VMS Software

VSI ACMS for OpenVMS Writing Applications

Operating System and Version: VSI OpenVMS Alpha Version 8.4-2L1 or higher
VSI OpenVMS IA-64 Version 8.4-1H1 or higher
VSI OpenVMS x86-64 Version 9.2-1 or higher

Software Version: ACMS for OpenVMS Version 5.3-3

VMS Software, Inc. (VSI)
Boston, Massachusetts, USA

VSI ACMS for OpenVMS Writing Applications

Copyright © 2025 VMS Software, Inc. (VSI), Boston, Massachusetts, USA

Legal Notice

Confidential computer software. Valid license from VSI required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for VSI products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. VSI shall not be liable for technical or editorial errors or omissions contained herein.

All other trademarks and registered trademarks mentioned in this document are the property of their respective holders.

Table of Contents

Preface xi
1. About VSI xi
2. Intended Audience xi
3. Document Structure xi
4. ACMS Help xiii
5. Related Documents xiv
6. OpenVMS Documentation xv
7. VSI Encourages Your Comments xv
8. Conventions xv

Part I. Writing Generic Applications

Chapter 1. Writing Definitions with ADU 3
1.1. Starting and Stopping ADU 3
1.1.1. Starting ADU 3
1.1.2. Stopping ADU 5
1.1.3. Assigning a Default Text Editor 5
1.2. ACMS and the Data Dictionary 6
1.2.1. Dictionary Path Names 6
1.2.2. Creating CDO Directories and Establishing Directory and Entity Protection 7
1.3. Establishing Your ADU Environment 9
1.4. Understanding ACMS Terminology 11
1.4.1. Identifier 11
1.4.2. File Specification 12
1.4.3. Workspace Field Name 12
1.4.4. Text Strings 13
1.5. Creating and Processing Definitions 14
1.6. How ACMS Uses Definitions 14
1.6.1. Creating ADU Definition Files 15
1.6.2. ACMS Definitions: Placing in and Retrieving from CDD 15
1.6.3. Annotating Definitions 17
1.6.4. Abbreviating Commands and Keywords 17
1.6.5. Using Command Qualifiers 18
1.6.6. Using the Command Continuation Character 18
1.6.7. Responding to Command Prompts 18
1.6.8. Leaving ADU Temporarily 19
1.6.9. Using ADU Interactively 19
1.6.10. Using the Language-Sensitive Editor to Create Definition Files 20
1.7. Getting Help 21

Chapter 2. Defining Tasks 23
2.1. Structure of an ACMS Task 23
2.2. Defining a Data Entry Task 24
2.2.1. Using Workspaces to Pass Data Between Steps 26
2.2.2. Defining the Block Step 28
2.2.3. Defining Characteristics of the Task 29
2.2.4. Storing a Task Definition in the Dictionary 30
2.2.5. Additional Considerations: Error Handling and Ease of Use 31
2.2.5.1. Using ACMS Workspaces 32
2.2.5.2. Repeating the Task Automatically 37

2.3. Defining an Inquiry Task 38
2.3.1. Getting Information from the User 38
2.3.2. Retrieving Information from a File 39
2.3.3. Displaying Information 40
2.3.4. Completing the Task Definition 41
2.3.5. Additional Considerations: Displaying Multiple Records 42
2.3.5.1. Getting Information from the User 43
2.3.5.2. Retrieving Information 44
2.3.5.3. Displaying Information to the User 45
2.3.5.4. Completing the Task Definition 45
2.4. Defining an Update Task 47
2.4.1. Getting Information from the User 48
2.4.2. Retrieving Information from a File 49
2.4.3. Letting the User Update the Information 50
2.4.4. Writing the New Information to the File 51
2.4.5. Completing the Task Definition 52
Chapter 3. Using VSI DECforms with ACMS 55
3.1. ACMS Interface to VSI DECforms 55
3.1.1. Calls to External Requests 55
3.1.2. Processing External Requests 55
3.1.3. Responses to External Requests 56
3.2. Writing and Compiling VSI DECforms Escape Units 57
3.2.1. Writing an Escape Unit 57
3.2.2. Calling an Escape Unit in a Form Source IFDL File 58
3.3. Linking VSI DECforms Form Objects 58
3.4. Linking Escape Units 58
3.4.1. Managing Escape Unit Files 60
3.4.2. Replacing Form Files 60
3.5. Creating Forms Trace Files 61
3.6. Naming Forms Image Files 61
3.7. User Interface Features with VSI DECforms 62
3.8. Comparison of VSI DECforms and TDMS 62
Chapter 4. Defining Workspaces 65
4.1. Understanding the Types of ACMS Workspaces 65
4.2. Handling Errors with a Task Workspace 66
4.3. Using Data Supplied at the Menu 69
4.4. Using Group Workspaces 72
4.5. Using User Workspaces 75
4.6. Moving Data to a Workspace Field 78
4.7. Passing Data with User-Written Agents 79
4.8. Using External Global Symbols in Task Definitions 79
Chapter 5. Using the Task-Call-Task Feature 81
5.1. Calling Another Task 81
5.1.1. Defining a Task Call 82
5.1.2. Task Call Example 83
5.1.3. Passing Workspaces 87
5.1.3.1. Using System Workspaces 88
5.1.3.2. Handling User and Group Workspaces 89
5.1.3.3. Tasks You Also Select from a Menu 89
5.1.3.4. Example of Updating Group and User Workspaces 90
5.1.4. Controlling Called Tasks 92

5.1.4.1. Passing Control Information in User-Defined Workspaces 92
5.1.4.2. Ending a Called Task 94
5.1.4.3. Controlling Parent Tasks 95
5.1.5. Defining Local Tasks 95
5.1.6. Mixing I/O Methods in Parent and Called Tasks 96
5.1.7. Form and Server Context in Task Calling 96
5.1.8. Using the NOT CANCELABLE BY TASK SUBMITTER Clause 96
5.1.9. Auditing and Operation 97
5.1.9.1. Task Auditing and Security 97
5.1.9.2. Using ACMS Operator SHOW and CANCEL Commands 98

Chapter 6. Using the Detached Task Feature 101

6.1. Overview of Detached Tasks 101
6.1.1. Detached Tasks 101
6.1.2. Designing Your Application to Use Detached Tasks 101
6.1.3. Characteristics of Detached Tasks 102
6.2. Managing Detached Tasks 102
6.2.1. Starting a Detached Task 102
6.2.2. Setting the Retry Limit 103
6.2.3. Setting the Retry Wait Timer 103
6.2.4. Showing the Status of Detached Tasks 103
6.2.4.1. Using the ACMS/SHOW TASK Command 103
6.2.4.2. Using the ACMS/SHOW APPLICATION/DETACHED_TASKS Command 104
6.2.5. Stopping a Detached Task 104
6.2.6. Forcing a Detached Task to Not Retry 105
6.2.6.1. Task Failures that Cause ACMS Not to Retry a Task 105
6.2.7. Broadcasting Detached Task Messages 106
6.3. Using Group Workspaces in a Detached Task 106
6.4. Concurrent-Use Licensing with Detached Tasks 107

Chapter 7. Defining Distributed Transactions 109

7.1. Why Use Distributed Transactions 109
7.2. Including Distributed Transactions Syntax in the Task Definition 110
7.3. Including Multiple Resource Managers in a Distributed Transaction 111
7.4. Using Task Sequencing Actions in a Distributed Transaction 112
7.5. Including a Called Task in a Distributed Transaction 113
7.6. How Distributed Transactions Affect Server Context 116
7.7. Excluding a Processing Step from a Distributed Transaction 116
7.8. Handling Deadlocks and Transaction Failures 119

Chapter 8. Handling Task Execution Errors 121

8.1. Why Use Exception Handling 121
8.2. What is an Exception 121
8.2.1. Step Exceptions 122
8.2.2. Transaction Exceptions 123
8.2.3. Nonrecoverable Exceptions 124
8.3. Using the RAISE EXCEPTION Clause 125
8.4. Using Exception Handler Actions 125
8.5. Examples of Exception Handling 127
8.5.1. Recovering from a VSI DECforms Time-Out Exception 128
8.5.2. Recovering from a Task-Call-Task Exception 129
8.5.3. Recovering from a Transaction Exception 130
8.6. How ACMS Performs Exception Handling 133

8.6.1. Executing a Step Exception Outside of a Distributed Transaction 133
8.6.2. Executing a Step Exception Within a Distributed Transaction 133
8.6.3. Executing a Transaction Exception 134
8.6.4. Executing Nonrecoverable Exceptions 134
8.7. How Exceptions Affect Server Cancel Procedures 134
8.7.1. Step Exceptions and Server Cancel Procedures 135
8.7.2. Nonrecoverable Exceptions Raised by Action Clauses 135
8.7.3. Other Nonrecoverable Exceptions and Transaction Exceptions 136
Chapter 9. Queuing ACMS Tasks 137
9.1. Understanding the ACMS Queuing Facility 137
9.2. Using ACMS Queuing with Distributed Transactions 139
9.3. Steps in Using ACMS Queuing 140
9.4. Defining Queue Security 141
9.5. Using the ACMS Queue Services to Queue and Dequeue Tasks 142
9.5.1. Queuing Tasks Using the ACMS$QUEUE_TASK Service 142
9.5.2. Dequeuing Task Elements Using the ACMS$DEQUEUE_TASK Service 143
9.6. Using the QTI to Dequeue Tasks 143
9.6.1. Characteristics of Queued Tasks That are Processed by the QTI 144
9.6.2. Setting ACMSGEN Parameters for the QTI Process 145
9.6.2.1. Assigning a User Name to the QTI Process 145
9.6.2.2. Assigning a Priority to the QTI Process 145
9.6.2.3. Controlling Submitter Sign-Ins 145
9.6.2.4. Setting the Retry Time for Failed Tasks 146
9.6.2.5. Setting the Polling Time for Task Queues 146
9.6.3. Auditing Done by the QTI Process 146
9.6.4. How the QTI Handles Errors 147
9.7. Processing Error Queues 149
9.8. Debugging Queued Tasks 155
9.9. Online Backup of Task Queue Files 156
9.10. Queuing Example 156
9.11. Procedure Parameter Notation for Programming Services 167
Chapter 10. Defining Task Groups 177
10.1. Defining a Task Group 177
10.2. Identifying Which Tasks Belong to the Task Group 177
10.3. Identifying Which Servers Are Required in the Group 178
10.3.1. Assigning Server Attributes 181
10.4. Naming Request Libraries 182
10.5. Identifying Which Message Files Are Used in the Group 182
10.6. Naming Workspaces in a Task Group Definition 183
10.7. Naming the Task Database for a Task Group 183
10.8. Changing Characteristics of Task Argument Workspaces 185
Chapter 11. Defining Applications 187
11.1. Defining a Simple Application 187
11.2. Describing the Application Environment 187
11.2.1. Naming Task Groups 187
11.2.2. Naming a User Name for the Application Execution Controller 188
11.2.3. Assigning Characteristics to Tasks and Servers 188
11.3. Controlling Tasks 190
11.3.1. Controlling Access to Tasks 190
11.3.2. Auditing Task Events 191
11.3.3. Controlling What Happens When a Task Ends 192

