.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ep_meth.c</td>
<td>A template generated by the TEIDL compiler. Contains the method implementation for the <code>dist</code> method.</td>
</tr>
<tr>
<td>up.c</td>
<td>Makes the stub call to the upcall collector.</td>
</tr>
<tr>
<td>configure.mk</td>
<td>Build settings for endpoint build. Includes interpreter types to build, directories, IDL compiler script to use (<code>ltid</code>), and so on. Used as input to <code>Makefile</code>.</td>
</tr>
<tr>
<td>Makefile</td>
<td>Makefile for endpoint build.</td>
</tr>
</tbody>
</table>

## Files in the Platform Directory

The following files are contained in the platform directory.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>samp.c</td>
<td>The platform-side executable for <code>record_gen</code>.</td>
</tr>
<tr>
<td>upserv.c</td>
<td>The code that implements the upcall collector.</td>
</tr>
<tr>
<td>my_t_MLM_meth.c</td>
<td>The skeleton for the MLM.</td>
</tr>
</tbody>
</table>
When designing your LCF application and writing the IDL files, it is important to understand the components your application will require and where each will run. For example, for \texttt{record\_gen}, there are three separate interfaces in the .\texttt{idl} file:

- **MLM** is the interface for the server component
- **UpcallCollector** is the interface for the gateway component
- **Endpoint** is the interface for the endpoint component

Each interface will have its own implementation, resulting in three different sets of binaries to distribute to three different systems. This differs from full-framework development, where you do not need to differentiate among application components according to where they will run.

For more information about the components of an LCF application, see Chapter 2, “LCF Application Design.”

The \texttt{record\_gen} application illustrates the following basic sequence of steps for writing and building any new LCF application. Keep in mind that these general steps apply to a new application, not for building \texttt{record\_gen} in particular.

### Sequence of Steps for Writing and Building an LCF Application

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{my_t_UpcallCollector.c}</td>
<td>The skeleton for the upcall collector.</td>
</tr>
<tr>
<td>\texttt{global.h}</td>
<td>Common declarations.</td>
</tr>
<tr>
<td>\texttt{configure.mk}</td>
<td>Build settings for gateway components. Includes interpreter types to build, libraries to include, and so on. Used as input to Makefile.</td>
</tr>
<tr>
<td>Makefile</td>
<td>Makefile for platform-side build.</td>
</tr>
</tbody>
</table>
1. Write the .idl, .imp, .prog, and .ist files for your application. These files are common to both the endpoint and server side of your application and reside in the src/app/common directory. Provide interfaces (as needed) for each component (server, gateway, endpoint) of your application.

2. Determine which parts are exclusive to the endpoint and which are exclusive to the platform, and be sure those files are in their respective directories. By the same token, determine which parts of the code in your application are common to both endpoint and platform. At a minimum, the IDL files (the .idl, .imp, .prog, and .ist files) will be common to both endpoint and platform.

   In record_gen, for example, samp.c is exclusive to the server side, and ep_meth.c is exclusive to the endpoint side. up.c is exclusive to the endpoint side, and upserv.c is exclusive to the gateway.

3. Build the files on the server side. To do this, run the TEIDL compiler (tidlc) in the platform directory. Be sure to include the common files when you build. (“Building the record_gen Sample Application” on page 9-28 contains more details.)

4. The first time you build the server side, the TEIDL compiler produces a stubbed-out version of the methods, among many other files. For example, for record_gen, you would copy the file t_Endpoint_meth.c (the method template produced by the TEIDL compiler) from the platform/interp directory to the record_gen endpoint source directory, renaming the file to something meaningful. In this case, it is named ep_meth.c. This file contains the C code for the dist method. Do the same for other endpoint files.

   It is important to note here that the stub files for the endpoint are produced by tidlc on the server side and must be moved to the endpoint.

   For more information about the output generated by the TEIDL compiler, see the chapter titled “TME 10 ADE Extended IDL” in the Framework Services Manual.
5. Build the endpoint side, using \texttt{ltid}. ("Building the record_gen Sample Application" on page 9-28 contains more details.)

Notice that you are compiling the IDL files twice: once for the server side, using \texttt{tidlc}, and once for the endpoint side, using \texttt{ltid}.

Building LCF applications differs somewhat from full framework builds. You build on both the endpoint and server sides, rather than in one place. The common files are generated on both sides (endpoint and platform), rather than being referenced from the common area. This is because there are two slightly different application development environments: the full framework and LCF. The two sides also build slightly differently: the server side uses \texttt{tidlc}, and the endpoint side uses \texttt{ltid}.

The IDL Files (Common)

The IDL Definition File: Samp.idl

The file \texttt{Samp.idl} defines a module, SAMP. It includes a data type and two structures:

- A data type that is a \texttt{sequence} of record structures.

  \begin{verbatim}
  typedef sequence<record> recordList;
  \end{verbatim}

  A sequence is a one-dimensional array of elements. The CORBA specification defines the sequence data type for operations that accept or return a set of data structures. TME 10 ADE provides a library of functions for manipulating sequences.

- A structure called \texttt{header} that maintains the value for a header record. The operations in the \texttt{record_gen} sample application use this structure to pass data.

\begin{verbatim}
1 /* header record */
2 struct header {
3     string      filename;
4     string      source;
5     long        flags;
6 }
\end{verbatim}
A structure called `record` that maintains the value for each record. The operations in the `record_gen` sample application use the `record` structure to pass data.

```c
/* file records */
struct record {
    long id;
    TMF_Types::StringList tokens;
    string leading_comment;
    string eol_comment;
};
```

The IDL Files (Common)

**Samp.idl** specifies three interfaces:

- **Endpoint**—It specifies one operation, `dist`, supported by `Endpoint`. The `dist` operation has two input parameters, `arg` and `dkey`, and one output parameter, `dout`. There is no exception handling. This interface constitutes the endpoint component of the application.

- **Upcall collector**—It specifies a method, `upcall`, which is the gateway component of the application.

- **Mid-level manager**—It specifies a method, `mlm_in`, which is the server component of the application.

The code for **Samp.idl** is shown here:

```c
#if !defined Samp_idl
#define Samp_idl

#pragma generate False
ifdef ENDPOINT_BUILD
#include <mrt/stypes.idl>
define Object string
#else
#include <tivoli/TMF_Types.idl>
endif
#pragma generate True

module SAMP {
    /* general arg type */
typedef sequence<octet> mdata;
```
The following code specifies the **Endpoint** interface and its method, `dist`.

```cshape
interface Endpoint {
  // testing endpoint
  void dist (in any arg,
             in mdata dkey,
             out mdata dout);
}
```

The following code specifies the **UpcallCollector** interface, which is the gateway component, and its method, `upcall`.

```cshape
interface UpcallCollector {
  typedef struct config_struct {
    long timeout;
    long queue_max;
  } config_t;

  oneway void router ();
  void upcall (in record item);
  config_t get_config ();
  void set_config (in config_t cfg);
}
```

The following code specifies the MLM interface, which is the server component, and its method `mlm_in`.

```cshape
#ifndef ENDPOINT_BUILD
```
The IDL Files (Common)

```c
51 interface MLM {
52   void mlm_in (in recordList list);
53 }
54 #endif
55 #endif
```

The Implementation File: Samp.imp

The file `Samp.imp` contains the implementation of the class `Samp`, defined in `Samp.idl`. It is an abstract class: it acts as the base class for inheritance and cannot be instantiated. It honors the SAMP::Endpoint interface.

`Samp.imp` specifies that one method, `dist`, will have a method implementation, and the language binding for the method is ANSI C. The code for `Samp.imp` is shown here:

```c
1 #if !defined Samp_imp
2 #define Samp_imp
3
4 #include "Samp.idl"
5
6 module imp_SAMP {
7
8 implementation abstract class Endpoint honors SAMP::Endpoint {
9   methods {
10     {dist } binding = ansi C; }
11   }
12 }
13
14 implementation abstract class UpcallCollector honors SAMP::UpcallCollector {
15   methods {
16     {router,
17       upcall,
18       get_config,
19       set_config,
20     } binding = ansi C;
21   }
22 }
23 #ifndef ENDPOINT_BUILD
24
```
#ifdef Samp_prog
#include "Samp.imp"

module prog_SAMP {
  program Endpoint for imp_SAMP::Endpoint {

The Program File: Samp.prog

A program file specifies the execution characteristics for methods in a particular class. The file Samp.prog specifies the following for the method dist:

- Method name
- Execution style
- Server type
- Start-up specification

In this case, Samp.prog specifies that the method dist is to be executed as a “per-method” method; that is, the server executes it and then terminates. The startup keyword specifies that when a call is made to dist, the server checks to see if tas_init is running. If not, it launches a process to execute it.

In addition, Samp.prog includes the program specifications for the methods in the UpcallCollector and MLM interfaces.

The code for Samp.prog is shown here:

```c
#include "Samp.imp"

module prog_SAMP {
  program Endpoint for imp_SAMP::Endpoint {
```
executes {
  dist
} per method;

startup {
  "tas_init"
};
}

program UpcallCollector timeout = 300 for imp_SAMP::UpcallCollector {
  executes {
    router,
    upcall,
    get_config,
    set_config
  } threaded daemon;

  startup {
    "tas_init", "UpcallCollector_init"
  };
}

#ifndef ENDPOINT_BUILD

program MLM timeout = 300 for imp_SAMP::MLM {
  executes {
    mlm_in
  } threaded daemon;

  startup {
    "tas_init"
  };

#endif

#endif
The IDL Files (Common)

The Installation File: Samp.ist

The installation file defines how the class implementation SAMP behaves when installed in TME 10. The file Samp.ist describes the authorization, privilege, and location of the executables for the class SAMP. The keyword acldefault specifies the default ACL within an installation, in this case “any.” The external path “default” specifies that a different executable is required for each platform and furnishes the path for the executable.

The code for Samp.ist is shown here:

```c
1  if !defined Samp_ist
2     #define Samp_ist
3
4     #include “Samp.prog”
5
6     module ist_SAMP {
7       installation Endpoint for imp_SAMP::Endpoint with
8         prog_SAMP::Endpoint {
9
10      acldefault { “any” };
11
12      external path {
13         { dist } = { “default”, “/LCF/SAMPLES/record_gen”; };
14       };
15
16      installation UpcallCollector
17         for imp_SAMP::UpcallCollector with
18           prog_SAMP::UpcallCollector {
19
20      acldefault { ADMIN_ACL };
21
22     // Set up authorization for upcall method. It is
23     // special because we want anyone to be able to call
24     // it, but it needs enough privileges to do some basic
25     // operations (like name registry lookups).
26     acl {
27         { upcall } = { ANY_ACL };
28       };
29     roles {
```
( upcall ) = ( GLOBAL_SECURITY_GROUP, ADMIN_ROLE, USER_ROLE );

external path {
    
    router,
    upcall,
    get_config,
    set_config
} = { "default", "/LCF/SAMPLES/upcall_collector"; }

initialize {
    "set_friendly_name $imp_SAMP_UpcallCollector_CO UpcallCollector";
}

#ifndef ENDPOINT_BUILD
installation MLM for imp_SAMP::MLM with prog_SAMP::MLM {

    acldefault { ADMIN_ACL };

    external path {
        mlm_in ) = { "default", "/LCF/SAMPLES/mlm"; }
    }

    initialize {
        "set_class_name_cli $imp_SAMP_MLM_CO SampMLM";
        "wregister SampMLM `objcall $imp_SAMP_MLM_CO _create_instance -l SampMLM`";
    }
}
#endif
When you run the TEIDL compiler on your IDL files, it generates, among other files, two empty method templates for every method: a CORBA template and a TME 10 template. Because the .imp file specified the ANSI C language binding, the method template is a function header with an empty body. The name of the method template is composed of the operation and class name and the module. The TME-style template also has a t_ prefix. In this, the template is named t_Endpoint_meth.c.