11.3.4. TASK ATTRIBUTES and TASK DEFAULTS Clauses 193
11.3.4.1. Using the TASK ATTRIBUTES Clause 193
11.3.4.2. Using the TASK DEFAULTS Clause 194
11.3.4.3. Defaulting Task and Task Group Names 195
11.3.4.4. Positioning TASK ATTRIBUTES and TASK DEFAULTS Clauses 196
11.3.5. Enabling and Disabling Tasks in the Application Definition 197
11.3.6. Controlling Transaction Timeouts in the Application Definition 198
11.4. Controlling Servers 199
11.4.1. Assigning a Server User Name 200
11.4.2. Assigning a Dynamic or Fixed Server User Name 201
11.4.3. Assigning Server Default Directories 202
11.4.4. Assigning Server Logical Names 203
11.4.5. Creating Logical Name Tables for Application Servers 204
11.4.6. Controlling the Number of Server Processes 205
11.4.7. Creating and Deleting Server Processes 206
11.4.8. Replacing an Active Server 207
11.4.9. SERVER ATTRIBUTES and SERVER DEFAULTS Clauses 208
11.4.10. Defaulting Server and Task Group Names 210
11.4.11. Positioning SERVER ATTRIBUTES and SERVER DEFAULTS Clauses 211
11.4.12. Auditing Servers 212
11.4.13. Enabling Procedure Server Process Dumps 212
11.5. Controlling Applications 213
11.5.1. Assigning an Application Execution Controller User Name 213
11.5.2. Auditing Applications 213
11.5.3. Assigning Application Default Directories 214
11.5.4. Assigning Application Logical Names 214
11.5.5. Assigning Application Database Files 215
11.5.6. Controlling the Number of Server Processes 216
11.5.7. Controlling the Number of Task Instances 217
11.6. Modifying an Active Application 218
11.7. Controlling Application Failover 218

Chapter 12. Defining Menus 221

12.1. Planning the Menu Structure 221
12.2. Defining Menus 223
12.2.1. Creating a Title for a Menu 224
12.2.2. Naming Entries on a Menu 224
12.2.3. Naming Menus 225
12.2.4. Naming Tasks on a Menu 225
12.2.5. Specifying WAIT or DELAY Action 225
12.2.6. Naming Default Application Files 226
12.2.7. Application Specifications and Remote Tasks 227
12.2.8. Naming Default Menu Files 228
12.2.9. Defining a Menu Forms Product 228
12.3. Processing the Menu Definition 231

Chapter 13. Defining Existing Applications as ACMS Tasks 233

13.1. Defining Single-Step Tasks in ACMS Task Groups 233
13.1.1. Defining OpenVMS Images as Tasks 233
13.1.2. Defining DCL Commands and Command Procedures as Tasks 234
13.1.3. Defining DATATRIEVE Commands and Procedures as Tasks 234
13.2. Defining Servers to Handle Processing 235
13.3. Using the Task Group in an Application 236

Chapter 14. Using the ACMS Request Interface 237
14.1. Overview of the ACMS Request Interface 237
14.2. The Request Interface and the ACMS Run-Time System 239
14.3. Defining Tasks and Task Groups 240
14.3.1. Task Definition 241
14.3.2. Defining a Task Group 241
14.3.3. How and When to Use the ACMS$RI_LIB Logical Name 243
14.4. Writing User Request Procedures 244
14.4.1. Writing an ACMS$RI_LIB_INIT Initialization Procedure 246
14.4.2. Writing an ACMS$RI_LIB_CANCEL Cancellation Procedure 248
14.4.3. Compiling and Linking URPs 249
14.5. Providing an RI Agent 251
14.5.1. Providing a Menu Interface 253
14.5.2. Compiling and Linking Menu Interface URPs with the RI Agent 255
14.5.3. User-Written Menus for the ACMS$RI_AGENT 256
14.6. Debugging Applications that Call URPs 256
14.6.1. Using the OpenVMS Debugger to Debug URPs Using a Running Application 257
14.6.2. Using the ACMS Task Debugger to Debug URPs and Their Tasks 258
14.7. Defining an Application that Uses the Request Interface 259
14.8. Running the Agent 259

Part II. Writing and Migrating Applications to OpenVMS

Chapter 15. Introduction 263

Chapter 16. Writing Applications for OpenVMS 265
16.1. Writing an ACMS Application for OpenVMS 265
16.2. Form Changes and Form File Caching 265
16.2.1. Formatting and Naming VSI DECforms Form Image Files 266
16.2.2. Caching with Multiple Submitter Platforms 266
16.3. Using Logical Names 267

Chapter 17. I/O Options and Restrictions 269
17.1. Restrictions for Distributing an Application 269
17.2. Selecting Tasks and Menus on OpenVMS 269
17.2.1. ACMS Menu Commands 270
17.2.2. ACMS Function Keys 271

Part III. Appendixes

Appendix A. Changing the ACMS Menu Format Using VSI DECforms 275
A.1. Modifying Menu Appearance Without Changing the Default Format 275
A.2. Modifying the ACMS Menu Using VSI DECforms 275
A.2.1. Obtaining the ACMS VSI DECforms Default Menu File 276
A.2.2. How ACMS Uses Menu Form Record Definitions 276
A.2.3. Instructions Performed by the Form File 278
A.2.4. Modifying the Menu Appearance Only 279
A.2.5. Changing SELECTION_STRING Field Lengths 284
A.2.6. Changing the Number of Entries per Screen 285
A.2.7. Changing the Size of the Command Line Recall Buffer 287
A.2.8. Changing the VSI DECforms Layout Size 287
A.2.9. Using a Customized Response and Panel Definition 288
A.2.10. Building and Installing the New Menu Form 289

A.3. Disabling the SELECT Command in the ACMS Command Menu 289
Appendix B. Changing the ACMS Menu Format Using TDMS 291
B.1. Modifying the Menu Format Using ACMS 291
B.2. Modifying the ACMS Menu Using TDMS Requests 291
B.2.1. Getting the ACMS Menu Request and Form 292
B.2.2. Modifying the Menu Form Only 293
B.2.3. Forms, Records, and Keypad Used by the Menu Request 294
B.2.4. Instructions Performed by TDMS When ACMS Calls a Request 297
B.2.5. Modifying the Menu Request 300
B.2.6. Using the REQUEST Clause 303
B.2.7. Changing the MENU_ENTRY_RECORD 304
B.2.8. Modifying and Building the ACMS Menu Request Library 305
B.3. Disabling the SELECT Command in the ACMS Command Menu 306
Appendix C. Using CDO for Pieces Tracking 309
C.1. Overview of Dictionary Object Types 309
C.2. Creating Relationships Between Entities 310
Appendix D. Using LSE with ACMS 319
D.1. Using LSE with ACMS 319
D.2. Creating ACMS Source Files with LSE 319
D.2.1. Using Placeholders and Tokens in LSE 321
D.2.2. Creating the Final Source File 325
D.2.2.1. Syntax Differences 326
D.2.2.2. Exiting Editing Mode 326
D.3. Compiling Definitions with LSE COMPILE 326
D.4. Examining Diagnostic Messages with LSE REVIEW 327
D.4.1. Generating the Diagnostics File 327
D.4.2. Examining the Diagnostics File with LSE REVIEW 328
D.5. Using HELP in LSE 330
Appendix E. Request Interface Kit Components 331
E.1. Application Independent Modules 331
E.2. RI Sample Application 333
E.2.1. RI Sample Application Source Modules 334
E.2.2. RI Sample Application Run-Time Files 338
E.3. FMS Sample Application 340
E.3.1. FMS Sample Application Source Modules 340
E.3.2. FMS Sample Application Run-Time Files 346
Appendix F. Modifying the FMS Menu Interface 349
Appendix G. Checklist of References to Platform-Specific Files 353
G.1. Task Group Definition 353
G.2. ADU BUILD GROUP Command 353

Preface

1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

2. Intended Audience

This manual explains how to define transaction processing applications using VSI ACMS for OpenVMS (ACMS) software. In particular, this manual describes:

- How to use the Application Definition Utility (ADU)
- How to write definitions of multiple-step tasks, task groups, menus, and applications

To use this manual effectively, you should be experienced in designing or defining online applications but not necessarily experienced with ACMS application design or definition. Less experienced applications programmers can review *VSI ACMS for OpenVMS Getting Started* [https://docs.vmssoftware.com/vsi-acms-get-started-guide/] before reading this manual.

Read this manual if you plan to use ADU to create and process task, task group, application, and menu definitions, or if you are working with the completed definitions.

You should be familiar with the ACMS operating system.

3. Document Structure

This manual contains two parts: *Part I, "Writing Generic Applications"* describes how to use ACMS to define transaction-processing applications. *Part II, "Writing and Migrating Applications to OpenVMS"* describes how to write ACMS applications for an OpenVMS system. The chapters and appendixes are described in the following table:

Part I, "Writing Generic Applications"	
Chapter 1, "Writing Definitions with ADU"	Introduces ADU and explains how to set up your environment to use ADU and create ACMS definitions.
Chapter 2, "Defining Tasks"	Provides an introduction to the design and definition of ACMS tasks; explains how to define data entry, inquiry, and update multiple-step tasks.
Chapter 3, "Using VSI DECforms with ACMS"	Describes the VSI DECforms interface to ACMS; shows how to write, compile, and link VSI DECforms escape units.
Chapter 4, "Defining Workspaces"	Describes and shows how to use the four types of workspaces: task, system, group, and user.
Chapter 5, "Using the Task-Call-Task Feature"	Shows how to call a task from another task; describes the rules for passing workspaces between tasks.
Chapter 6, "Using the Detached Task Feature"	Describes how to use detached tasks for applications that need batch-style processing,

	where no user interaction occurs through a terminal or device.
Chapter 7, "Defining Distributed Transactions"	Shows how to define tasks that start and end transactions involving multiple resource managers, such as databases or files.
Chapter 8, "Handling Task Execution Errors"	Describes the different types of errors that can interrupt the execution of tasks, and shows how to define tasks that test for and recover from task execution errors.
Chapter 9, "Queuing ACMS Tasks"	Explains how to queue tasks in an application for later processing.
Chapter 10, "Defining Task Groups"	Explains how to write task group definitions to establish common characteristics or shared resources for a group of tasks.
Chapter 11, "Defining Applications"	Explains how to write application definitions to assign control characteristics for tasks and applications.
Chapter 12, "Defining Menus"	Explains how to write menu definitions.
Chapter 13, "Defining Existing Applications as ACMS Tasks"	Explains how to implement existing programs, DCL commands, and DATATRIEVE procedures as ACMS tasks.
Chapter 14, "Using the ACMS Request Interface"	Explains how to develop applications that use the ACMS Request Interface.
Part II, "Writing and Migrating Applications to OpenVMS"	
Chapter 15, "Introduction"	Describes ACMS on OpenVMS.
Chapter 16, "Writing Applications for OpenVMS"	Describes how to write applications to run on an OpenVMS system.
Chapter 17, "I/O Options and Restrictions"	Describes the I/O options and restrictions in a distributed environment.
Part III, "Appendixes"	
Appendix A, "Changing the ACMS Menu Format Using VSI DECforms"	Shows how to modify the standard ACMS menu format using by VSI DECforms.
Appendix B, "Changing the ACMS Menu Format Using TDMS"	Shows how to modify the standard ACMS menu format by using TDMS.
Appendix C, "Using CDO for Pieces Tracking "	Shows how to use CDO to track relationships between ACMS entities in the CDD dictionary.
Appendix D, "Using LSE with ACMS"	Demonstrates how to use the optional Language-Sensitive Editor productivity tool to enter ACMS code on line.
Appendix E, "Request Interface Kit Components"	Lists the files included with the ACMS Request Interface examples and software supplied by ACMS.
Appendix F, "Modifying the FMS Menu Interface"	Explains how to modify the FMS menu interface that you use with the ACMS Request Interface.

Appendix G, "Checklist of References to Platform-Specific Files"	Contains a list of references to platform-specific files in an ACMS application.

4. ACMS Help

ACMS and its components provide extensive online help.

- DCL level help

  Enter **HELP ACMS** at the DCL prompt for complete help about the **ACMS** command and qualifiers, and for other elements of ACMS for which independent help systems do not exist. DCL level help also provides brief help messages for elements of ACMS that contain independent help systems (such as the ACMS utilities) and for related products used by ACMS (such as DECforms or Oracle CDD/Repository).

- ACMS utilities help

  Each of the following ACMS utilities has an online help system:

  - ACMS Debugger ACMSGEN Utility
  - ACMS Queue Manager (ACMSQUEMGR)
  - Application Definition Utility (ADU)
  - Application Authorization Utility (AAU)
  - Device Definition Utility (DDU)
  - User Definition Utility (UDU)
  - Audit Trail Report Utility (ATR)
  - Software Event Log Utility Program (SWLUP)

  The two ways to get utility-specific help are:

  - Run the utility and type **HELP** at the utility prompt.
  - Use the DCL **HELP** command. At the "Topic?" prompt, type @ followed by the name of the utility. Use the ACMS prefix, even if the utility does not have an ACMS prefix (except for SWLUP). For example:

    ```
    Topic? @ACMSQUEMGR
    Topic? @ACMSADU
    ```

    However, do not use the ACMS prefix with SWLUP:

    ```
    Topic? @SWLUP
    ```

Note

Note that if you run the ACMS Debugger Utility and then type **HELP**, you must specify a file. If you ask for help from the DCL level with @, you do not need to specify a file.

- ACMSPARAM.COM and ACMEXCPAR.COM help

  Help for the command procedures that set parameters and quotas is a subset of the DCL level help. You have access to this help from the DCL prompt, or from within the command procedures.

- LSE help

  ACMS provides ACMS-specific help within the LSE templates that assist in the creation of applications, tasks, task groups, and menus. The ACMS- specific LSE help is a subset of the ADU help system. Within the LSE templates, this help is context-sensitive. Type **`HELP/IND (PF1-PF2)`** at any placeholder for which you want help.

- Error help

  ACMS and each of its utilities provide error message help. Use **`HELP ACMS ERRORS`** from the DCL prompt for ACMS error message help. Use **`HELP ERRORS`** from the individual utility prompts for error message help for that utility.

- Terminal user help

  At each menu within an ACMS application, ACMS provides help about terminal user commands, special key mappings, and general information about menus and how to select tasks from menus.

- Forms help

  For complete help for DECforms or TDMS, use the help systems for these products.

5. Related Documents

The following table lists the documents in the VSI ACMS for OpenVMS documentation set.

ACMS Information	Description
VSI ACMS Version 5.0 for OpenVMS Installation Guide [https://docs.vmssoftware.com/vsi-acms-installation-guide/]	Description of installation requirements, the installation procedure, and postinstallation tasks.
VSI ACMS for OpenVMS Getting Started [https://docs.vmssoftware.com/vsi-acms-get-started-guide/]	Overview of ACMS software and documentation. Tutorial for developing a simple ACMS application. Description of the AVERTZ sample application.
VSI ACMS for OpenVMS Concepts and Design Guidelines [https://docs.vmssoftware.com/vsi-acms-concepts-and-design-guide/]	Description of how to design an ACMS application.
VSI ACMS for OpenVMS Writing Applications [https://docs.vmssoftware.com/vsi-acms-writing-apps/]	Description of how to write task, task group, application, and menu definitions using the Application Definition Utility. Description of how to write and migrate ACMS applications on an OpenVMS system.
VSI ACMS for OpenVMS Writing Server Procedures [https://docs.vmssoftware.com/vsi-acms-writing-server-proc/]	Description of how to write programs to use with tasks and how to debug tasks and programs.
VSI ACMS for OpenVMS Systems Interface Programming [https://docs.vmssoftware.com/vsi-acms-sys-interface-prog/]	Description of using Systems Interface (SI) Services to submit tasks to an ACMS system.

ACMS Information	Description
VSI ACMS for OpenVMS ADU Reference Manual [https://docs.vmssoftware.com/vsi-acms-adu-ref-manual/]	Reference information about the ADU commands, phrases, and clauses.
VSI ACMS for OpenVMS Quick Reference [https://docs.vmssoftware.com/vsi-acms-quick-ref/]	List of ACMS syntax with brief descriptions.
VSI ACMS for OpenVMS Managing Applications [https://docs.vmssoftware.com/vsi-acms-managing-applications/]	Description of authorizing, running, and managing ACMS applications, and controlling the ACMS system.
VSI ACMS for OpenVMS Remote Systems Management Guide [https://docs.vmssoftware.com/vsi-acms-remote-systems-management-guide/]	Description of the features of the Remote Manager for managing ACMS systems, how to use the features, and how to manage the Remote Manager.
Online help	Online help about ACMS and its utilities.

6. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at https://docs.vmssoftware.com.

7. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: <docinfo@vmssoftware.com>. Users who have VSI OpenVMS support contracts through VSI can contact <support@vmssoftware.com> for help with this product.

8. Conventions

The following conventions are used in this manual:

Ctrl/*x*	A sequence such as **Ctrl/*x*** indicates that you must press and hold the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as **PF1 *x*** indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.
Return	In the HTML version of this document, this convention appears as brackets rather than a box.
. . .	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
⋮	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.

`Monospace text`	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example. In the HMTL version of this document, this text style may appear as italics.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.
bold text	Bold text represents the introduction of a new term or the name of an argument, an attribute, or a reason. In the HMTL version of this document, this text style may appear as italics.
italic text	Italic text indicates important information, complete titles of manuals, or variables. Variables include information that aries in system output (Internal error *number*), in command lines (**/PRODUCER=*name***), and in command parameters in text (where *dd* represents the predefined code for the device type).
UPPERCASE	Uppercase text indicates the name of a routine, the name of a file, the name of a file protection code, or the abbreviation for a system privilege. In command format descriptions, uppercase text is an optional keyword.
UPPERCASE	In command format descriptions, uppercase text that is underlined is required. You must include it in the statement if the clause is used.
lowercase	In command format descriptions, a lowercase word indicates a required element.
<lowercase>	In command format descriptions, lowercase text in angle brackets indicates a required clause or phrase.
( )	In command format descriptions, parentheses indicate that you must enclose the options in parentheses if you choose more than one.
[| |]	In command format descriptions, vertical bars within square brackets indicate that you can choose any combination of the enclosed options, but you can choose each option only once.
{ | | }	In command format descriptions, vertical bars within braces indicate that you must choose one of the options listed, but you can use each option only once.

Part I. Writing Generic Applications

This part explains how to define transaction-processing applications using ACMS software. In particular, this part describes:

- How to use the Application Definition Utility (ADU)
- How to write definitions of multiple-step tasks, task groups, menus, and applications

Chapter 1. Writing Definitions with ADU

One of the primary features of ACMS is the Application Definition Utility (ADU). The ADU is the principal tool used to create and process the definitions that comprise an ACMS application. You use ADU commands to write definitions for ACMS tasks, task groups, menus, and applications. When processing these definitions, ACMS builds the task group, menu, and application databases that it uses at run time to operate the application.

This book contains tutorial information showing how to use ADU to create ACMS applications. The *VSI ACMS for OpenVMS ADU Reference Manual* [https://docs.vmssoftware.com/vsi-acms-adu-ref-manual/], the companion volume, contains reference material for all the ADU commands. Together these books contain much of the information needed to create and build ACMS applications.

Chapter 1, "Writing Definitions with ADU" contains the information on how to use ADU. Topics covered include:

- Using ACMS commands to start and stop ADU
- Creating and processing definitions
- Using CDD path names
- Explaining ADU terminology
- Using ADU commands

1.1. Starting and Stopping ADU

This section explains how to invoke the Application Definition Utility (ADU) and exit from it.

1.1.1. Starting ADU

There are three ways to invoke ADU. Two methods use startup qualifiers; the third allows you to enter the utility only in default mode. After invoking the utility, ACMS displays the ADU> prompt.

The three ways to start ADU are:

- By using the **MCR** command

  Start ADU by entering the following command at the DCL prompt:

  ```
  $ MCR ACMSADU
  ADU>
  ```

  Include startup command qualifiers on the MCR ACMSADU command line.

- By defining a foreign command

  Define a foreign command in your LOGIN.COM file to invoke ADU. Then, whenever you enter that command at the DCL prompt, you are in ADU.

Before using the foreign command ADU to invoke the utility, put the following definition in your LOGIN.COM file. Then process the LOGIN.COM by entering @LOGIN.COM at the DCL prompt to make the foreign command available for the current session. After the current session, the command is automatically defined whenever you log in.