Before you begin filling in the template, rename it when you copy it to the endpoint directory. In this case, it’s renamed to ep_meth.c. The following shows the code for ep_meth.c:

```c
1 */
2 *****************************************/
3 *
4 * File Name: t_Samp_main_meth.c
5 * Tivoli EIDL Compiler (Version 1.05, Date 01/30/95)
6 * generated ANSI C Tivoli method implementation File.
7 * LCF node side
8 *
9 * Edit this file to fill in method implementations.
10 *****************************************/
11 */
12 #include <mrt/tiv_mrt.h>
13 #include <mrt/mrt_run.h>
14 #include <mrt/mrt_wire_adr.h>
15 #include <mrt/iom_mdist.h>
16 #include “t_Samp.h”
17 #include “Samp.h”
18
19 void t_imp_SAMP_Endpoint_dist(
20     SAMP_Endpoint _SAMP_Endpoint,
21     Environment *_ev,
22     any* arg,
23     SAMP_mdata* dkey,
24     SAMP_mdata* dout)
25 {
```
ich = ioch_mdata_init((char *)dkey->_buffer, dkey->_length);

/* get header record */
cnt = ioch_recv(ich, &data, 0);
tmf_decode(TC_SAMP_header, data, cnt, &hdr);
if ((ofs = fopen(hdr.filename, "w")) == NULL) {
    vaThrow(CatSpec(cat, fopen),
            "$1$s: Open failed: $2$s",
            hdr.source, "$errno");
}
/* write header record to file */
fprintf(ofs, "#\nsource=%s flags=%ld\n",
        hdr.source, hdr.flags);
tmf_free_generic(&hdr, TC_SAMP_header);
while ((cnt=ioch_recv(ich, &data, 0)) > 0) {
    tmf_decode(TC_SAMP_record, data, cnt, &rec);

    /* write record */
    if (rec.leading_comment)
        fprintf(ofs, "%s",
                rec.leading_comment);
    for (i=0; i<rec.tokens._length; i++)
        fprintf(ofs, "%s ",
                rec.tokens._buffer[i]);
    if (rec.eol_comment)
        fprintf(ofs, "%s\n",
                rec.eol_comment);
    else
        fprintf(ofs, "\n");
    tmf_free_generic(&rec, TC_SAMP_record);
Endpoint Methods

LCF Sample Application

63  index++;
64  }
65  fclose(ofs);
66  
67  dout->_buffer = mg_malloc(512);
68  sprintf(dout->_buffer, “Done. index=%d\n”,
       index);
69  dout->_length = strlen(dout->_buffer);
70 }
71

up.c

up.c contains the implementation for the upcall CLI. It calls the upcall stub, t_SAMP_UpcallCollector_upcall (line 43). It makes a bogus record and then calls the upcall stub.

1 #include <mrt/tiv_mrt.h>
2 #include <mrt/mrtwrap.h>
3 #include <mrt/mrt_wire_adr.h>
4 
5 #include “t_Samp.h”
6 #include “Samp_aux.h”
7 
8 void
9 main (int argc, char *argv[])
10 {
11   int count = 1;
12 
13 /* for this test upcall, it should complete within
14 * 60 seconds or something is wrong.
15 */
16   mrt_set_upcall_timeout (60);
17 
18 if (argc > 2)
19 {
20     fprintf (stderr, “usage: %s [count]\n”, argv[0]);
21     exit (1);
22 }
23 
24 if (argc == 2)
25   count = atoi (argv[1]);
26 if (count < 1)
27 {
27 fprintf (stderr, "usage: \%s \[count\]\n\", argv[0]);
28 exit (1);
29 }
30
31 Try {
32 Environment ev = {0};
33 SAMP_record rec;
34 int i;
35 Seq_zero (&rec.tokens);
36 rec.leading_comment = strdup ("leader");
37 rec.eol_comment = strdup ("eol");
38 for (i = 0; i < count; i++)
39 {
40   fprintf (stderr, "id = \%d\n", rec.id = i);
41   t_SAMP_UpcallCollector_upcall
42     ("UpcallCollector", &ev, 0, &rec);
43 }
44 }
45 }
46 Catch (Exception, ex) {
47 char *msg = def_ex_bind (ex);
48 fprintf (stderr, "EXCEPTION: \%s\n", msg);
49 msg_free (msg);
50 }
51 EndTry;
52 }
53
Platform Files

samp.c

samp.c is the implementation for the dist method that runs on the server. Only a part of the file is included here.
1 #include <stdio.h>
2
3 #include <tivoli/iom_mdist.h>
4 #include <tivoli/ExException.h>
5 #include <tivoli/sequence.h>
6 #include <tivoli/msg_utils.h>
```c
#include "Samp.h"
#include "Samp_defs.h"

extern char *optarg;
extern int optind;

/* config file */

char *prog;
char *cfile="./cfile";
char *meth_name = SAMP_EP_DIST;

/*
 * record_gen - data producer callback
 */

int record_gen(io_chan_t *oh, void *arg)
{
    SAMP_record rec;
    SAMP_header hdr;
    FILE *fs = (FILE *)arg;
    char *data;
    long datalen;
    int id = 1;

    /*
     * send header record
     */
    hdr.filename = cfile;
    hdr.source = prog;
    hdr.flags = 0;
    encode_tc(TC_SAMP_header, &hdr, &data, &datalen);
    if (ioch_write(oh, data, datalen) == -1) {
        fprintf(stderr, "record_gen: write error\n");
        return -1;
    }

    free(data);

    /*
     * read records, send them out
     */
```
while (par(fs, &rec) != -1) {
    rec.id = id++;
    /* encode it */
    encode_tc(TC_SAMP_record, &rec, &data, &datalen);
    /* send it */
    if (ioch_write(ch, data, datalen) == -1) {
        fprintf(stderr, "record_gen: write error\n");
        return -1;
    }
    free(data);
}
ioch_write(ch, NULL, 0); /* mark EOF */
return 0;

void
main1(int argc, char **argv) {
    TMF_Types_ObjectList olist={0};
    TMF_mdist_respList rlist={0};
    TMF_mdist_resp *resp;
    char cbuf[512];
    /* build target list */
    if (argc-optind < 1) {
        fprintf(stderr, "targets?\n");
        exit(1);
    }
    for (i=optind; i<argc; i++)
        seq_add((sequence_t *)&olist, &argv[i], sizeof(char *));
    /* open record file */
    if ((fs=fopen(cfile, "r"))==NULL) {
        perror(cfile);
        exit(1);
    }
upserv.c

upserv.c is half of the gateway component that, along with my_t_upcallCollector_meth.c, implements the upcall method.

It queues upcall requests from the endpoints and forwards them to the MLM. To do this, it uses semaphores to batch up the records and send them to the MLM when timeout is met or queue_max is reached.

```c
#include <tivoli/defines.h>
#include <sys/time.h>

#include <tivoli/pthread.h>
#include <tivoli/tmfthr.h>
#include <tivoli/timers.h>
#include <samp/t_Samp.h>
#include <samp/SampSeqMacros.h>
#include "global.h"
static struct node *head = NULL;
struct node **tail;
u_long list_len;
Object MLM;
```
15
16 extern u_long timeout;
17 extern u_long queue_max;
18
19 tmf_sema_t list_sem, full_sem;
20
21 int tm_id;
22
23 static void forward ();
24 void forward_when_full ();
25
26 void
27 UpcallCollector_init ()
28 { 
29   extern tmf_sema_t boot_mutex;
30
31   tail = &head;
32   list_len = 0;
33
34   tmf_PVinit (&list_sem, 0);
35   tmf_PVinit (&full_sem, -1);
36   tmf_PVinit (&boot_mutex, 0);
37 }
38
39 void
40 set_forward_timeout (long seconds)
41 { 
42   struct timeval how_long;
43
44   how_long.tv_sec = seconds;
45   how_long.tv_usec = 0;
46   tm_id = tmf_timeout (forward, 0, &how_long);
47 }
48
49 static void
50 forward (any_t arg)
51 { 
52   Environment ev = {0};
53   SAMP_recordList rlist;
54
55   SAMP_recordList_init (&rlist);
57   tmf_untimeout (tm_id);
58   tmf_P (&list_sem);
59
60   Try {
61     struct node *n;
62     ex_add_unwind (tmf_V, &list_sem);
63
64     n = head;
65     while (n)
66       {
67         struct node *prev;
68         SAMP_recordList_add (&rlist, n->data);
69         prev = n;
70         n = n->next;
71         mg_free (prev);
72       }
73     list_len = 0;
74     head = NULL;
75     tail = &head;
76   }
77   EndTry;
78
79   t_SAMP_MLM_mlm_in (MLM, &ev, Trans_none, &rlist);
80   SAMP_recordList_free (&rlist);
81   set_forward_timeout (timeout);
82 }
83
84 void
85 forward_when_full ()
86 {
87   while (1)
88     {
89     tmf_P (&full_sem);
90     forward (0);
91   }
my_t_MLM_meth.c

my_t_MLM_meth.c is the skeleton for the MLM. The MLM receives the calls but, in this case, does no processing; it drops the upcalls after it receives them and then listens again for upcalls. Usually, the application’s server component would process the data and possibly forward it to a top-level manager.

```c
/*
  *****************************************************
  *   File Name: t_MLM_meth.c
  *   Tivoli EIDL Compiler (Version 1.10, Date 02/15/96)
  *   generated ANSI C Tivoli method implementation File.
  *
  *   Edit this file to fill in method implementations.
  *****************************************************/

#include <samp/t_Samp.h>
#include <samp/t_Samp_imp.h>
#include <tivoli/ExException.h>

void t_imp_SAMP_MLM_mlm_in(
    SAMP_MLM _SAMP_MLM,
    Environment * _ev,
    transaction _transaction,
    SAMP_recordList* list) {
}
```

my_t_UpcallCollector_meth.c

my_t_UpcallCollector_meth.c is the other half of the gateway component that, along with upserv.c, implements the upcall method. It is the skeleton for the upcall collector.

```c
/*
  *****************************************************
  *   File Name: t_UpcallCollector_meth.c
  *   Tivoli EIDL Compiler (Version 1.10, Date 02/15/96)
  *   generated ANSI C Tivoli method implementation File.
  *
  */

#include <samp/t_Samp.h>
#include <samp/t_Samp_imp.h>
#include <tivoli/ExException.h>

void t_imp_SAMP_MLM_mlm_in(
    SAMP_MLM _SAMP_MLM,
    Environment * _ev,
    transaction _transaction,
    SAMP_recordList* list) {
}
```
8  *   Edit this file to fill in method implementations.
9  *****************************************************
10 */
11 #include <samp/t_Samp.h>
12 #include <samp/t_Samp_imp.h>
13 #include <tivoli/ExException.h>
14 #include <tivoli/TNR.h>
15 #include <tivoli/timers.h>
16 #include <tivoli/dir.h>
17 #include <tivoli/msg_supp.h>
18 #include "global.h"
19
20 long timeout = 10;
21 long queue_max = 10;
22
23 tmf_sema_t boot_mutex;
24 static int booted = 0;
25
26 /* Boot the router method. */
27 static void
28 boot (SAMP_UpcallCollector _SAMP_UpcallCollector)
29 {
30   Environment ev = {0};
31
32   tmf_P (&boot_mutex);
33   if (!booted)
34     {
35     t_SAMP_UpcallCollector_router
36       (_SAMP_UpcallCollector, &ev, Trans_none);
37     booted = 1;
38   }
39   tmf_V (&boot_mutex);
40 }
41
42 void t_imp_SAMP_UpcallCollector_set_config(
43   SAMP_UpcallCollector _SAMP_UpcallCollector,
44   Environment *_ev,
45   transaction _transaction,
46   SAMP_UpcallCollector_config_t* cfg)
47 {
48   boot (_SAMP_UpcallCollector);
if (timeout != cfg->timeout)
    timeout = cfg->timeout;

if (queue_max != cfg->queue_max)
{
    queue_max = cfg->queue_max;
    if (list_len >= queue_max)
        tmf_V (&full_sem);
}

SAMP_UpcallCollector_config_t
    t_imp_SAMP_UpcallCollector_get_config(SAMP_UpcallCollector _SAMP_UpcallCollector,
                                             Environment * _ev,
                                             transaction _transaction)
{
    SAMP_UpcallCollector_config_t config;
    config.timeout = timeout;
    config.queue_max = queue_max;
    return config;
}

void t_imp_SAMP_UpcallCollector_router(SAMP_UpcallCollector _SAMP_UpcallCollector,
                                         Environment * _ev,
                                         transaction _transaction)
{
    extern void forward_when_full ();
    MLM = dir_lookup_instance(TMF_TNR_Resource_DISTINGUISHED, "SampMLM");
    set_forward_timeout (timeout);
    forward_when_full ();
}

void t_imp_SAMP_UpcallCollector_upcall(SAMP_UpcallCollector _SAMP_UpcallCollector,
                                         Environment * _ev,
89     transaction _transaction,
90     SAMP_record* item)
91 {
92   boot (_SAMP_UpcallCollector);
93   tmf_P (&list_sem);
94   Try {
95     ex_add_unwind (tmf_V, &list_sem);
96     *tail = mg_malloc (sizeof (struct node));
97     (*tail)->next = NULL;
98     (*tail)->data = mg_malloc (sizeof (SAMP_record));
99     *(*tail)->data = *item;
100    tail = &(*tail)->next;
101    list_len++;
102    if (list_len >= queue_max)
103       tmf_V (&full_sem);
104   }
105  EndTry;
106  item->tokens._buffer = NULL;
107  item->leading_comment = NULL;
108  item->eol_comment = NULL;
109 }
110
Where Each Piece Executes

The following illustration shows where each part of the record_gen application executes.

| Server | Run tidlc here to generate files for some platforms.  
Platform-side files execute here: samp.c |
|--------|--------------------------------------------------------------------------------------------------|
| Gateway| Run ltd here to generate endpoint-side files. 
Endpoint-side files execute here: ep_meth.c, up.c |
| Endpoint|                                                                                                  |

Building the record_gen Sample Application

This section discusses how to build the record_gen application, the export directory, and the make files.

Building the Application

To build the record_gen application, follow these steps:

1. Check out the source to the directory where you want to build.
2. Change to the top-level directory.
3. Enter this command:
   make
4. The top-level Makefile (in the record_gen directory) does a make on the server side, then on the endpoint side. The top-level Makefile contains the following code:

```make
all:
  (cd platform; make)
  (cd endpoint; make)

clean:
  (cd platform; make $@)
  (cd endpoint; make $@)
```

During the build, the IDL compiler builds both sides of the application (platform and endpoint), generating many files. For details about the files created, see the Framework Services Manual.

The Export Tree

When the build completes, you find the following subdirectories in both the platform and endpoint directories:

- **export**—Directory where the entire build is merged. This directory serves as a collection point for files for all interpreter types. Created during the build.
- **interp**—One interp directory for each platform you build on; created during the build.
- **src**—Directory where you checked out the source to build.

When you run make in the record_gen build tree, it copies all targets from the build tree to the export directory. The location of the export directory is defined by the EXPORT_DIR variable in configure.mk.
Both the endpoint side and the server side contain an export directory. The figure below shows the structure of the export directory for record_gen.

- The bin directory contains the application binaries. In this case, on the server side it contains the executable samp, while on the endpoint side, it contains the executable ep_meth.
- The cfg directory contains the scripts and configuration files necessary to install the application’s classes. In this case, it contains the files Samp.cfg, Samp_ir.tar, and Samp_ist.tar. This directory exists only on the server side, not the endpoint side.
- The include directory contains the application’s public header files. This directory exists only on the server side, not the endpoint side.
- The generic directory includes shell scripts for general use of the application.

Endpoint and Platform Makefiles

Both the endpoint and the platform directories include a file called configure.mk, which is included by Makefile. configure.mk in the endpoint directory defines the LCF build area, the location of the config directory, and specifies that the IDL compiler script for
Building the record_gen Sample Application

endpoints is `ltid`. It also includes the file `defaults.mk` from the `config` directory.

```bash
# lcf endpoint build settings
#
INTERPS = solaris2 sunos4 hpux9 hpux10 aix3-r2 aix4-r1
BUILD_MACHINES = kenya gran-paradiso mansfield stout ozark goliath

LCF_BUILD = /tivoli/drm/5/francis/lcf
CONFIG_DIR = $(LCF_BUILD)/config
IMPORT_DIR = $(LCF_BUILD)/export
IMPORT_DIRS = $(IMPORT_DIR)
TOOLS_PATH = $(IMPORT_DIR)/bin/$(INTERP)/ade:/tivoli/tools/bldenv/$(INTERP)

# build system defaults
include $(CONFIG_DIR)/defaults.mk
LIBS += -lmrt -lcpl -ldes $(LIBCOMM)

# idl compiler script is ltid for lcf
IDLC=ltid

CPPFLAGS += -DDEBUG -DENDPOINT_BUILD
IDLFLAGS += -DENDPOINT_BUILD
```

Creating a Multi-Platform Environment

The Makefiles in the `record_gen` environment share many common variables, macros, and rules. These are stored in the `config` directory.

- **General defaults**—Basic variables and macros that are common to all components. These are stored in `global.mk`.
- **Platform-specific defaults**—Platform-specific information such as compiler flags, libraries, and command paths. These are stored in files named `interp.mk`.
- **Implicit build rules**—Definitions for implicit rules, such as `install`. These are contained in `rules.mk`.

Each supported platform has a rule file in the `config` directory that contains platform-specific bindings for make variables, such as command paths and compiler flags. The name of each file is `interp.mk`. 

Application Development for the Lightweight Client Framework 9–31
where interp is the TME 10 interpreter type. The table below shows interpreter types that have interp.mk files in the config directory for LCF.

<table>
<thead>
<tr>
<th>Interpreter Type</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>aix3-r2</td>
<td>AIX 3.0</td>
</tr>
<tr>
<td>aix4-r1</td>
<td>AIX 4.1</td>
</tr>
<tr>
<td>hpx9</td>
<td>HP-UX 9.0</td>
</tr>
<tr>
<td>hpx10</td>
<td>HP-UX 10</td>
</tr>
<tr>
<td>nw3</td>
<td>NetWare 3.x</td>
</tr>
<tr>
<td>nw4</td>
<td>NetWare 4.x</td>
</tr>
<tr>
<td>os2</td>
<td>OS/2</td>
</tr>
<tr>
<td>os400</td>
<td>AS/400</td>
</tr>
<tr>
<td>solaris2</td>
<td>Solaris 2</td>
</tr>
<tr>
<td>sunos4</td>
<td>SunOS 4</td>
</tr>
<tr>
<td>w32-ix86</td>
<td>Windows NT</td>
</tr>
<tr>
<td>win3x</td>
<td>Windows 3.x</td>
</tr>
<tr>
<td>win95</td>
<td>Windows 95</td>
</tr>
</tbody>
</table>
This appendix contains the manual pages for commands specific to LCF application development and for functions in libmrt. It includes manual pages for the following groups of commands and functions:

- CLI commands for dependencies: wdepset, wchdep, and wgetdeps
- Log functions
- File system input/output functions
- Miscellaneous functions, including
  - decrypt_data
  - encrypt_data
  - ep_stream_read
  - ioch_recv
  - mrt_set_method_exit_mode
  - mrt_machine_id
  - nw_echo_command_to_console
  - mrt_test_dependency
  - set_lang
  - still_alive
  - tiv_io_create
  - tiv_io_destroy
• tiv_spawn
• tiv_spawn_ui
• tiv_user_token_create
• tiv_user_token_destroy
• tiv_wait
wchdep

NAME

wchdep

PURPOSE

In the LCF, associates a dependency set with a particular method header.

SYNOPSIS

wchdep @Classes:class-name @DependencyMgr:depset method-name

DESCRIPTION

The wchdep command associates a dependency set, created with the wdepset -c command, with a particular method header.

The wchdep command is usually placed in an application’s .ist file.

ARGUMENTS

class-name  The name of the class.

depset  The label specified in the wdepset command.

method-name  The name of the method the dependency set supports.

EXAMPLE

The following example associates the dependency set hello_depset with the method named hello_meth.

wchdep @Classes:Hello @DependencyMgr:hello_depset hello_meth

ENVIRONMENT

This command is only used in the LCF.

SEE ALSO

wdepset
NAME

wdepset

PURPOSE

In the LCF, specify dependencies that a method needs to execute.

SYNOPSIS

wdepset -c label [-C class] -a tag path [ +flags ]
wdepset -d objspec
wdepset -v objspec
wdepset -r objspec

DESCRIPTION

The wdepset command specifies dependencies that a method needs so it can execute. The command has four mutually exclusive options that let you create, delete, view, or resolve a dependency set. After you specify the dependencies with the wdepset command, use the wchdep command to associate the dependency set with a method header.

When you create a dependency set, you include a tag that specifies the type of dependency (for example, bin, lib, or $INTERP).

The wdepset command is usually placed in an application’s .ist file.

ARGUMENTS

The wdepset command supports the following arguments:

- c  Create a dependency set.
- d  Delete a dependency set.
- v  View a dependency set.
- r  Resolve a dependency set.
label  The name of the dependency set.
class  The name of the class (if not DependencyMgr).
tag  One of five possible resolve tags: bin, lib, $INTERP, generic, or depset. They are defined as follows:
**bin**  A binary program. The dependency file resides in the *bin* directory of the gateway repository. The path to the file is resolved at run-time based on the target endpoint’s interpreter type.

**lib**  A shared library. The dependency file resides in the *lib* directory of the gateway repository. The path to the file and the shared library suffix are resolved at run-time based on the target endpoint’s interpreter type.

**$INTERP**  The file is downloaded only for endpoints of the type $INTERP. No path resolution is performed.

**generic**  The dependency file is downloaded for each endpoint. No path resolution is performed.

**depset**  The dependency is another dependency object.

**path**  The path to the file.

**flags**  Two values are supported:

- **+d**  Specify the directory in the endpoint cache where the dependency will be put.
- **+p**  The path (in the dependency name) being passed down is an absolute path, not a path relative to the cache.
- **+x**  Signal to the LCF endpoint to attempt to execute the dependency before receiving any remaining dependencies and before running the method.

**objspec**  The name of the dependency set.

**EXAMPLE**

The following example creates a dependency set named **hello_depset**. The example specifies that the dependency set is a library (**-a lib**) named **libfoo**:

```bash
wdepset
```
wdepset

wdepset -c hello_depset -a lib libfoo

The following example creates a dependency set named foo and uses the +d flag to tell the gateway to send the dependency (sh.exe) down to the endpoint as bin/w32ix-86/sh.exe:

wdepset -c foo w32-ix86 ../w32-ix86/tools/sh.exe +d bin/w32-ix86

ENVIRONMENT

This command is only used in the LCF.

SEE ALSO

wchdep
NAME
wgetdeps

PURPOSE
In the LCF, update all dependencies for a class or method.

SYNOPSIS
wgetdeps scoped-name

DESCRIPTION
The wgetdeps command updates all dependencies for a class or method.

scoped-name is a path to a class or method in the interface repository.

ARGUMENTS

scoped-name The name of the class or method for which to update dependencies.

EXAMPLE
The following command updates all the dependencies for the class plbo.
wgetdeps plbo

The following example pushes or updates all the dependencies for the method update_phone.
wgetdeps plbo::update_phone

ENVIRONMENT
This command is only used in the LCF.