The following definition creates ADU as the foreign command to invoke the utility:

```
$ ADU :== $ACMSADU
```

After defining the foreign command and processing the login file, enter ADU at the DCL prompt to invoke the utility:

```
$ ADU
ADU>
```

Include startup command qualifiers on the command line by using the command:

```
$ ADU /COMMAND=RESERVATIONS
```

- By using the **RUN** command

  Enter the **RUN** command at the DCL prompt to invoke ADU:

  ```
  $ RUN SYS$SYSTEM:ACMSADU
  ADU>
  ```

  Do not include ADU command qualifiers when invoking the utility with the **RUN** command.

Table 1.1, "Startup Qualifiers and Their Functions" lists the startup command qualifiers and their functions. Use these qualifiers when invoking ADU with the **MCR** command or a foreign command.

Table 1.1. Startup Qualifiers and Their Functions

Qualifier	Function
/COMMAND [=*file-spec*] NOCOMMAND	Tells ADU whether or not to execute a startup command file when you invoke the utility. By default, when you invoke ADU, it runs a command file named ADUINI.COM, located in your default directory. To invoke a different startup command file, include its file specification with the **/COMMAND** qualifier. When you specify the **/NOCOMMAND** qualifier, ACMS starts the ADU without executing any startup command file.
/JOURNAL /NOJOURNAL	By default, ADU creates a journal file that contains every keystroke made during your ADU session. The journal file, named ADUJNL.JOU, is located in your default directory. The journal file is saved if your ADU session is interrupted. When you exit normally (by using the **EXIT** command or entering **Ctrl/Z**), the journal file is not saved. Use the **/NOJOURNAL** qualifier to turn off the journaling feature.
/PATH=*path-name*	Assigns a CDD directory. If you do not specify a path name, ADU uses the default CDD directory.
/RECOVER /NORECOVER	If you specify the **/RECOVER** qualifier, ADU runs the journal file, ADLJNL.JOU, to restore an ADU session that has ended abnormally.

Qualifier	Function
	With **/RECOVER** in effect, ADU replays the interrupted session to recover your work. **/NORECOVER** is the default.

1.1.2. Stopping ADU

There are three methods to stop the ADU utility. Two methods result in an orderly exit from the utility. The third method causes an abrupt exit and should be used only when the other methods fail. *Table 1.2, "Ways to Exit from ADU"* lists the ways to end an ADU session.

Table 1.2. Ways to Exit from ADU

Command	Meaning
EXIT	Ends your ADU session and returns control to the DCL command level without issuing any messages. Using the **EXIT** command produces the same results as pressing **Ctrl/Z**. When you create a file of ADU commands to automatically run a session, enter only the **EXIT** command in the file to terminate the automated session.
Ctrl/Z	Ends your ADU session and returns control to the DCL command level without issuing any messages.
Ctrl/Y	Abruptly ends your ADU session and returns control to the DCL command level without displaying any messages. Using **Ctrl/Y** can leave your definitions in an inconsistent state; so, use this method of exiting from ADU only when other methods fail.

1.1.3. Assigning a Default Text Editor

Two ADU commands, **EDIT** and **MODIFY**, require a text editor. By default, ADU uses the OpenVMS EDT editor. If you plan to use an editor other than EDT, you must set up a logical name, ADU$EDIT, to point to a command file naming the editor you want to use. You can use the DCL **DEFINE** or **ASSIGN** command to set up a pointer to the command file. For example:

```
$ DEFINE ADU$EDIT MYDISK$:[MYDIRECT]ADUEDIT.COM
```

This command assigns the logical name ADU$EDIT to the ADUEDIT.COM command file in the directory MYDIRECT on the device MYDISK$. Including this command in your LOGIN.COM file defines in the ADUEDIT.COM the editor you want. If the ADUEDIT.COM file is not in your default directory, be sure to include the disk and directory specification.

The following example demonstrates how to name TPU by using the EDT interface as the default editor. To set up the ADUEDIT.COM file, enter the following commands into a file named ADUEDIT.COM:

```
$ ASSIGN/USER 'F$LOGICAL("SYS$OUTPUT")' SYS$INPUT
$ IF P1 .EQS. "" THEN GOTO INPUT
$ EDIT/TPU/SECTION=EDTSECINI/OUTPUT='P2' 'P1'
$ EXIT
$ NOINPUT:
$ EDIT/TPU/SECTION=EDTSECINI 'P2'
```

The third and last lines of this command procedure name TPU with the EDT interface as the editing environment. You can change these lines to name any other OpenVMS editor you want to use. See

Section 1.6.10, "Using the Language-Sensitive Editor to Create Definition Files" for information on using the Language-Sensitive Editor (LSE).

For information on DCL commands, see *VSI OpenVMS DCL Dictionary: A–M* [https://docs.vmssoftware.com/vsi-openvms-dcl-dictionary-a-m/]. *VSI OpenVMS User's Manual* [https://docs.vmssoftware.com/vsi-openvms-user-s-manual/#COMMS_CH1] has information on writing command and startup procedures.

1.2. ACMS and the Data Dictionary

Prior to ACMS Version 3.1, the ADU Utility placed ACMS definitions in the CDD dictionary in DMU format. Within CDD the definitions are called dictionary objects.

With ACMS Version 3.1 and higher, ADU places ACMS definitions in the CDD dictionary in CDO format. Within CDD the definitions are called entities.

Specify a dictionary path name either in DMU or CDO format. In either case, ACMS stores the dictionary object in CDO format.

You can continue to use DMU-format objects created with earlier versions of ACMS, whether or not you create new CDO format entities with ACMS. However, you cannot use CDO to manipulate previously created DMU-format objects. Nor can you use DMU to manipulate CDO-format objects.

1.2.1. Dictionary Path Names

Every dictionary definition has a path name that uniquely identifies it. The naming conventions for DMU and CDO differ only in their specification of the dictionary origin.

To refer to CDD dictionary definitions, list all dictionary directory given names. Begin with CDD$TOP and end with the dictionary directory or dictionary object you want to identify. Separate names by periods. In the following example, the path name indicates that the RESERVATION_TASK dictionary object is located in the CUSTOMERS directory, which in turn resides in the AVERTZ directory. The AVERTZ directory is in the top-level directory CDD$TOP.

```
CDD$TOP.AVERTZ.CUSTOMERS.RESERVATION_TASK
```

Refer to CDD dictionary entities by specifying a dictionary anchor and a path name. A dictionary anchor specifies the OpenVMS directory where the CDD hierarchy is stored. DISK1:[CDDPLUS] is an example of an anchor. The anchor is followed by the dictionary path. A dictionary path consists of dictionary names separated by periods. All but the last name are dictionary directory names. The last is an entity name. The following example illustrates the full CDD path name for the RESERVATION_TASK dictionary entity:

```
DISK1:[CDDPLUS]AVERTZ.CUSTOMERS.RESERVATION_TASK
```

DISK1:[CDDPLUS] is the anchor. RESERVATION_TASK is the entity in the CUSTOMERS directory, which is located in the AVERTZ directory.

A CDD name can consist of given names containing a maximum of 31 characters. Characters can include the letters A through Z, digits 0 through 9, underscores (_), and dollar signs ($). The first character of each given name must be a letter. The last character can be either a letter or a digit. All lowercase letters are translated to uppercase.

Identify a dictionary object with a full or relative path name. If a default dictionary directory exists, use a relative path name. Relative path names include the portion of the path name that is not part of the default dictionary definition.

Suppose, for example, that the following command establishes your default directory:

```
$ DEFINE CDD$DEFAULT DISK1:[CDDPLUS]AVERTZ.CUSTOMERS
```

With AVERTZ.CUSTOMERS as the default dictionary directory, the relative path name is simply the object name itself: RESERVATION_TASK. If the default directory is [CDDPLUS], the relative dictionary path name is AVERTZ.CUSTOMERS.RESERVATION_TASK.

In DMU, you can assign a password to each dictionary directory, subdirectory, and object in a path name. You cannot assign passwords to CDO entities. When specifying the path name, put the password in parentheses and attach it to the end of the given name to which it applies.

Passwords can contain a maximum of 64 printable ASCII characters, including spaces and tabs. Eight-bit characters from the DEC Multinational Character Set can also be used in passwords. Full support for the 8-bit character set requires software and hardware support. For instance, the display terminal and printer used must both support the 8-bit character set. Lowercase letters are translated to uppercase. The only printable characters you cannot include in path name passwords are parentheses, either left or right, and periods.

The following are legal given names with passwords:

```
JONES(ACMSPASS)

SMITH(CAPT JOHN)
```

The following example shows a path name with a password associated with a dictionary directory:

```
CDD$TOP.PERSONNEL.MENU(ACMSPASS).MENU_WORK
```

You can also specify CDD dictionary specifications for ACMS workspaces. These workspace specifications are stored in the CDD dictionary.

OpenVMS logical names may be used in all or part of a path name. For example, if you define PERSONNEL_CDD as DISK1:[CDDPLUS]PERSONNEL, you can use PERSONNEL_CDD.MENU as a path name.

For more information on CDD path names, see the appropriate CDD documentation.

1.2.2. Creating CDO Directories and Establishing Directory and Entity Protection

Before you create a new ACMS definition or replace an existing one that is in DMU format, you must first create a CDO directory in which to store the dictionary entity. This practice contrasts with the DMU behavior, in which the creation of a dictionary object also created the dictionary directory, if one did not exist. ADU returns an error if you attempt to create an entity in a CDO directory that does not exist.

Default protection for CDO directories and entities is stricter beginning with CDD Version 4.0. When you create a CDO directory or when you create or replace ACMS definitions in CDO format, be aware of the following protection issues:

- Default protection for CDO directories

  Only the owner of the directory can put entities into the directory. You may want to allow other users to be able to place ACMS definitions in the CDO directory. See the CDD documentation for details of establishing default protection for CDO directories.

- Default protection for CDO entities

  Prior to ACMS Version 3.1, ADU used the CDD default protection when creating dictionary objects. Beginning with CDD Version 4.0, the stricter CDO default protections allow only the owner of a CDO entity access to that entity. Therefore, beginning with ACMS Version 3.1, ACMS provides a means for establishing your own default protection scheme for newly created ACMS dictionary entities. This scheme applies the first time that the entity is placed in the CDO directory; that is, when someone creates a new entity or replaces an existing DMU object with a CDO entity. If you replace an existing CDO entity, the replacement entity inherits protection from the entity it replaces.

The ACMS default protection scheme uses the logical name ACMS$ADU_ACL_DEFAULT. Define this process logical to point to an access control list (ACL) file specification.