SEE ALSO
wdepset, wchdep.
LogData

NAME

LogData

PURPOSE

Formats and logs binary data.

LIBRARY

libmrt

C SYNOPSIS

void LogData(int display_level,
             lh_p_t *logHndl,
             char *label,
             void *dataIn,
             long length)

PARAMETERS

display_level An integer indicating the message level; used for comparison against the log’s display_threshold.

logHndl A pointer (of type lh_p_t) to the log structure.

label A null-terminated character array containing a title for the output.

dataIn A pointer (of type void*) to binary data.

length The length of dataIn; a long integer.

DESCRIPTION

The LogData function is a wrapper function around LogMsg; it takes in binary data and passes HEX values to LogMsg. Use this function to output binary data to the logfile, for example, to receive or send buffers.

RETURN VALUES

None.

EXCEPTIONS

None.
NAME

LogDeinit

PURPOSE

Deallocates resources set by LogInit.

LIBRARY

libmrt

C SYNOPSIS

#include "mrt_log.h"

void LogDeinit(lh_p_t logHndl)

PARAMETERS

logHndl A handle to the log structure.

DESCRIPTION

The LogDeinit function deallocates resources set by a call to LogInit.

RETURN VALUES

None.

EXCEPTIONS

None.

SEE ALSO

LogInit
LogGetAppName

NAME

LogGetAppName

PURPOSE

Returns the value of the application identifier for a log.

LIBRARY

libmrt

C SYNOPSIS

char * LogGetAppName(lh_p_t logHndl)

PARAMETERS

logHndl A pointer (of type lh_p_t) to the log structure.

DESCRIPTION

The LogGetAppName function returns the value of the application name (appName) for the requested log. The application name is used in the log to identify which application generated a particular log message. LogGetAppName returns the appName set by LogSetAppName or by the parameter appName in LogInit().

RETURN VALUES

The return value is a null-terminated character array containing the requested log’s appName.

EXCEPTIONS

None.

SEE ALSO

LogSetAppName, LogInit.
NAME

LogGetDefault

PURPOSE

Returns a pointer to the default log structure.

LIBRARY

libmrt

C SYNOPSIS

lh_p_t LogGetDefault(void)

PARAMETERS

None.

DESCRIPTION

The LogGetDefault function returns a pointer to the default log structure.

RETURN VALUES

On successful completion, LogGetDefault returns lh_p_t, which is a void to the default pointer value.

EXCEPTIONS

None.

SEE ALSO

LogSetDefault
LogGetOutputStdout

NAME

LogGetOutputStdout

PURPOSE

Returns the value of output_stdout for a log.

LIBRARY

libmrt

C SYNOPSIS

bool_t LogGetOutputStdout(lh_p_t logHndl)

PARAMETERS

logHndl A pointer (of type lh_p_t) to the log structure.

DESCRIPTION

The LogGetOutputStdout function returns the value of the output_stdout for the log. output_stdout is a boolean; if set to TRUE, output displays to stdout as well as to the log file. If set to FALSE, output is sent only to the log file.

RETURN VALUES

LogGetOutputStdout returns a boolean that is the current output_stdout for the requested log.

EXCEPTIONS

None.

SEE ALSO

LogSetOutputStdout, LogInit
NAME

LogGetThreshold

PURPOSE

Returns the value of the display threshold for a log.

LIBRARY

libmrt

C SYNOPSIS

int LogGetThreshold(lh_p_t logHndl)

PARAMETERS

logHndl A pointer (of type lh_p_t) to the log structure.

DESCRIPTION

The LogGetThreshold function returns the value of the display_threshold for that log. The values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Log Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No logging</td>
</tr>
<tr>
<td>1</td>
<td>Minimal output</td>
</tr>
<tr>
<td>2</td>
<td>Tracing and moderate output</td>
</tr>
<tr>
<td>3</td>
<td>Data buffers and tight loops</td>
</tr>
</tbody>
</table>

RETURN VALUES

The function returns an integer that is the current display_threshold of the requested log. Values are 0, 1, 2, or 3.

EXCEPTIONS

None.
LogGetThreshold

SEE ALSO

LogSetThreshold, LogInit.
NAME
LogInit

PURPOSE
Create a new log file.

LIBRARY
libmrt

C SYNOPSIS

```
int lh_p_t LogInit(char *path,
                   char *appname,
                   int display_threshold,
                   bool_t output_to_stdout)
```

PARAMETERS

- **path**: A null-terminated character array containing the absolute path to create the log file and output messages.
- **appname**: A null-terminated character array containing the name of the calling application.
- **display_threshold**: An integer that defines the ceiling of display level to output. Any messages with a display level greater than `display_threshold` will not be output.
- **output_to_stdout**: A boolean that allows messages to be displayed on the user’s console.

DESCRIPTION

The **LogInit** function creates a new log file. The default name of the log file is `./lcfd.log`.

**LogInit** also backs up the old log file and allocates resources needed by the log module. Each time you call **LogInit**, the resources for a log file are allocated. The caller is responsible for calling the **LogDeinit** function to deallocate resources.

After you call **LogInit** to create a new log file, you may call **LogSetDefault** to establish the log as the default. Setting the log as
default allows the caller to pass LOGDEF as the log handle for logging functions, making it unnecessary to define the log handle globally.

To create additional logs, call **LogInit**, and use the returned `lh_p_t` in the desired log functions.

When a method is spawned from `lcfd`, a log resource is initialized for the method, and its output is appended to `lcfd`'s log. Use LOGDEF as the log handle for the logging functions if you want the method's logging to append to `lcfd`'s log.

**RETURN VALUES**

On successful completion, **LogInit** returns `lh_p_t`, which is a void to the log structure. NULL is returned on error. However, NULL (LOGDEF, the default log handle) is a valid value to be passed to the other log modules.

**EXCEPTIONS**

None.

**SEE ALSO**

LogDeinit, LogSetAppName, LogSetThreshold, LogSetDefault, LogSetOutputStdout.
NAME

LogMsg

PURPOSE

Formats an internationalized message and outputs it to console and log file.

LIBRARY

libmrt

C SYNOPSIS

void LogMsg(int display_level,
             lh_p_t logHndl,
             char *catalog,
             long key,
             char *msg,
             ...)

PARAMETERS

- **display_level**  
  An integer indicating the message level for comparison against the log’s display_threshold.

- **logHndl**  
  A pointer (of type *lh_p_t*) to the log structure.

- **catalog, key**  
  Output from the CatSpec, genKey and nullKey macros identifying the NLS internationalized message.

- **msg**  
  A null-terminated char array containing the format of the message.

- **...**  
  The input arguments to the format string.

DESCRIPTION

The LogMsg function uses the Tivoli National Language Support (NLS) to format an internationalized message and then outputs it to console and log file.

The message catalog entries must meet these requirements:

- Only string and integer format types are used (currently string, decimal, unsigned decimal, hex, octal).
The default message references each argument exactly once and in order.

Specific exception classes are not required.

The following example illustrates the use of LogMsg:

```
LogMsg(1, LOGDEF, genKey(UsingMethCacheDir), "Using method cache directory '%1$s/%2$s'", gcs->cache_loc, gcs->current_interp);
```

In this example:

- LOGDEF is the log initialized for you at startup.
- genKey(key) is a short-hand macro around CatSpec. It incorporates the catalog name. The key must be unique for the catalog.
- During program development, the macro ignores the catalog and key values, supplying null values instead. A null catalog tells the standard message catalog routines to use the default string rather than search an actual message catalog. A `#define` tells the CatSpec macro to start generating true file and key values. The `#define` is HAVE_CATALOG and is placed in the `make` command line when the files are rebuilt.
- The format string plays the dual role of `printf` format and default message catalog entry; there are restrictions on format types and argument order. The format string is scanned for `%` entries. Each entry results in the next vararg being processed. Note the sequential use of 1$, 2$, ... between the `%` and format type.

The LogMsg routines take special action when they encounter a string format type with the argument $errno. In this case, the string is converted to the corresponding sys_errno[] entry (or errno=<n>, if out of range).

RETURN VALUES

None.

EXCEPTIONS

None.
SEE ALSO

LogQ
LogQ

NAME

LogQ

PURPOSE

Implements a circular queue in memory of the last n messages for inclusion in exceptions. It also outputs messages to the console and/or the log file.

LIBRARY

libmrt

C SYNOPSIS

void LogQ(char *msg, ...)

PARAMETERS

msg A null-terminated character array containing the format string of the message.

... The input arguments to the format string.

DESCRIPTION

LogQ is used for debugging. Its queue can be retrieved for output with exceptions. LogQ does not support NLS or multiple log files.

The LogQ function is a wrapper function around LogMsg; it implements a circular queue in memory of the last n messages. Each message is also forwarded to the default LogMsg with a bypassKey that bypasses the NLS action. The storage is allocated and deallocated by the log module. The display_level for LogQ is hardcoded to '2.'

RETURN VALUES

None.

EXCEPTIONS

None.
SEE ALSO

LogMsg, LogQGetBuffer.
LogQueueAlloc

NAME

LogQueueAlloc

PURPOSE

Allocates the size of buffer for LogQ messages.

LIBRARY

libmrt

C SYNOPSIS

long LogQueueAlloc (long newBufSz)

PARAMETERS

newBufSz The size of the buffer, in bytes.

DESCRIPTION

The LogQueueAlloc function allocates the size of buffer for LogQ messages. You must call this function before calling any LogQ messages.

RETURN VALUES

LogQueueAlloc returns the previous size of the LogQ buffer.

EXCEPTIONS

None.

SEE ALSO

LogQ, LogQueueDealloc, LogQGetSize.
NAME

LogQueueDealloc

PURPOSE

Shuts down and deallocates the LogQ buffer.

LIBRARY

libmrt

C SYNOPSIS

void LogQueueDealloc (void)

PARAMETERS

None.

DESCRIPTION

The LogQueueDealloc function deallocates the LogQ buffer.

RETURN VALUES

None.

EXCEPTIONS

None.

SEE ALSO

LogQ, LogQueueAlloc, LogQueueGetSize.
LogQueueGetSize

NAME

LogQueueGetSize

PURPOSE

Returns the size of the LogQ buffer in bytes.

LIBRARY

libmrt

C SYNOPSIS

long* LogQueueGetSize (void)

PARAMETERS

None.

DESCRIPTION

The LogQueueGetSize function returns the size of the LogQ buffer in bytes.

RETURN VALUES

The size of the buffer, in bytes.

EXCEPTIONS

None.

SEE ALSO

LogQ, LogQueueAlloc, LogQueueDealloc.
NAME

LogQGetBuffer

PURPOSE

Returns a character array containing the circular queue of LogQ messages.

LIBRARY

libmrt

C SYNOPSIS

char * LogQGetBuffer(void)

PARAMETERS

None.

DESCRIPTION

The LogQGetBuffer function returns a character array containing the circular queue of LogQ messages.

RETURN VALUES

LogQGetBuffer returns a null-terminated character array containing the circular queue of LogQ.

EXCEPTIONS

None.

SEE ALSO

LogQ
LogSetAppName

NAME

LogSetAppName

PURPOSE

Sets an application identifier to be used in logging messages.

LIBRARY

libmrt

C SYNOPSIS

void LogSetAppName(lh_p_t logHndl, char *appName)

PARAMETERS

logHndl A pointer (of type lh_p_t) to the log structure.
appName A null-terminated character array containing the calling application’s identifier or name.

DESCRIPTION

The LogSetAppName function sets the identifier to be used in logging messages. It allows the user to set the value at times other than initialization. Several applications may have output going to one log file, and LogSetAppName provides an identifier to reflect which application generated a particular message.

RETURN VALUES

None.

EXCEPTIONS

None.

SEE ALSO

LogGetAppName, LogInit.
NAME

LogSetDefault

PURPOSE

Sets a pointer to the default log structure.

LIBRARY

libmrt

C SYNOPSIS

lh_p_t LogSetDefault(lh_p_t lh)

PARAMETERS

lh A pointer (of type lh_p_t) to the log structure.

DESCRIPTION

The LogSetDefault function maintains a static (private) pointer to the default log structure. After the pointer to the default log structure is set, all log routines using LOGDEF as the log handle will use this log.

RETURN VALUES

On successful completion, LogSetDefault returns lh_p_t, which is a void to the previous default pointer value.

EXCEPTIONS

None.

SEE ALSO

LogGetDefault
LogSetOutputStdout

NAME

LogSetOutputStdout

PURPOSE

Sets the output to go to stdout, as well as to the log file.

LIBRARY

libmrt

C SYNOPSIS

bool_t LogSetOutputStdout(lh_p_t logHndl,
                          bool_t new_output_stdout)

PARAMETERS

logHndl A pointer (of type lh_p_t) to the log structure.
new_output_stdout A boolean set to TRUE for output to also display on stdout. Set to FALSE to output only to log file.

DESCRIPTION

The LogSetOutputStdout function sets a boolean to send messages to stdout as well as to the log file. It allows the user to set the value at times other than initialization.

new_output_stdout is a boolean; if set to TRUE, output displays to stdout as well as to the log file. If set to FALSE, output is sent only to the log file.

RETURN VALUES

LogSetOutputStdout returns the boolean that was the previous output_stdout of the requested log.

EXCEPTIONS

None.

SEE ALSO

LogGetOutputStdout, LogInit.
NAME

LogSetThreshold

PURPOSE

Set the output threshold of the debug level for the log.

LIBRARY

libmrt

C SYNOPSIS

```c
int LogSetThreshold(lh_p_t logHndl,
                    int new_display_threshold)
```

PARAMETERS

- `logHndl`: A pointer (of type `lh_p_t`) to the log structure.
- `new_display_threshold`: An integer that defines the ceiling of messages to output. Any messages with a display (debug) level greater than display threshold will not be output.

DESCRIPTION

The `LogSetThreshold` function sets the output threshold of the display (debug) level of the requested log. It is usually called by `LogInit`, but it may be called directly by the user to dynamically override the initial value of `display_threshold`. It returns the value of the previous threshold.

The default `display_threshold` for a log is 0, or no output. The valid values for `display_threshold` can be set by the user in the configuration file or on the command line. They are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Log Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No logging</td>
</tr>
<tr>
<td>1</td>
<td>Minimal output</td>
</tr>
<tr>
<td>2</td>
<td>Tracing and moderate output</td>
</tr>
</tbody>
</table>
LogSetThreshold

<table>
<thead>
<tr>
<th>Value</th>
<th>Log Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Data buffers and tight loops</td>
</tr>
</tbody>
</table>

RETURN VALUES
The function returns an integer that was the previous display_threshold of the requested log.

EXCEPTIONS
None.

SEE ALSO
LogGetThreshold, LogInit.
NAME

close_ex

PURPOSE

Closes a file, specified by a file descriptor, returned by a prior call to open_ex.

LIBRARY

libmrt

C SYNOPSIS

    void write_ex(void *fdp);

PARAMETERS

    fdp            A file descriptor pointer (of type void *) to an open file.

DESCRIPTION

    The close_ex function closes the file specified by the parameter fdp. The file descriptor must be the pointer returned by a previous call to the function open_ex.

RETURN VALUES

    If the file is successfully close, close_ex returns a 1. Otherwise, an exception is thrown.

EXCEPTIONS

    Exception text: Close Failed filename, errno.

SEE ALSO

    lseek_ex.
NAME

copy_file_ex

PURPOSE

Copies the contents of a file specified by its pathname to another file.

LIBRARY

libmrt

C SYNOPSIS

long copy_file_ex(char *srcPath,
                  char *dstPath);

PARAMETERS

srcPath A null-terminated character array containing the absolute path of the current file’s name.
dstPath A null-terminated character array containing the absolute path of the file’s new name.

DESCRIPTION

The copy_file_ex function copies the contents of a file from srcPath to dstPath. Both srcPath and dstPath files are opened and the bytes are read from srcPath and written to dstPath.

RETURN VALUES

copy_file_ex returns the number of bytes copied. Exceptions generated by open_ex, read_ex, or write_ex are rethrown.

EXCEPTIONS

An error message: Copy failed: filename, errno.

SEE ALSO

open_ex, read_ex, write_ex.
NAME
does_file_exist

PURPOSE
Tests to determine if a specified file exists.

LIBRARY
libmrt

C SYNOPSIS

```c
bool_t does_file_exist(char *path);
```

PARAMETERS

- `path` A null-terminated character array containing the absolute path of the file to test.

DESCRIPTION
The `does_file_exist` function checks to determine if the file identified by the parameter `path` exists.

RETURN VALUES
The function returns TRUE if the file exists or FALSE if it does not exist.

EXCEPTIONS
None.
get_file_length_ex

NAME

get_file_length_ex

PURPOSE

Returns the number of bytes in a file.

LIBRARY

libmrt

C SYNOPSIS

long get_file_length_ex(char *path);

PARAMETERS

path A null-terminated character array containing the absolute path of the file for which you want to obtain the length.

DESCRIPTION

The get_file_length_ex function gets the length of a file specified in the pathname path.

RETURN VALUES

The get_file_length_ex function returns the number of bytes in a file.

EXCEPTIONS

Exception text: path: Stat failed: errno

SEE ALSO

get_open_file_length_ex
NAME

get_open_file_length_ex

PURPOSE

Gets the length in bytes of a file opened by open_ex.

LIBRARY

libmrt

C SYNOPSIS

long get_open_file_length_ex(void *fdp);

PARAMETERS

fdp A file descriptor point (of type void *) to an open file.

DESCRIPTION

The get_open_file_length_ex function gets the length of a file, specified by fdp, opened with open_ex.

RETURN VALUES

The get_open_file_length_ex function returns the length of the file, in bytes. If there is an error, an exception is thrown.

EXCEPTIONS

Exception text: path: fstat failed: errno.

SEE ALSO

get_file_length_ex
NAME
lseek_ex

PURPOSE
Move to a new position in a file opened by open_ex for unbuffered and unformatted I/O.

LIBRARY
libmrt

C SYNOPSIS

long lseek_ex(void *fdp,
              long offset,
              int origin);

PARAMETERS

fdp A file descriptor pointer (of type void *) to an open file.
offset Offset of new position, in bytes, from origin.
origin Constant indicating the position from which to offset.

DESCRIPTION

The lseek_ex function sets the current read or write position of the file specified by the parameter fdp to a new value indicated by the parameters offset and origin. The offset is a long integer indicating how far way the new position is from a specific location given in origin. The origin, defined in <stdio.h>, must be one of the following:

- SEEK_SET—Beginning of file.
- SEEK_CUR—Current position in the file.
- SEEK_END—End of file.

The file position may be set anywhere in the file except before the beginning of the file.
RETURN VALUES

If successful, `lseek_ex` returns the offset from the beginning of the file. Otherwise, an exception is thrown.

EXCEPTIONS

Exception text: `filename`: Seek failed: `errno`.

SEE ALSO

`open_ex`, `read_ex`. 
NAME

make_path

PURPOSE

Creates all the directories in the specified path.

LIBRARY

libmrt

C SYNOPSIS

void make_path(char *path,
               bool_t skiplast);

PARAMETERS

path
A null-terminated character array containing the absolute path of the directories to create.

skiplast
A boolean value that flags whether or not to create the last item specified in the path.

DESCRIPTION

The make_path function creates new directories with the pathname path. make_path parses the path, creating each of the directories. This function also corrects the path for operating system-specific needs using cpl_correct_path. make_path is dependent on the C function mkdir not failing the directory exists.

RETURN VALUES

None. Throws an exception if an error occurs.

EXCEPTIONS

Exception text: path: Make path failed: errno.
Exception text: path cpl_correct_path failed.

SEE ALSO

cpl_correct_path.
NAME

mkdir_ex

PURPOSE

Creates a new directory with a specified path.

LIBRARY

libmrt

C SYNOPSIS

void mkdir_ex(char *path);

PARAMETERS

path

DESCRIPTION

The mkdir_ex function creates a new directory with the pathname path. The pathname can include drive specification and directory or subdirectory names, but because mkdir_ex can only create one directory at a time, all but the last subdirectory must already exist.

RETURN VALUES

When mkdir_ex succeeds in creating the directory, it returns a 0. Otherwise, it throws an exception.

EXCEPTIONS


SEE ALSO

make_path
open_ex

NAME

open_ex

PURPOSE

Opens a file for low-level I/O.

LIBRARY

libmrt

C SYNOPSIS

```c
void *open_ex(char *path,
               int flags,
               int mode);
```

PARAMETERS

- **path**: A null-terminated character array containing the absolute path of the file to open.
- **flags**: An integer formed from the bitwise OR of the constants (found in `fcntl.h`):
  - `O_APPEND`
  - `O_BINARY`
  - `O_CREAT`
  - `O_EXCL`
  - `ORDONLY`
  - `O_RDWR`
  - `O_TRUNC`
  - `O_WRONLY`

- **mode**: An integer formed from the bitwise OR of the constants (found in `<sys/stat.h>`):
  - (For UNIX platforms):
    - `S_IRUSR`
    - `S_IWUSR`
    - `S_IRGRP`
    - `S_IWGRP`
    - `S_IROTH`
    - `S_IWOTH`
  - For PCs:
    - `S_IWRITE` | `S_IREAD`

DESCRIPTION

The **open_ex** function opens the file specified in the parameter **path**. The types of operations you intend to perform on the file after it is opened are indicated by the parameter **flags**. The third parameter, **mode**, is only needed when you request the creation of a new file by using the flag **O_CREAT**. If the file does not exist, the value of **mode** is used to set the permission of the newly created file.
**RETURN VALUES**

If successful, `open_ex` returns a file descriptor pointer (of type `void *`) to an open file. Otherwise, an exception is thrown.

**EXCEPTIONS**

An error message: Open failed: `filename`, `errno`.

**SEE ALSO**

`close_ex`, `read_ex`, `write_ex`, `lseek_ex`, `get_open_file_length_ex`. 
NAME

read_ex

PURPOSE

Retrieve a specified number of bytes of data, without any formatting, from the current position in a file that has been opened for unformatted I/O by open_ex.

LIBRARY

libmrt

C SYNOPSIS

```c
int read_ex(void *fdp,
    char *data,
    int datalen);
```

PARAMETERS

- **fdp**: A file descriptor pointer (of type void *) to an open file.
- **data**: A pointer (of type char *) to a buffer into which data will be copied.
- **datalen**: Number of bytes to be read.

DESCRIPTION

The `read_ex` function copies the number of bytes specified in the parameter `datalen` from the file whose file descriptor is in the parameter `fdp` to the array of characters, `data`.

RETURN VALUES

The `read_ex` function returns the number of bytes actually read from the file. In case of an error, `read_ex` returns -1 and throws an exception.

EXCEPTIONS

An error message: Read failed: *filename*, errno.
SEE ALSO

open_ex, write_ex.
write_ex

NAME

write_ex

PURPOSE

Saves a specified number of bytes of data, without any formatting, at the current position in a file opened for unformatted I/O by open_ex.

LIBRARY

libmrt

C SYNOPSIS

```c
void write_ex(void *fdp,
             char *data,
             int datalen);
```

PARAMETERS

- **fdp** A file descriptor pointer (of type `void *`) to an open file.
- **data** A pointer (of type `char *`) to a buffer representing data to be written.
- **datalen** Number of bytes to be written.