ACMS also provides the ACMS$ADU_ACL_DEFAULT.COM command procedure in the SYS$MANAGER directory to automate the definition of the logical and the creation or modification of the file to which it points. ACMS$ADU_ACL_DEFAULT.COM consists of a series of menus and forms that allow you to:

- Define or reassign the logical name to point to a file specification, or deassign the logical name. The logical is a process logical. You need to redefine it each time the process starts.

- Add, delete, resequence, or display Access Control Entries (ACE) in the file pointed to by the logical name ACMS$ADU_ACL_DEFAULT. You must define the logical name before you can create or modify the ACE listing. The creation of the first ACE causes the creation of the file, if it does not exist. The command procedure lists the ACE options in both CDD and OpenVMS representations; however, the file created contains only OpenVMS representations.

Following is an example of an ACL file with three ACE entries. You can create this file by using SYS$MANAGER:ACMS$ADU_ACL_DEFAULT.COM. The example was generated by issuing the CDD command **DIR/FULL**.

```
(IDENTIFIER=[ACMS,RICK],ACCESS=READ+WRITE+MODIFY+ERASE+SHOW+DEFINE+CHANGE+
             DELETE+CONTROL+OPERATOR+ADMINISTRATOR)
(IDENTIFIER=[ACMS,HEINZ],ACCESS=READ+WRITE+DELETE)
(IDENTIFIER=[*,*],ACCESS=READ+WRITE+MODIFY+ERASE+SHOW+OPERATOR
+ADMINISTRATOR)
```

Access is determined by the first ACE encountered that applies to the creator of the entity. See the CDD documentation and OpenVMS Systems Services documentation for more information about ACLs for dictionary entities.

Table 1.3, "Establishing Protection for ACMS Dictionary Entities" summarizes the determination of entity protection.

Table 1.3. Establishing Protection for ACMS Dictionary Entities

Action	DMU (ACMS Version 3.0)	CDO (ACMS Version 3.1 or higher)	DMU to CDO Migration
Adding a new object or entity (ADU **CREATE**, ADU **REPLACE**, ADU **COPY**)	Inherits directory's ACL	Use logical; if not, takes CDO default	N/A
Modifying an existing object or entity (ADU	Inherits object's ACL	Inherits entity's ACL	Error if CDO directory does not exist; else, use

Action	DMU (ACMS Version 3.0)	CDO (ACMS Version 3.1 or higher)	DMU to CDO Migration
REPLACE, ADU **MODIFY**)			logical. If no logical, use CDO default.

1.3. Establishing Your ADU Environment

To set up certain characteristics for the ACMS environment, you can create an initialization command file called ADUINI.COM, in which you can put commands for ADU to run at startup. You can include commands assigning the CDD default directory, logging information to a log file, displaying commands processed from an indirect command file, and so forth. Creating the ADUINI.COM file eliminates the need to type the commands each time ADU starts. This command file serves the same purpose for ADU that the LOGIN.COM file serves for the OpenVMS operating system.

Before displaying the ADU> prompt at the beginning of a terminal session, ADU searches for the logical name ADUINI in the logical name table. If it finds the logical name ADUINI, ADU uses it with the default file type .COM. To define the logical name ADUINI, place a line similar to the following in your LOGIN.COM:

```
DEFINE ADUINI MYDISK:[MYDIRECT]MYADUINI.COM
```

If your LOGIN.COM file contains a line similar to this one, ACMS looks for ADU commands at ADU startup in the file MYADUINI.COM located in the directory MYDIRECT on the disk pointed to by the logical name MYDISK. When you use the **DEFINE** or **ASSIGN** DCL commands to assign the logical ADUINI to a disk, directory, and file, ACMS can find the file containing your startup commands.

If you do not assign a logical name using **DEFINE** or **ASSIGN**, ADU searches for the ADUINI.COM file in your default directory (SYS$LOGIN). If it finds the file there, ADU automatically runs it from that location. *Table 1.4, "ADU Commands for ADUINI.COM File"* lists the commands commonly included in the ADUINI.COM file.

Table 1.4. ADU Commands for ADUINI.COM File

Command	Description
SET DEFAULT	Assigns a default CDD directory.
SET [NO]LOG	Starts or stops logging of information to a log file.
SET [NO]VERIFY	Controls whether or not ADU displays commands it runs from an indirect command file.
SHOW DEFAULT	Displays your current CDD default directory on your terminal screen.
SHOW LOG	Displays a message on your terminal screen telling you if logging is active or inactive and the name of the log file.

To create the file ADUINI.COM, use a text editor such as EDT. Include in the file any commands you want ADU to run when you start the utility. Because ADUINI.COM is an ADU startup file and not a DCL command file, do not begin the commands with the DCL prompt. *Example 1.1, "A Sample ADUINI.COM File"* shows a sample ADUINI.COM file.

Example 1.1. A Sample ADUINI.COM File

```
SET VERIFY
SET LOG [ACMS.SAMPLE]DEFINE.LOG
!Change default directory from CDD$DEFAULT
```

```
SET DEFAULT DISK1:[CDDPLUS]ACMS$DIR.ACMS$SAMPLE.DEPARTMENT
SHOW DEFAULT
```

Start the ADUINI.COM file with the **SET VERIFY** command. The **SET VERIFY** command displays any commands or definition clauses that ADU processes from an indirect command file. For example, if you write a menu definition in a source definition file and include the ADU **REPLACE** command, you can submit that file to ADU as an indirect command procedure by using the at sign (@). The **SET VERIFY** command displays each command or clause as it processes that source definition file, as *Example 1.2, "SET VERIFY Display"* shows.

Example 1.2. SET VERIFY Display

```
ADU> @MENU.COM
REPLACE MENU MENU_FIVE
APPLICATION IS APPLONE;
HEADER IS    "P E R S O N N E L   D E P A R T M E N T",
             "                           M E N U";
ENTRIES ARE
       DATR :     TASK IS  DATR;
                  TEXT IS  "Datatrieve";
       DUE :      TASK IS  DUE;
                  TEXT IS  "Display Reviews Due";
END ENTRIES;
END DEFINITION;
```

If you use ADU often, you can also include the **SET DEFAULT** command in your ADUINI.COM file. This command names the CDD directory in which you want ADU commands, such as **CREATE** and **REPLACE**, to put definitions. Because some CDD path names can be long, setting the default directory in the ADUINI.COM file saves you from typing long CDD path names before beginning each utility session.

The **SET DEFAULT** command overrides the CDD default defined by the CDD$DEFAULT logical name. By defining this logical name, you can set your default directory. However, you cannot define CDD$DEFAULT in your ADUINI.COM file or while running ADU. Define CDD$DEFAULT only from DCL command level.

You can include the **SHOW DEFAULT** command in your ADUINI.COM file after the **SET DEFAULT** command. Use the **SHOW DEFAULT** command to see the directory to which your default is set when you start ADU. For example:

```
$ ADU
current CDD default path is 'DISK1:[CDDPLUS]ACMS$DIR.ACMS
$SAMPLE.DEPARTMENT'
ADU>
```

Because the **SET DEFAULT** command is in ADUINI.COM, ADU sets the default to the DISK1:[CDDPLUS]ACMS$DIR.ACMS$SAMPLE.DEPARTMENT directory. With the **SHOW DEFAULT** command in ADUINI.COM, ADU displays the default directory you assigned with the **SET DEFAULT** command *before* it displays the ADU> prompt.

Even if you set the default directory in the ADUINI.COM file, you can always override the default. Use the **SET DEFAULT** command to change the default while you use the utility. Remember that an interactive **SET DEFAULT** command changes the default directory for only the current utility session.

By using **SET LOG**, you can record your use of the utility. The command creates the file ADULOG.LOG if you do not include a file name with the command. With logging in effect, ADU copies the following information to the log file while you run the utility:

- Commands you enter at the ADU> prompt
- Messages that ADU displays on your terminal

The **SET NOLOG** command is the default. Unless you either include the **SET LOG** command in ADUINI.COM or enter the command during a utility session, ADU does not record session information in a log file. If you put in ADUINI.COM either the **SET LOG** command to enable logging or the **SET NOLOG** command to disable logging, you should also include the **SHOW LOG** command, which indicates:

- Whether logging is enabled or disabled
- The name of a log file (if applicable)

1.4. Understanding ACMS Terminology

Several terms are used throughout this manual in a specialized manner. The following sections explain the usage for each of these terms:

- Identifier
- File specification
- Workspace field name
- Text string

1.4.1. Identifier

Several ADU clauses use identifiers in their syntax. **Identifiers** are names you create, such as request library file names and task names. Identifiers can have a maximum of 31 characters, but no spaces. You can use the following characters:

- All letters, including 8-bit alphabetic characters from the DEC Multinational Character Set
- Digits 0 through 9
- Underscores (_)
- Dollar signs ($)

The first character of an identifier must be a letter, a dollar sign, or an underscore. By convention, identifiers containing dollar signs are reserved for VSI use.

Note

ACMS converts all lowercase letters to uppercase.

When creating an identifier, do not use the ACMS reserved words ARE, IS, USING, or WITH. In addition, some commands or clauses have special restrictions for identifiers, such as whether the identifier must be unique within a definition. These restrictions are included with the description of the command or clause.

Identifier is also an OpenVMS term used to describe a special name that a user is allowed to hold. Some identifiers represent the user names and user identification codes (UICs). Others are more general names that a group of users holds. During login, OpenVMS identifiers are copied into a rights list that

becomes part of the OpenVMS process. Access control lists (ACLs) associate identifiers with the type of access to be granted or denied to a system object such as a file or logical name table. The application definition ACCESS clause uses ACLs to grant and deny access to ACMS tasks. For more information about OpenVMS identifiers see the OpenVMS documentation set.

1.4.2. File Specification

A **file specification** can be either an identifier (for logical names) or a quoted string. The contents of the quoted string must be an OpenVMS file specification. A valid OpenVMS file specification can contain all or a subset of the following fields:

```
node"access-string"::device-name:[directory-spec]file-name.file-type;version-number
```

node	Identifies the system on which the file resides. Node names are optional.
access-string	Contains information that enables access to files that are otherwise protected. Access control strings are optional. However, when you include one in a file specification, you must enclose it within quotation marks and precede it with the appropriate node name.
device-name	Identifies the physical device where the file is located. The device name can be the device code or a logical name specifying the device. Device names are optional, but you must include one if you specify a node name.
[*directory-spec*]	Identifies the directory in which the file is located. Directory specifications are enclosed within square brackets ([ ]). Directory specifications are optional, but you must include one if you specify a device name. The device name and the directory specification go together in a single logical name.
file-name	Specifies the name of the file. The file name field is required.
file-type	Specifies the type of file. Always use a period (.) to separate the file type from the file name. File types are optional.
version-number	Specifies which version of the file you want. Always use a semicolon (;) to separate the version number from the file type. Version numbers are optional. When specifying a version number, always specify the file type and file name.

See *VSI OpenVMS DCL Dictionary: A–M* [https://docs.vmssoftware.com/vsi-openvms-dcl-dictionary-a-m/] for information about file specification defaults and default file types.

1.4.3. Workspace Field Name

You refer to a workspace field name by specifying a sequence of identifiers which are separated by periods. A full workspace field name consists of the workspace name followed by the structure names and ending with the name of the elementary data item. Refer to the CDD documentation for more information on structure names.

Any identifiers in the full workspace field name can be omitted, except for the name of the elementary data item, as long as the result is a unique field name. For example, the following is a valid CDO record definition:

```
DEFINE RECORD SALARY_RECORD.
SALARY STRUCTURE.
     EMPLOYEE_ID.
     PAY STRUCTURE.
          JOB_CLASS.
          WEEKLY_PAY.
```

```
     END PAY STRUCTURE.
END SALARY STRUCTURE.
END SALARY_RECORD RECORD.
```

There are four different ways to reference the WEEKLY_PAY field in the SALARY_RECORD example:

- SALARY_RECORD.SALARY.PAY.WEEKLY_PAY
- SALARY.PAY.WEEKLY_PAY
- PAY.WEEKLY_PAY
- WEEKLY_PAY

1.4.4. Text Strings

A text string contains a sequence of characters that are enclosed within single (' ') or double (" ") quotation marks. Text strings can include the following:

- All printable characters, including 8-bit characters from the VSI Multinational Character Set
- Spaces
- Tabs (except in the header of an ACMS menu)

To include quotation marks in a string that is enclosed by the same quotation mark (either single or double), use two quotation marks; for example, "a text string" "with" " embedded quotation marks".

The HEADER clause of the menu definition uses text strings to define the menu title. For example:

```
HEADER "                    ACMS",
        "          EMPLOYEE SAMPLE APPLICATION";
```

The TEXT subclause in the menu definition also uses text strings.

When two or more legal text strings are joined by an ampersand (&), ACMS concatenates the strings and treats the result as a single text string. This technique can be used to break quoted strings that are too long to fit on one line without wrapping. For example, certain SQL Data Manipulation Language (DML) strings do not fit on a single line:

```
PROCESSING WITH
     SQL RECOVERY "SET TRANSACTION READ WRITE " &
                      "RESERVING DEPART, ADMIN FOR PROTECTED WRITE"
```

ACMS concatenates the string into "SET TRANSACTION READ WRITE RESERVING DEPART, ADMIN FOR PROTECTED WRITE" before passing it to Rdb. When using the ampersand character to join lines, be sure to include all necessary spaces. The following example results in a string that cannot be interpreted correctly:

```
BLOCK WITH SQL RECOVERY
     "SET TRANSACTION READ WRITE RESERVING"   &
          "EMPLOYEES FOR SHARED READ,"          &
          "SALARY_HISTORY FOR SHARED WRITE,"    &
          "JOBS FOR EXCLUSIVE WRITE"
```

ACMS concatenates the above example into the string
"SET TRANSACTION READ WRITE RESERVINGEMPLOYEES FOR SHARED READ,

SALARY_HISTORY FOR SHARED WRITE, JOBS FOR EXCLUSIVE WRITE" before passing it to Rdb. To send Rdb the correct string, add spaces before the end quotation marks:

```
BLOCK WITH SQL RECOVERY
     "SET TRANSACTION READ WRITE RESERVING "    &
          "EMPLOYEES FOR SHARED READ, "           &
          "SALARY_HISTORY FOR SHARED WRITE, "     &
          "JOBS FOR EXCLUSIVE WRITE"
```

The new version translates to
"SET TRANSACTION READ WRITE RESERVING EMPLOYEES FOR SHARED READ, SALARY_HISTORY FOR SHARED WRITE, sJOBS FOR EXCLUSIVE WRITE". Rdb can now interpret the example correctly.

1.5. Creating and Processing Definitions

When you use ADU to create an application, first write the definitions that compose the application and then process those definitions. There are four kinds of definitions in an ACMS application: task, task group, application, and menu. In each case, first create the definition of an element and then process it. Either write a definition in a file and then submit the file to ADU for processing, or write a definition interactively in ADU.

In general, it is easier to write a definition in a file and then process it, than to create the definition interactively. The file method offers more control over modifying and reorganizing the definition. Creating a definition file and then processing it is analogous to writing a source file for a computer program and then compiling the source code.

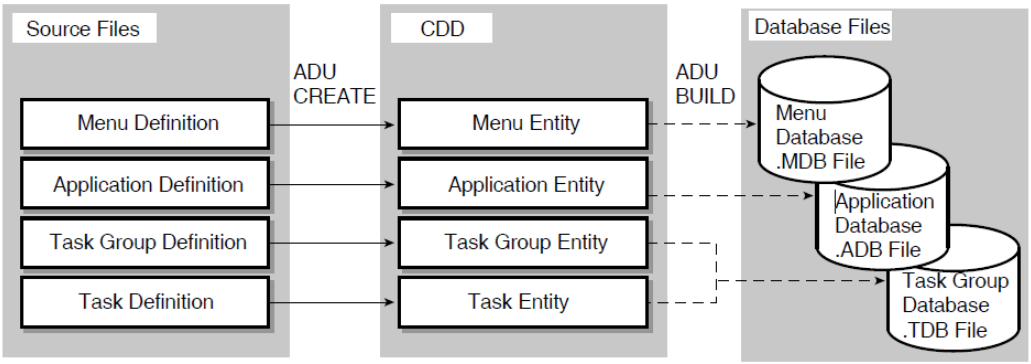
1.6. How ACMS Uses Definitions

When you write the definitions for ACMS tasks, task groups, applications, and menus using ADU, ACMS stores the definitions in CDD. The definitions must then be translated into binary format. At run time, the definitions are represented by databases. For example, a task group definition is represented by a **task group database**, or .TDB, that contains a binary representation of the task group definition, including descriptions of the tasks in the group. Similarly, an application definition is represented by an **application database** (.ADB), and a menu definition is represented by a **menu database** (.MDB).

Use the ADU **`CREATE`** and **`BUILD`** commands to process task group, application, and menu definitions. **`CREATE`** stores the definitions in CDD, and **`BUILD`** creates the databases that ACMS uses at run time.

Figure 1.1, "Creation of Definition Database Files" shows how to process each type of definition to create the three database files.

Figure 1.1. Creation of Definition Database Files



Using the **CREATE** or **REPLACE** command is like compiling a program. Using the **BUILD** command is like linking a program.

The next three sections present an overview of writing ADU definition files and processing those files.

1.6.1. Creating ADU Definition Files

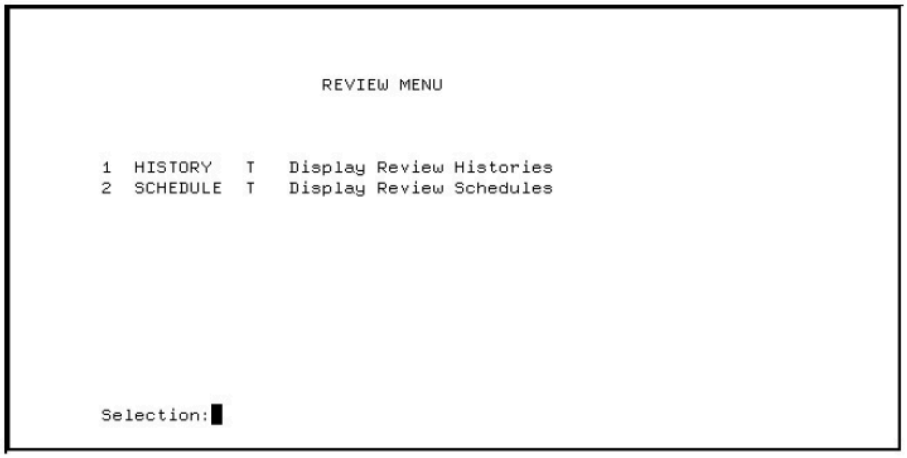
To create ADU definition files, use an OpenVMS text editor (for example, EDT, TPU, or the Language-Sensitive Editor) to enter the definition clauses. At the end of each definition, include the **END DEFINITION** clause. Use a semicolon (;) to mark the end of each clause. Indent the clauses to make it easy to understand the code. Use an exclamation mark (!) to introduce comments.

The following example shows a complete menu definition:

```
! Title for the menu
HEADER IS   "                    REVIEW MENU";
! User Selections
ENTRIES ARE
   HISTORY  : TASK IS REVIEW_HISTORY IN PERSONNEL;
              TEXT IS "Display Review Histories";
   SCHEDULE : TASK IS REVIEW_SCHEDULE IN PERSONNEL;
              TEXT IS "Display Review Schedules";
END ENTRIES;
END DEFINITION;
```

After creating a definition, process the file with the **BUILD** command. *Figure 1.2, "The Review Menu"* shows how the menu appears on the terminal screen after the definition file has been built.

Figure 1.2. The Review Menu



1.6.2. ACMS Definitions: Placing in and Retrieving from CDD

The ADU **REPLACE** command replaces the previous version of a CDD entity that describes an ACMS definition. Either the ADU **REPLACE** or ADU **CREATE** command creates a CDD entity when you first place an ACMS definition in the dictionary:

- **CREATE** – Checks and stores a definition being created for the first time. If the definition already exists, the command fails.

- **REPLACE** – Checks and stores a definition either being created for the first time or replacing a previous version of the definition.

To submit a file containing a task group definition, enter:

```
ADU> CREATE GROUP CUSTOMERS_GROUP CUSTOMERS.GDF/LIST=CUSTOMERS
```

This command tells ADU:

- You want to create a task group definition.
- The CDD path name of the task group definition you are creating is CUSTOMERS_GROUP in your current CDD default directory.
- The file containing the source definition is CUSTOMERS.GDF in your default device and directory. The default file type for a task group source definition file is .GDF.
- You want ADU to create a listing file of the definition. The name of the listing file is CUSTOMERS, with a default file type of .LIS. This listing is much like the listing obtained when you compile a program. The default name of the list file is derived from the full CDD path name. It is located on your default device and in your default directory. In this case, the default list file name is CUSTOMERS.LIS.