DESCRIPTION

In a file opened for writing or appending by a call to open_ex, the write_ex function copies the number of bytes specified in the parameter datalen from the array data to the file whose file descriptor is in the parameter fdp.

The writing of data begins at the current position in the file and the current position is appropriately incremented after writing. If the file is opened for appending, the writing will always take place at the end of the file.

RETURN VALUES

The write_ex function returns the number of bytes actually written to the file.
EXCEPTIONS

Exception text: Write failed: filename, errno.
Short write filename, errno.

SEE ALSO

open_ex, read_ex.
rename_file_ex

NAME

rename_file_ex

PURPOSE

Changes the name of a file specified by its pathname.

LIBRARY

libmrt

C SYNOPSIS

void rename_file_ex(char *srcPath,
                    char *dstPath);

PARAMETERS

srcPath         A null-terminated character array containing the
                absolute path of the current file’s name.

dstPath         A null-terminated character array containing the
                absolute path of the file’s new name.

DESCRIPTION

The rename_file_ex function changes the name of a file from srcPath to dstPath.

RETURN VALUES

None. Throws an exception on error.

EXCEPTIONS

Exception text: Rename srcPath to dstPath failed: errno
decrypt_data

NAME
decrypt_data

PURPOSE
Transforms cipher text to the original plain text.

LIBRARY
libmrt

C SYNOPSIS

void decrypt_data(encrypt_mech_t encrypt_type,
void *key,
char **ciphertext,
long *ciphertext_len
char *plaintext,
long plaintext_len);

PARAMETERS

encrypt_type Is one of the following:

**ENCRYPT_NONE**
No encryption performed. When the call completes, the cipher text is the same as the plain text.

**ENCRYPT_XOR**
Simple (exclusive OR) algorithm.

**ENCRYPT_DES_KRB4**
56-bit DES encryption.

**ENCRYPT_MD5**
Cipher text is plain text with an md5 hash of the key and plain text appended. If the md5 hash does not match the data, an exception is generated.

key The encryption key used by the encryption algorithm in encrypt_type.

ciphertext The encrypted data to be decrypted.

ciphertext_len The length (in bytes) of ciphertext.
decrypt_data

plaintext  The resulting clear text.
plaintext_len The length (in bytes) of plaintext.

DESCRIPTION

Given the correct key and encryption algorithm, decrypt_data converts cipher text to plain text. It is used for decrypting application data within LCF.

RETURN VALUES

If the md5 hash does not match the data, an exception is generated.
encrypt_data

NAME
encrypt_data

PURPOSE
Converts plain text to cipher text.

LIBRARY
libmrt

C SYNOPSIS

```c
void encrypt_data(encrypt_mech_t encrypt_type,
                  void *key,
                  char *plaintext,
                  long plaintext_len,
                  char **ciphertext,
                  long *ciphertext_len);
```

PARAMETERS

- `encrypt_type` is one of the following:
  - **ENCRIPT_NONE**
    - No encryption performed. When the call completes, the cipher text is the same as the plain text.
  - **ENCRIPT_XOR**
    - Simple (exclusive OR) algorithm.
  - **ENCRIPT_DES_KRB4**
    - 56-bit DES encryption.
  - **ENCRIPT_MD5**
    - Cipher text is plain text with an md5 hash of the key and plain text appended.

- `key` The encryption key used by the encryption algorithm in `encrypt_type`.
- `plaintext` The data to be encrypted.
- `plaintext_len` The length (in bytes) of `plaintext`.
- `ciphertext` The resulting encrypted text.
encrypt_data

\[ \text{ciphertext_len} \quad \text{The length of ciphertext.} \]

**DESCRIPTION**

Given a key and a choice of encryption algorithms, `encrypt_data` converts plain text to cipher text. It is used for encrypting data within the LCF.

**RETURN VALUES**

None.
ep_stream_read

NAME
ep_stream_read

PURPOSE
Reads an MDist stream.

LIBRARY
libmrt

C SYNOPSIS

glong ep_stream_read(char **datap);

PARAMETERS

datap  Pointer to the data stream.

DESCRIPTION

dep_stream_read reads an MDist stream and returns the number of bytes read. If the stream is non-NULL, it allocates a pointer and puts it in datap.

RETURN VALUES

Returns 0 if the stream is empty. Otherwise, returns the number of bytes in datap.
ioch_recv

NAME

ioch_recv

PURPOSE

Reads bytes from an existing channel.

LIBRARY

libmrt

C SYNOPSIS

```c
long ioch_recv(io_chan_t * ip,
              char **datap,
              long maxwant)
```

PARAMETERS

- `maxwant` Maximum number of bytes to read from the channel.
- `datap` Pointer to the data stream.
- `ip` Handle for the channel.

DESCRIPTION

`ioch_recv` reads bytes from an existing channel. You specify the maximum number of bytes to read with the `maxwant` parameter. A `maxwant` of 0 is considered infinite.

RETURN VALUES

Returns the actual count or EOF.
NAME

mrt_set_method_exit_mode

PURPOSE

Sets the exit mode for a method.

LIBRARY

libmrt

C SYNOPSIS

```c
int mrt_set_method_exit_mode(unsigned int mode)
```

PARAMETERS

- `mode`: Exit mode for the method. The supported modes are:
  - METH_EXIT_REBOOT_MACHINE
    - Reboot the machine.
  - METH_EXIT_RESTART_OS
    - Restart the operating system.

DESCRIPTION

Use `mrt_set_method_exit_mode` to set the exit mode for a method. When the method completes, the appropriate action, either restarting or rebooting, is taken just prior to exiting the method’s `main()`.

Be aware that using the reboot modes are catastrophic to other applications. Currently, `mrt_set_method_exit_mode` does no synchronization with the completion of other methods. I.e., the action will take place independent of the condition or status of other methods.

NOTE

For NetWare, the interpretation of REBOOT and RESTART are identical. In the case of Windows, the REBOOT option will reboot the machine, and the RESTART option will restart the operating system only.
mrt_set_method_exit_mode

RETURN VALUES

Returns 0 on success or a non-zero value on error.
NAME

mrt_machine_id

PURPOSE

Returns the unique ID of the machine.

LIBRARY

libmrt

C SYNOPSIS

int mrt_machine_id(unsigned char *id)

PARAMETERS

id Unique ID of the machine.

DESCRIPTION

mrt_machine_id fills your string buffer with the unique ID of the machine. This ID is generated in different depending on the interpreter type. For example, for PCs (NetWare, Windows, NT, OS/2), this value is uniquely generated by lcfd. Note that the machine ID will not exceed 128 characters in length; therefore, the caller should allocate at least 128 bytes in the return buffer.

RETURN VALUES

Returns 0 on success and a non-zero value on error.
mrt_check_dependency

NAME

mrt_check_dependency

PURPOSE

Checks a dependency state.

LIBRARY

libmrt

C SYNOPSIS

int mrt_check_dependency (char *dependency)

PARAMETERS

dependency    The dependency name.

DESCRIPTION

mrt_check_dependency tests to determine if the dependencies for a method have been updated. This function is intended for cases where a long-running method, such as a daemon, needs to determine if a dependency has been updated and, if it has, kill the old process and start a new one.

RETURN VALUES

Returns zero if the dependency is unchanged and 1 if the dependency has been updated. Returns less than zero on error.
nw_echo_command_to_console

NAME
	nw_echo_command_to_console

PURPOSE

On NetWare only, sends a command to the console.

LIBRARY

libmrt

C SYNOPSIS

nw_echo_command_to_console(char *str)

PARAMETERS

str The command string to send to the console.

DESCRIPTION

On NetWare platforms, enables you to send a command string to the console.

NOTE

This function is for NetWare only.

RETURN VALUES

Returns 0 on success and non-zero on error.
set_lang

NAME

set_lang

PURPOSE

Sets the language the gateway uses to bind messages.

LIBRARY

libmrt

C SYNOPSIS

void set_lang(char *lang)

PARAMETERS

lang Language.

DESCRIPTION

The set_lang function sets the language the endpoint gateway uses when it binds messages.

RETURN VALUES

None.

EXCEPTIONS

None.
stilled

NAME

still_alive

PURPOSE

Reinitializes timer to keep the gateway from timing out the session.

LIBRARY

libmrt

C SYNOPSIS

void still_alive(void *ipc, unsigned char status)

PARAMETERS

ipc
IPC handle.

status
Status field (for application use).

DESCRIPTION

When the endpoint gateway is waiting to receive data (for example, the results from a downcall to an endpoint), by default it times out after 10 minutes. still_alive sends a packet to the gateway to reinitialize the timer on the gateway and maintain the connection to the endpoint.

The length of the default timeout on the gateway is configurable.

RETURN VALUES

None.
tiv_io_create

NAME

tiv_io_create

PURPOSE

Creates a handle to the stdin, stdout, and stderr for the process to be launched.

LIBRARY

libmrt

C SYNOPSIS

```c
void* tiv_io_create(io_type_t in_type,
                     void* in,
                     io_type_t out_type,
                     void* out,
                     io_type_t err_type,
                     void* err);
```

PARAMETERS

- `in_type`: A value for `io_type_t` the caller passes in as a handle to stdin.
- `in`: Stdin.
- `out_type`: A value of for `io_type_t` the caller passes in as a handle to stdout.
- `out`: Stdout.
- `err_type`: A value of for `io_type_t` the caller passes in as a handle to stderr.
- `err`: Stderr.

DESCRIPTION

`tiv_io_create` creates an array that can be passed to `tiv_spawn` to represent the handles for stdin, stdout, and stderr. The caller may pass in one of the types specified in `io_type_t`. The supported types are:

- `IO_UNUSED`—stdin, stdout, and stderr are unused.

A–60

Version 3.2
tiv_io_create

- IO_STDIO—stdin, stdout, and stderr is FILE.
- IO_FILEDESC—stdin, stdout, and stderr is int.
- IO_FILENAME—stdin, stdout, and stderr is char*
- IO_DEFAULT—The implementation chooses the default. The caller is not supplying a specific value for in/out/err and allows the implementation to provide a default value for the requested resource.
- IO_NATIVE—Platform-specific stdin, stdout, and stderr.

RETURN VALUES

If tiv_io_create returns NULL, this indicates the specified input type is not supported, one of the arguments was invalid, or the combination of inputs is not supported.

If it returns non-NULL void *, this is an opaque pointer that can be passed as is to tiv_spawn_ui() or tiv_create_process(). The caller must free the pointer when done.

SEE ALSO

tiv_io_destroy
NAME

tiv_io_destroy

PURPOSE

Frees the pointer allocated by a call to tiv_io_create().

LIBRARY

libmrt

C SYNOPSIS

void tiv_io_destroy(void *io_ptr,
                    int do_close);

PARAMETERS

io_ptr Pointer to files to close.
do_close Flag to close file descriptors.

DESCRIPTION

  tiv_io_destroy frees the pointer allocated by a call to tiv_io_create. In
  addition to destroying the handle, you can optionally, you can close file
  descriptors (represented by io_ptr) that may have been opened. This
  implementation will not close any files related to stdin, stdout, or
  stderr.

RETURN VALUES

  None.

SEE ALSO

  tiv_io_create
tiv_spawn

NAME
tiv_spawn

PURPOSE
Launches a process.

LIBRARY
libmrt

C SYNOPSIS

```c
int tiv_spawn(char* exec_path,
              char** argv,
              char** envp,
              void* std_handles,
              TIV_USER_T user_info);
```

PARAMETERS

- **exec_path**
  If NULL, argv[0] is used. Otherwise, exec_path is the file to execute. If it is a relative path, PATH is searched. If the file is not found, platform-specific file name extensions for executables are tried.

- **argv**
  An array of arguments to be passed to the new process. Arguments must be NULL-terminated.

- **envp**
  An array of arguments that specify the environment for the process. If NULL, the new process receives the environment of the calling process. Otherwise, envp specifies the environment for the new process. It must be NULL-terminated.