The following example uses **REPLACE** instead of **CREATE** to take the task group definition in the file CUSTOMERS. GDF and store the CUSTOMERS_GROUP task group entity in the CDD directory DISK1:[CDDPLUS]AVERTZ. The GROUP keyword indicates that the definition file is for a task group.

```
ADU> REPLACE GROUP DISK1:[CDDPLUS]AVERTZ.CUSTOMERS_GROUP CUSTOMERS.GDF
```

To continue an ADU command on a second line, use the hyphen (-) as the continuation character just for DCL commands. You can abbreviate keywords, such as GROUP, according to the DCL convention.

Instead of entering the **CREATE** or **REPLACE** command at the ADU> prompt each time you process a source definition, you can include the entire command line at the beginning of the source definition file. When including the command line in your source definition files, use the **REPLACE** command instead of the **CREATE** command, as *Example 1.3, "REPLACE Command in a Source Definition File"* shows.

Example 1.3. REPLACE Command in a Source Definition File

```
REPLACE GROUP DISK1:[CDDPLUS]AVERTZ.CUSTOMERS_GROUP
SERVER IS
   CUSTOMER_SERVER : DCL PROCESS;
END SERVER;
TASKS ARE
   ADD  : PROCESSING IS IMAGE IS "SYS$SAMPLE:CUSTOMERS.EXE";
   DATR : PROCESSING IS DCL COMMAND IS "$MCR DTR32";
END TASKS;
END DEFINITION;
```

Use the **REPLACE** command to store in the dictionary a definition that does not already exist or to replace one that does. The **CREATE** command processes only definitions for which no dictionary location exists. Using the **REPLACE** command saves you from having to change the **CREATE** command to **REPLACE** when you change a definition and process it again.

After you include the **REPLACE** command in the source file, you then can use the at sign (@) to submit this file to ADU.

```
ADU> @CUSTOMERS.GDF
```

After using the **CREATE** or **REPLACE** command to create a CDD entity for an ACMS definition, you use the ADU **BUILD** command to build a binary database of each ACMS task group, application, and menu definition for use by ACMS at run time.

In the following example, the ADU **BUILD** command uses the GROUP keyword to take the CDD CUSTOMER task group entity and create the CUSTOMER.TDB task group file. ACMS displays messages indicating the work being done and the size of the file. The resulting .TDB file is located in your default directory.

```
ADU> BUILD GROUP DISK1:[CDDPLUS]AVERTZ.CUSTOMERS_GROUP CUSTOMERS.TDB
```

If the **BUILD** command detects any errors, you can correct them in the definition source file. You must process the revised definition before you reissue the **BUILD** command to create the database file. Use the **REPLACE** command to process and store the revised definition in the same CDD location. If the revised definition has no errors, the **REPLACE** command replaces the old definition with the new version.

You process application and menu definitions the same way as task group definitions. Use the **CREATE** and **REPLACE** commands with task definitions, but not the **BUILD** command. Task definitions are not built; they are included in task group files (.TDBs) when task groups are built.

1.6.3. Annotating Definitions

You can include comments in ADU definitions the same as when writing programs. ADU uses the exclamation point (!) as the comment character. All text to the right of an exclamation point is ignored. Either begin a comment line with an exclamation point or insert an exclamation point at the end of a clause and add a comment on the same line. The following example includes both types of comments:

```
!-------------------------------------------------------------
! Notes included in a definition help document the definition's
! use.  Notes can  make it easier to  maintain the source
| definition file.
!-------------------------------------------------------------
!
! There is one server in this SERVERS clause.
!
SERVERS ARE  ! More servers will be added later.
UTILITY_SERVER   : DCL PROCESS;
                     REUSABLE;
END SERVERS;
!
! There are three tasks in this TASKS clause.
!
TASKS ARE ! More tasks will be added later.
     DATR    : DCL COMMAND IS "$MCR DTR32";
     EDITOR  : DCL COMMAND IS "$EDIT/EDT 'P1'";
     RESTORE : DCL COMMAND IS "$@ACMS$EXAMPLES:RESTORE.COM";
END TASKS;
END DEFINITION;
```

1.6.4. Abbreviating Commands and Keywords

All ADU commands and keywords can be abbreviated to the first four characters. Shorten command names and keywords to fewer characters as long as the abbreviation is unique. For example, abbreviate the **BUILD** command to the letter **B**, because no other ADU command begins with B. The GROUP

keyword can also be abbreviated to a single letter, **G**, because no other keyword begins with G. The following ADU command calls for ACMS to build the group EMPLOYEE_GROUP and assign it the name EMPLOYEE:

```
ADU> B G EMPLOYEE_GROUP EMPLOYEE
```

1.6.5. Using Command Qualifiers

A number of ADU commands include a definition component type as a keyword. These commands include: **BUILD**, **COPY**, **CREATE**, **DELETE**, **LIST**, **MODIFY**, and **REPLACE**. The definition component keywords are APPLICATION, GROUP, MENU, and TASK.

When including qualifiers with these commands, always place the qualifier *after* the keyword. For example:

```
ADU> BUILD GROUP /LOG DISK1:[CDDPLUS]PERSONNEL -
ADU>_ PERSONNEL.TDB
```

You can place the qualifier after other elements in the command line, between parameters, or at the end of the line. For example:

```
ADU> BUILD GROUP DISK1:[CDDPLUS]PERSONNEL /LOG -
ADU>_ PERSONNEL.TDB /PRINT
```

1.6.6. Using the Command Continuation Character

ADU commands can have a maximum of 256 characters. However, only 132 characters can be on a single line. When entering commands interactively, use the command continuation character, the hyphen (-), to enter long commands on several lines. Of course, you do not have to wait until you have entered 132 characters on a line before making a break. Break your command at any convenient place.

When you reach the point for a break on the command line, enter a hyphen and then press **Return**. ADU responds with the continuation prompt, ADU>_. Now continue typing the command line.

The following example breaks the **BUILD** command onto several lines:

```
ADU> BUILD MENU -
DISK1:[CDDPLUS]DALLAS.PERSONNEL.EMPLOYEE_MENU-
ADU>_ MENU.MDB-
ADU>_ /LOG-
ADU>_ /LIST=MENU_DATABASE_BUILD.LIST
```

It is convenient to enter the command qualifiers last, with each one on a separate line, as shown in the previous example.

Note that ACMS concatenates to the contents of the previous line whatever you type in response to the continuation prompt. If elements need to be separated with spaces, always include those spaces at the beginning of the next line. In the preceding example, it was necessary to include a space before the name of the menu database file MENU.MDB. No spaces were needed before the **/LOG** and **/LIST** qualifiers, although inserting spaces before them is permissible. Be sure to insert spaces wherever the syntax requires them.

1.6.7. Responding to Command Prompts

When you enter an ADU command, ADU can supply prompts for the command parameters. This feature is helpful to those not sure of the order of the parameters or not sure of which ones are needed.

To have ADU supply prompts for the required parameters, press **Return** at the end of each input. In the following example, the user entered the **BUILD** command and pressed **Return**. ADU first prompts for the definition component type:

```
ADU> BUILD Return
Component_type    :
```

You can enter the component type and the dictionary path name in response to this prompt, as well as include a database file specification and any desired qualifiers. When you press **Return** after entering the component type, ADU supplies prompts for the dictionary path name.

ADU does not prompt you for the database file specification or for any qualifiers. These elements are not required. Enter the optional elements when responding to a prompt for a required element.

1.6.8. Leaving ADU Temporarily

The ADU **SPAWN** command works the same way as the DCL **SPAWN** command. It allows you to temporarily leave ADU to do other work and then return to the same place. The **SPAWN** command creates a subprocess and attaches the terminal to it. You can then issue other commands to do such things as invoke DCL commands, get information about dictionary objects, or check how something works in DECforms or DATATRIEVE. When you finish that work, you need to log out of the subprocess or use the ADU **ATTACH** command to return to your ADU session. The ADU **ATTACH** command allows you to switch control of your terminal back and forth to previously created processes.

The following command creates the subprocess VIV_1 and transfers control of your terminal to that process:

```
ADU> SPAWN
%DCL-S-SPAWNED, process VIV_1 spawned
```

If you include a command on the line with **SPAWN**, that command is executed and you return to your ADU session. By including the **/NOWAIT** qualifier with the **SPAWN** command, you return to your ADU session immediately. You can resume your ADU session while the subprocess session processes your command.

For more details, see the descriptions of the **SPAWN** and **ATTACH** commands in this manual.

1.6.9. Using ADU Interactively

When you want to use ADU interactively, enter the **CREATE** or **REPLACE** command but do not include a source definition file specification. Then ADU displays the ADUDFN> prompt. For example:

```
ADU> CREATE GROUP CUSTOMERS
ADUDFN>
```

At this prompt, you enter definition clauses, line by line. After entering a clause or part of a clause, you press **Return** again to display the ADUDFN> prompt. As you write the definition, ADU checks each definition clause and displays error messages as it finds them, although ADU might not display an error message until several lines after the line causing the error.

If you write a definition that has errors in it, ADU does not create that definition in the CDD. It does, however, store the definition so that you can edit it with the ADU **EDIT** command. You can also get a copy of the incorrect definition if you enable the logging facility. Logging saves all output that ADU sends to your terminal and keeps a copy of all commands and definitions you enter during one run of the utility. The log file lets you see the definition or definitions you wrote and lets you make changes to

them. As a result of the structure of the log file, you can also use the at sign (@) to process with ADU any definitions included in the log file.

To enable logging, enter in response to the ADU> prompt. You can specify the file you want ADU to use for logging:

```
ADU> SET LOG AUG18.LOG
```

In this example, ADU uses the log file, AUG18.LOG. If you do not specify a device or directory, ADU creates the file in your default directory. If you omit the file specification, ADU creates the file ADULOG.LOG on your default device and in your default directory. You can create an ADUINI.COM file that includes the **SET LOG** command. This enables logging every time you start ADU.

In addition to using **SET LOG** to save definitions that have errors, you can use the **EDIT** or **SAVE** command after you finish writing the definition.

ADU displays errors automatically when you are creating definitions interactively. During processing, use the **SET VERIFY** command. This command lets you see where errors are occurring and thus lets you correct them more easily. You can put the **SET VERIFY** command in the ADUINI.COM file to have ADU keep track of errors automatically during processing.

Refer to the *VSI ACMS for OpenVMS ADU Reference Manual* [https://docs.vmssoftware.com/vsi-acms-adu-ref-manual/] for information about these commands.

1.6.10. Using the Language-Sensitive Editor to Create Definition Files

The Language-Sensitive Editor (LSE) contains five templates used for creating ADU definitions. These templates are especially helpful if you are unfamiliar with ADU syntax or cannot recall the exact definition clause syntax needed. The templates cover all four types of ADU definitions, and they contain online help about ADU phrases and clauses, plus optional diagnostics files. To use LSE, the Language-Sensitive Editor software must be installed on your system.