- **std_handles**
  If NULL, the new process inherits standard input, output, and error handle from the calling process. Otherwise, std_handles[0], std_handles[1], and std_handles[2] contain standard input, output, and error handles (created by tiv_io_create) for the new process.

- **user_info**
  If NULL, the new process runs as the same user as the calling process. Otherwise, the new process runs as the user (returned by tiv_user_token_create)
tiv_spawn

specified in user_info. On UNIX, you pass a TIV_USER_T containing uid/gid settings. On
WIN32, you pass a token created by calling TAP, LogonUser(), or by extracting the user token from the
current process.

DESCRIPTION

tiv_spawn is used to start a process on UNIX and PC platforms.

RETURN VALUES

On success, tiv_spawn returns the process ID of the newly created process (for all but WIN32 platforms). For WIN32, it returns the handle to the newly created process. You must call CloseHandle to close it when you are done.

    tiv_spawn returns -1 on failure.

SEE ALSO

tiv_wait, tiv_io_create, tiv_spawn_ui, tiv_user_token_create
NAME

tiv_spawn_ui

PURPOSE

Launches a GUI-based command on a desktop other than the default desktop of the current process.

LIBRARY

libmrt

C SYNOPSIS

extern int tiv_spawn_ui(
    char* exec_path,
    char** argv,
    char** envp,
    void* std_handles,
    TIV_USER_T user_info);

PARAMETERS

exec_path
If NULL, argv[0] is used. Otherwise, the file to execute. It may be a relative path, in which case PATH is searched. If the file is not found, platform-specific file name extensions for executables are tried.

argv
An array of arguments to be passed to the new process. Arguments must be NULL-terminated.

envp
An array of arguments that represent the environment. If NULL, the new process receives the environment of the calling process. Otherwise, the environment for the new process. Must be NULL-terminated.

std_handles
If NULL, the new process inherits standard input, output, and error handle from the calling process. Otherwise, std_handles[0], std_handles[1], and std_handles[2] contain standard input, output, and error handles (created by tiv_io_create) for the new process.

user_info
If NULL, the new process runs as the same user as the calling process. Otherwise, the new process runs as
tiv_spawn_ui

the user (returned by tiv_user_token_create) in user_info.

DESCRIPTION

tiv_spawn_ui launches a GUI-based command on a desktop other than the default desktop of the current process. The desktop used is the active input desktop (currently visible to the logged on user).

The user token must be on the ACL of the desktop where the command is to be executed; otherwise, the launch will fail.

Other than the above requirement, this function behaves the same as a call to tiv_spawn(). The process to be launched should be a windows-based command, as the overhead for using this function is higher than simply calling tiv_spawn().

RETURN VALUES

On success, tiv_spawn_ui returns the process ID of the newly created process (for all but WIN32 platforms). For WIN32, it returns the handle to the newly created process. You must call CloseHandle to close it when you are done.

tiv_spawn_ui returns -1 on failure.

SEE ALSO

tiv_spawn
tiv_user_token_create

NAME
tiv_user_token_create

PURPOSE
Allocates a pointer to a token that contains user information for use by tiv_spawn.

LIBRARY
libmrt

C SYNOPSIS

TIV_USER_T tiv_user_token_create(char* user,
                   char* group,
                   char* password);

PARAMETERS

user User name.
group Group name.
password Password of user account.

DESCRIPTION

tiv_user_token_create allocates a pointer to a token that contains information used by tiv_spawn() for launching processes in the context of another user (setid).

On UNIX platforms, user and group are used; password is ignored
On NT platforms, user is used, password is optional; group is ignored
On NetWare platforms, user and password are used; group is ignored
When the token is no longer needed, use tiv_user_token_destroy to destroy a token.

RETURN VALUES

On success, returns a pointer to a valid token. On failure, returns NULL.
tiv_user_token_create

SEE ALSO

tiv_user_token_destroy, tiv_spawn, tiv_spawn_ui
NAME

tiv_user_token_destroy

PURPOSE

Deallocates a token created by calling tiv_user_token_create.

LIBRARY

libmrt

C SYNOPSIS

void tiv_user_token_destroy(TIV_USER_T token);

PARAMETERS

token	Token to be destroyed.

DESCRIPTION

tiv_user_token_destroy destroys the token created by the call to tiv_user_token_create.

RETURN VALUES

None.

SEE ALSO

tiv_user_token_create, tiv_spawn, tiv_spawn_ui
tiv_wait

NAME

tiv_wait

PURPOSE

Waits for a process launched asynchronously.

LIBRARY

libmrt

C SYNOPSIS

```c
int tiv_wait (int pid,
              int* status,
              int flags_or_timeout);
```

PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pid</code></td>
<td>Process ID.</td>
</tr>
<tr>
<td><code>status</code></td>
<td>(Return parameter) Status of the process.</td>
</tr>
<tr>
<td><code>flags_or_timeout</code></td>
<td>Varies according to architecture; can be flag information or a value for timeout.</td>
</tr>
</tbody>
</table>

DESCRIPTION

`tiv_wait` waits for a process launched asynchronously. Implementations that support asynchronously spawning can use this call.

`tiv_wait` is similar to the UNIX `waitpid` function. When you call `tiv_wait`, furnish the `pid` to wait for, and check for the exit code. `status` is returned in the second parameter. The following “W” macros for WIN32 are implemented to check for exit code and status (if supported):

- **WIFSTOPPED**(status)
  This macro evaluates to a non-zero value if status was returned for a child process that is currently stopped.
WSTOPSIG(status)
If the value of WIFSTOPPED(status) is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

WIFSIGNALED(status)
This macro evaluates to a non-zero value if status was returned for a child process that terminated because it received a signal.

WTERMSIG(status)
If the value of WIFSIGNALED(status) is non-zero, this macro evaluates to the number of the signal that caused the child process to terminate.

WCOREDUMP(status)
If the value of WIFSIGNALED(status) is non-zero, this macro evaluates to a non-zero value if a core image of the terminated child process was created.

WIFEXITED(status)
This macro evaluates to a non-zero value if the status was returned for a child process that terminated normally.

WEXITSTATUS(status)
If the value of WIFEXITED(status) is non-zero, this macro evaluates to the exit code the child process passed to _exit() or the value the child process returned from the main function.

On varying architectures, you can use the third argument to pass flag information or a timeout value for the underlying API calls. The varying architectures interpret the third argument as follows:

NW A timeout in units of seconds.
NT A timeout in units of seconds.
WIN95 A timeout in units of seconds.
WIN32S flags_or_timeouts is unused.
UNIX flags_or_timeouts is the third argument to waitpid().
tiv_wait

In the case where a timeout can be specified, the following conventions apply:

- timeout = 0  Do not wait; check status and return immediately.
- timeout > 0  Time out after specified number of seconds.
- timeout < 0  Do not time out; wait until process is done.

RETURN VALUES

A return value of -1 indicates an error in wait.
A return value of -2 indicates a timeout on PC platforms. This is not an error.
A non-zero that is the same as the pid indicates the wait was successful.

EXCEPTIONS

None.

SEE ALSO

- tiv_spawn
Glossary

This glossary contains terms related to application development for the lightweight client framework. It also includes platform-specific terms used in this guide.

C

cache

See method cache.

common porting layer (CPL)

A set of functions that implement functions found on some but not all platforms.

Common Object Request Broker Architecture (CORBA)

Common Object Request Broker Architecture. A specification produced by the Object Management Group (OMG) that presents standards for various types of object request brokers (such as client-resident ORBs, server-based ORBs, system-based ORBs, and library-based ORBs). Implementation of CORBA standards enables object request brokers from different software vendors to interoperate.

D

dependencies

A condition that must be met for a method to execute. Dependencies are libraries or files that support a method and must be present for the method to execute. In the LCF, you specify dependencies with the wdepset command.

downcall

A method invocation from the TMR server or gateway “down” to an endpoint.
**endpoint**
A client system (UNIX workstation or PC) running the LCF and that is not involved in the daily operations of managing a network; instead, it is used for desktop operations. An endpoint communicates only with its assigned gateway.

**endpoint application**
An application that runs on an endpoint. Endpoint methods use some of the tools and services of the TME 10 ADE, but they are implemented using a special application mini-runtime library (**libmrt**) specific to LCF.

**endpoint list**
A list maintained by the endpoint manager on the TMR server. It contains all information necessary to uniquely identify and manage endpoints, including which gateway the endpoint is using.

**endpoint manager**
A service that runs on the TMR server. It controls and configures both gateways and endpoints. It assigns endpoints to gateways and maintains the endpoint list.

**endpoint method**
A method that runs on an endpoint as the result of a request from other managed resources in the TMR. Results of the method are forwarded to the gateway and then to the call managed resource. When an endpoint method is invoked on an endpoint, the endpoint uses the method executable store in its method cache. If the method is not in the cache or is not the most current version, the gateway passes the correct method to the endpoint.
File allocation table (FAT)

File allocation table. A file system type, supported on Windows NT and other PC operating systems, that requires files to conform to the 8.3 naming convention.

Full framework

The full TME 10 Framework, including the full ADE services.

gateway

A service running on a managed node that provides all communications between a group of endpoints and the rest of the TME 10 environment. The gateway also has the MDist repeater functionality built in, enabling it to act as the fan-out point for distributions to a very large number of endpoints.

gateway method

A method that runs on the managed node associated with the gateway of the endpoint referred to in the object reference. A method that runs on the gateway’s proxy managed node on behalf of an endpoint. Results of the method are forwarded to the calling managed resource.

gateway proxy managed node

The managed node on which a gateway runs.

gateway repository

The repository, located on the managed node associated with the gateway, used to store methods and their dependencies for downloading to endpoints.
lightweight client framework

An enhancement to the TME 10 Framework that extends the framework to large numbers of dataless endpoints. It includes the endpoint manager, the gateway, and large numbers of endpoints, each of which communicates with the TMR through a gateway.

LCF

*See* lightweight client framework.

LCF application

An application that runs on an endpoint, within the LCF environment.

managed node

In the TME 10 environment, any managed resource on which the TME 10 Framework is installed.

MDist

Multiplexed distribution. In the TME 10 environment, a service that enables efficient distribution of large amounts of data across complex networks.

method

In object-oriented design or programming, the software that implements the behavior specified by an operation.

method body

The binary program or script that contains the method entry point.

method cache

On an endpoint, a cache used to store methods to execute on the endpoint. The size of the cache is limited to 20.5MB. When the cache reaches its size limit, methods are removed from it on a least-recently-used basis.
**Mid-level Manager (MLM)**
A function that performs a subset of systems and network management tasks, off-loading these tasks from the top-level manager.

**Microsoft Visual C compiler (MSVC)**
C compiler for the Win32 platform.

**multiplexed distribution (MDist)**
*See* MDist.

**object dispatcher**
In the TME 10 environment, synonym for object request broker.

**object reference**
In the TME 10 environment, the object identifier (OID) given to an object during its creation.

**OID**
Object identifier.

**oserv**
The name of the CORBA-compliant object request broker used by the TME 10 environment. Oserv runs on the TMR server and each of its clients.

**per-upcall authorization**
A type of authorization performed by applications that have short-lived methods that need to perform only one upcall.

**profile**
In the TME 10 environment, a container for application-specific information about a particular type of resource. A TME 10 application specifies the template for its profiles; the template includes
information about the resources that can be managed by that TME 10 application. Following are some examples of profiles:

- In TME 10 Distributed Monitoring, a monitor
- In TME 10 Net.Commander, a host namespace profile
- In TME 10 Software Distribution, a file package
- In TME 10 User Administration, a user or group profile

A profile is created in the context of a profile manager; the profile manager links a profile to the TME 10 resource (for example, a managed node) that uses the information contained in the profile. A profile does not have any direct subscribers.

**profile manager**

In the TME 10 environment, a container for profiles that links the profiles to a set of resources, called “subscribers.” Subscribers can be managed nodes or other profile managers. A profile manager can contain (a) profiles of multiple types or (b) multiple profiles of the same type. TME 10 administrators use profile managers to organize and distribute profiles. A profile manager is created in the context of a policy region and is a managed resource in a policy region. See also subscription list.

**profile manager (dataless mode)**

In the LCF, there is no database on an endpoint, as there is on a managed node. To distribute to endpoints, the profile manager operates in dataless mode. Data is sent to the application on the endpoints, but it is not written to a database on the endpoint.

There is an additional attribute you set so that the profile manager runs in dataless mode. When you set the attribute so that a profile manager runs in a dataless mode, the profile manager writes the data only to the application.

Note that the profile manager is the same for endpoint applications and all other applications. The only difference is whether or not this one attribute is set—when it is not, the profile managers are exactly the same.
pull
In the TME 10 environment, an operation (for example, a TME 10 UserLink file-package request) that initiates an action by requesting it of a resource.

push
In the TME 10 environment, an operation (for example, a file package distribution) that sends information to other resources.

S

server
A system that furnishes services to multiple clients.

subscription list
In the TME 10 environment, a list that identifies the managed resources that are subscribed to a profile manager. These managed resources (which can include other profile managers) are the recipients of profile distributions. Including a profile manager on a subscription list (in effect, a list within a list) is a way of subscribing several managed resources simultaneously rather than adding each one individually. In TME 10 Plus modules, a profile manager functions as a subscription list.

T

Top-Level Manager (TLM)
In an application, the highest level that processes application data.

TME 10 ADE
TME 10 Application Development Environment. A TME 10 toolkit that contains the complete application programming interface (API) for the TME 10 Framework. This toolkit enables customers and Tivoli Partners to develop their own applications for the TME 10 environment.
TME 10 Framework

The base software that is required to run any of the TME 10 systems management applications. This software infrastructure enables the integration of systems management applications from Tivoli and the Tivoli Partners. The Framework includes the following:

- Object request broker (oserv)
- Distributed object database
- Basic administration functions
- Basic application services
- Basic desktop services such as the graphical user interface (GUI)

In a TME 10 environment, the TME 10 Framework is installed on every client and every server with these exceptions:

- The TME 10 Framework is never installed on a client PC; rather, the PC agent is installed on the PC.
- The TMR server is the only server that contains the full object database.

Tivoli Management Region (TMR)

In the TME 10 environment, a TMR server and the set of clients that it serves. An organization can have more than one TMR. The use of interconnected TMRs is transparent to the user.

Note: A TMR addresses the physical layout of resources, and a policy region addresses the logical organization of resources.

TMR

See Tivoli Management Region (TMR).

TMR client

In a TME 10 environment, any computer—except the TMR server—on which the TME 10 Framework is installed. A TMR client runs the oserv and maintains a local object database.
TMR server

In a TME 10 environment, the server that holds or references the complete set of TME 10 software, including the full object database. Contrast with TMR client.

UNC

Uniform naming convention. In Windows NT and other PC operating systems, names prefixed with a \ indicate resources located on a remote system.

upcall

Indicates a method invocation from an endpoint “up” to the gateway.

upcall collector

A service that acts as a collection point for upcalls from endpoints. The upcall collector groups upcall requests before forwarding them to the TMR server.
Index

A
ADR routines 4-22
adrgen 3-2, 4-23
application development 3-1
application development for LCF 2-1

B
binary tree search routines 5-2

C
C data types 4-23
Catch 4-4
CatchAll 4-4
CCMS 7-1
  differences between LCF and full framework 7-3
  header files 7-10
  LCF applications 7-1
  profile managers 7-4
CCMS functions 7-2
CCMS in LCF 7-3
CCMS library 7-10
ccms_find_record_item 7-2
ccms_read_next_record 7-2
ccms_read_pushed_records 7-2
ccms_read_some_records 7-3
ccms_read_update_parameters 7-2
close_ex 4-10, A-31
common porting layer 5-1
compound messages 4-6
copy_file_ex 4-10, A-32
cpl_correct_path 5-2
cpl_fclose 5-7
cpl_fflush 5-3
cpl_fopen 5-7
cpl_fprintf 5-7
cpl_get_current_location 5-3
cpl_gete 5-8
cpl_getcwd 5-4
cpl_gethostname 5-6
cpl_getopt 5-5
cpl_getpass 5-5
cpl_gettimeofday 5-5
cpl_ltoa 5-6
cpl_putenv 5-4
cpl_register_print_callback 5-8
cpl_register_thread_yield_callback 5-9

cpl_THREADyield 5-11
cpl_tmpdir 5-11
cpl_tmpfile 5-10
cpl_tmpnam 5-10
creating data types 4-23

data types
  creating 4-23
  Task Library 8-1
data types in libmrt 4-23
depending 3-3
  lcf 3-3
  wdebug 3-3
decrypt_data 4-27
dependencies
  creating 6-3
  defined 6-1
  deleting 6-4

Application Development for the Lightweight Client Framework  Index–1
DependencyMgr object 6-2
endpoint 6-1
examples 6-7
implementing 6-2
introduction 6-1
out-of-cache 6-5
resolve tags 6-2
resolving 6-4
specifying 6-2
viewing 6-4
dirent functions 5-4
does_file_exist 4-10, A-33

F
file input and output 4-10

G
gateway repository 6-1
get_file_length 4-10
get_file_length_ex A-34
get_file_size 4-10
get_open_file_length_ex A-35
getenv 5-4

E
encrypt_data 4-27
endpoint 1-2
endpoint gateway 1-2
endpoint manager 1-2
endpoint method 2-3
EndTry 4-4
ep_stream_read 4-10, 4-27, 4-29, A-51
ev_to_exception 4-4
exception_to_ev 4-4
exceptions 4-4
  Catch 4-4
  CatchAll 4-4
  EndTry 4-4
  ev_to_exception 4-4
  exception_to_ev 4-4
  ReThrow 4-4
  Throw 4-4
  Try 4-4
  Try/Catch macros 4-4
  variable argument 4-4

I
ioch_recv 4-27, A-52

L
launching processes 4-25
LCF
  application development 2-1
  CCMS 7-1
  CCMS functions 7-2
  file input and output 4-10
  IDL files 3-1
  methods 2-3
  overview 1-1
  programming environment 2-1, 3-1
  sequence routines 4-8
  Task library 8-1
LCF application development 2-15, 3-1
LCF applications
  dependencies 6-1
LCF Architecture 1-2
LCF debugging
  lcfd 3-3
  wdebug 3-3
LCF methods 2-5
  debugging 3-3
  linking 3-2
LCF objects 2-4
libcpl 5-1
  callback registration functions 5-8
    cpl_correct_path 5-2
    cpl_fclose 5-7
    cpl_fopen 5-7
    cpl_fprintf 5-7
    cpl_get_current_location 5-3
    cpl_getc 5-8
    cpl_getcwd 5-4
    cpl_gethostname 5-6
    cpl_getopt 5-5
    cpl_gettimeofday 5-5
    cpl_insque 5-13
    cpl_ltoa 5-6
    cpl_putenv 5-4
    cpl_remque 5-13
    cpl_THREADyield 5-11
    cpl TMPDIR 5-11
    cpl tmpfile 5-10
    cplTmpnam 5-10
dirent functions 5-4
getenv 5-4
MSVC runtime 5-6
removing and inserting elements in a queue 5-13
systhread 5-11
temporary file functions 5-10
UCB compatibility functions 5-11
uname 5-11
wstat macros 5-12
libmrt 2-4
  C data types 4-23
  exception 4-4
  memory management 4-2
libpcl
  cpl_getpass 5-5
libtask_lcf.a 8-2
lightweight client framework
  overview 1-1
linking LCF methods 3-2
log
  default 4-12
  lcfd.log 4-12
log file 4-12
  display threshold 4-12, A-29
  setting display threshold 4-12, A-29
LogData 4-13, 4-22, A-8
LogDeinit 4-11, 4-13, 4-14, A-9
LogGetAppName 4-13, 4-20, A-10
LogGetDefault 4-13, 4-17, A-11
LogGetOutputStdout 4-13, 4-19, A-12
LogGetThreshold 4-13, 4-18, A-13
logging routines
  LogData 4-13
  LogDeinit 4-13
  LogGetAppName 4-13
  LogGetDefault 4-13
  LogGetOutputStdout 4-13
  LogGetThreshold 4-13
  LogInit 4-13
  LogMsg 4-13
  LogQ 4-13
  LogQGetBuffer 4-13
  LogQueueAlloc 4-13
  LogQueueDealloc 4-13
  LogQueueGetSize 4-13
  LogSetAppName 4-13
  LogSetDefault 4-13
  LogSetThreshold 4-13
logging utility 4-10, 4-11
LogInit 4-12, 4-13, 4-17, A-15
LogMsg 4-13, 4-15, 4-20, 4-22,
  A-17, A-18
LogQ 4-13, 4-20, 4-22, A-20
LogQGetBuffer 4-13, 4-21, A-25
LogQueueAlloc 4-13, A-22
LogQueueDealloc 4-13, A-23
LogQueueGetSize 4-13, A-24
LogSetAppName 4-13, 4-19, A-26
LogSetDefault 4-12, 4-13, 4-16, A-27
LogSetOutputStdout 4-13, 4-18, A-28
LogSetThreshold 4-13, 4-17, A-29
lseek_ex 4-10, A-36
ltid 3-2

M
make_path 4-10, A-38
mkdir_ex 4-10
marshalling routines 4-22
memory management routines 4-2
message catalog 4-5
message catalogs 4-5, 4-6
messages
    compound 4-6
method dependencies. See dependencies methods
    and dependencies 6-1
full framework 2-2
IDL files 3-1
methods in LCF 2-3
mg calloc 4-3
mg cleanup 4-3
mg free 4-3
mg malloc 4-3
mg realloc 4-3
mg_strdup 4-3
mkdir_ex A-39
ml calloc 4-3
ml create 4-3
ml destroy 4-3
ml_ex calloc 4-3
ml_ex malloc 4-3
ml_ex realloc 4-3
ml_ex_strdup 4-3
ml_free 4-3
ml malloc 4-3
ml realloc 4-3
ml_strdup 4-3
ml_to_mg 4-3
mrt_check_dependency A-56
mrt_machine_id 4-27, A-55
mrt_set_method_exit_mode 4-27, A-53
mrt_test_dependency 4-27
msvc_stat_macros 5-6

N
nw_echo_command_to_console 4-27, A-57

O
open_ex 4-10, A-40
out-of-cache dependency 6-5

P
processes
    launching 4-25
profile managers 7-3
    dataless mode 7-4
programming environment 2-1, 3-1

R
read_ex 4-10, A-42
rename_file_ex 4-10, A-46
ReThrow 4-4
### S
- seq.h 4-8
- sequence_t type 4-9
- sequences 4-8
  - sequence_t type 4-9
- set_current_credentials 4-27, 4-28
- set_lang 4-27, 4-28, A-58
- set_locale 4-28
- still_alive 4-27, 4-28, A-59
- systhread 5-11

### U
- uname 5-11
- upcall 2-3

### V
- vaAddMsg 4-5, 4-6, 4-7
- vaMakeException 4-4
- variable argument exceptions 4-4
- vaThrow 4-4, 4-5, 4-8
- vaThrowDerived 4-4, 4-8

### W
- wchdep A-3
- wchdep command 6-3
- wdchdep command 6-4
- wdepset A-4
- wdepset command 6-2, 6-3, 6-4
- wgetdeps 6-5, A-7
- write_ex 4-10, A-44

---

**Application Development for the Lightweight Client Framework**

**Index–5**