The ACMS LSE templates supply syntax keywords for inserting directly into the definition file you are creating. After you select the keyword, LSE prompts you for the required parameters. Use the templates to choose syntax paths and to expand subclause entries until they are complete.

LSE generates messages that explain syntax errors in the ADU definitions. When you use the **/DIAGNOSTICS** qualifier with the ADU **CREATE**, **MODIFY**, or **REPLACE** command, ADU outputs these messages to a file that you can review by using the LSE **REVIEW** command. ACMS lets you use the LSE **COMPILE** command from within an LSE editing session to create, modify, or replace a task, task group, application, or menu definition. See *Appendix D, "Using LSE with ACMS"* for information on using the **COMPILE** command.

To invoke LSE at the DCL prompt, issue the **LSEDIT** command followed by a file name with one of the following file extensions:

- .ADF – Invokes LSE with the template for application definitions.
- .GDF – Invokes LSE with the template for task group definitions.
- .TDF – Invokes LSE with the template for task definitions.
- .MDF – Invokes LSE with the template for menu definitions.

- .ADU – Invokes LSE with a template for any of the four preceding ACMS definitions. You can include more than one type of ACMS definition in a file with a .ADU extension.

The following example invokes LSE to create or edit a file called UPDATE_TASK.TDF. The .TDF file type tells LSE to use the task definition template.

```
$ LSEDIT UPDATE_TASK.TDF
```

By default, the ADU **EDIT** and **MODIFY** commands invoke the EDT editor. You can change that default to automatically invoke LSE. The ACMS system logical, ADU$EDIT, points to a DCL command file that ADU runs when you invoke the utility. The command file, in turn, invokes the text editor that the **EDIT** and **MODIFY** commands use. The command file contains the following DCL code:

```
$ ASSIGN/USER 'F$LOGICAL("SYS$OUTPUT")' SYS$INPUT
$ IF P1 .EQS. "" THEN GOTO NOINPUT
$ EDIT /OUTPUT ='P2' 'P1'
$ EXIT
$ NOINPUT:
$ EDIT 'P2'
```

To invoke LSE automatically, replace the previous command lines with the following code:

```
$ ASSIGN/USER 'F$LOGICAL("SYS$OUTPUT")' SYS$INPUT
$ IF P1 .EQS. "" THEN GOTO NOINPUT
$ LSEDIT /LANGUAGE=ACMSADU /OUTPUT ='P2' 'P1'
$ EXIT
$ NOINPUT:
$ LSEDIT /LANGUAGE=ACMSADU 'P2'
```

See *Appendix D, "Using LSE with ACMS"* and *VSI DECset for OpenVMS Language-Sensitive Editor/ Source Code Analyzer Reference Manual* [https://docs.vmssoftware.com/vsi-decset-for-openvms-language-sensitive-editor-source-code-analyzer-reference-manual/] for more information on using LSE. *VSI ACMS Version 5.0 for OpenVMS Installation Guide* [https://docs.vmssoftware.com/vsi-acms-installation-guide/] contains information on installing ACMS with the LSE option.

1.7. Getting Help

When you are running ADU and need information about its commands and clauses, use the ADU **HELP** command. ADU **HELP** works the same way as **HELP** for the OpenVMS operating system. Enter **HELP** in response to the ADU> prompt:

```
ADU> HELP
```

ADU then displays a list of topics on which you can get help. When finished with the help information, press **Return** repeatedly until the ADU> prompt returns. To return immediately to the ADU> prompt, press **Ctrl/Z**

If you know the topic about which you want information, you can enter that topic after the **HELP** command. For example, if you want information on the **CREATE** command and you do not want to see the list of available topics, type:

```
ADU> HELP CREATE
```

ADU displays information about the **CREATE** command, and issues the `CREATE Subtopic?` prompt. At this prompt, you can type PARAMETERS for information about **CREATE** parameters or type **QUALIFIERS** to get information about its qualifiers. If you do not need any further information

about **CREATE**, press **Return** once to return to the TOPIC prompt. Press **Return** twice to return to the ADU> prompt.

To return immediately to the ADU> prompt after obtaining information about a command or clause, use the **/NOPROMPT** qualifier:

```
ADU> HELP/NOPROMPT CREATE
```

ADU then displays information about the **CREATE** command. Rather than display the Topic prompt, however, ADU returns to the ADU> prompt so that you can continue the utility session.

For more information on help systems available in ACMS, consult the Preface.

Chapter 2. Defining Tasks

Chapter 2, "Defining Tasks" explains how you can use ADU syntax to define the components of an ACMS task. The examples in this chapter show tasks that a car rental agency might use in a typical car reservation transaction processing application.

2.1. Structure of an ACMS Task

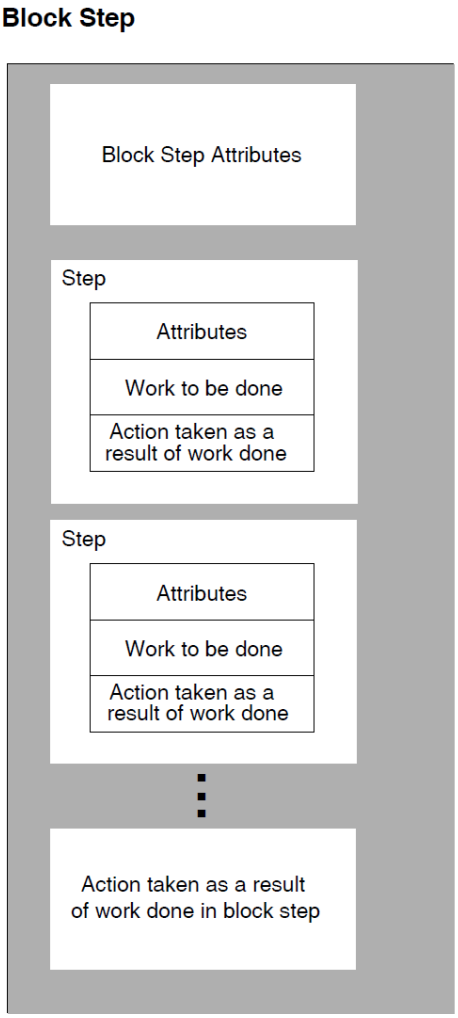
The central part of a task definition is made up of a sequence of exchange and processing steps that are grouped into a **block step**.

In a simple data entry task, there is just one exchange of information with the terminal user and one processing step. Therefore, you define the task to have just two steps within the block step. You can use the ACMS Application Definition Utility (ADU) to define three parts for each block, exchange, and processing step:

- Attributes that describe processing characteristics for the steps in the block
- Work done in the step
- Action taken as a result of that work

The **work** of a block step consists of the exchange and processing steps that it contains. The task itself contains overall characteristics that affect the block step and its work. *Figure 2.1, "Structure of a Task Definition"* shows the structure of a task definition and, in addition to an exchange and processing step, includes a block step that contains attributes and actions taken as a result of the work done.

Figure 2.1. Structure of a Task Definition



2.2. Defining a Data Entry Task

A data entry task can record information about a car reservation, a new part in inventory, or a day's events. This section discusses the case of adding information to a database. The name of this sample task is Add Car Reservation task.

A simple data entry task requires just one exchange step and one processing step. To display errors that might occur during the data entry task, you might want to add another exchange step to the task definition. *Table 2.1, "Data Entry Task"* summarizes the steps and keywords you use to describe each step in the task definition.

Table 2.1. Data Entry Task

Step	Type
Display a form to get information	Exchange step
Write information to the database	Processing step
Display an error message if necessary	Exchange step

To begin the task definition, you can assign names, or **labels**, to these steps. In the Add Car Reservation task, the step labels are GET_RENTAL_INFORMATION, WRITE_RESERVATION_INFORMATION, and ERROR_PROCESS_MSG:

You need a step label only if the step is referred to by name elsewhere in the task definition. However, step labels often help you to identify the function of steps in a definition.

```
GET_RENTAL_INFORMATION:
   EXCHANGE ...
WRITE_RESERVATION_INFORMATION:
   PROCESSING ...
ERROR_PROCESS_MSG:
   EXCHANGE...
```

You can use any step labels you want. A step label begins with an alphabetic character and can contain as many as 31 alphanumeric characters, including dollar signs ($) and underscores (_). A step label cannot contain spaces. You follow the step label with a colon (:) and the definition for that step.

Always use the keywords EXCHANGE and PROCESSING to identify the type of step you are defining. The GET_RENTAL_INFORMATION and the ERROR_PROCESS_MSG steps are exchange steps, and the WRITE_RESERVATION_INFORMATION step is a processing step.

You use the TRANSCEIVE clause in an exchange step to accomplish two operations, a send followed by a receive. First, the TRANSCEIVE clause instructs DECforms to display a panel asking the terminal user for information about the customer's car reservation plans. Then the TRANSCEIVE clause instructs DECforms to return that information in an ACMS workspace.

```
GET_RENTAL_INFORMATION:
   EXCHANGE
     TRANSCEIVE FORM RECORD ADD_RESERVE_FORM_REC, ADD_RESERVE_FORM_REC...
WRITE_RESERVATION_INFORMATION:
   PROCESSING ...
```

In this example, the name of the DECforms form record is ADD_RESERVE_FORM_REC. This is the name of the run-time form record definition in the form; it is not a CDD path name or CDD given name. DECforms uses the form record as a map of form data items and their corresponding workspace fields when transferring data between the form and your task.

You can use a task definition to identify the form in which a form record belongs. If you do not name a form, ACMS uses the first form named in the task group definition. *Chapter 10, "Defining Task Groups"* explains how to define task groups. The *VSI ACMS for OpenVMS ADU Reference Manual* [https://docs.vmssoftware.com/vsi-acms-adu-ref-manual/] explains in detail how to name forms as part of the exchange step.

In DECforms, you use the Form Development Environment (FDE) to create a form. Based on your input, the FDE generates an Independent Form Description Language (IFDL) file. Because of the length of the IFDL code, this chapter does not show the definition of the ADD_RESERVE_FORM that this data entry task uses. When the terminal user fills in the fields of a DECforms panel and presses **Enter**, the DECforms passes the data from the form data items to their corresponding fields in your application workspaces. See *VSI DECforms Guide to Developing an Application* [https://docs.vmssoftware.com/vsi-decforms-for-openvms-guide-to-developing-an-application/] for more information on defining forms.

After a form displays a panel that gets information from the terminal user, the first part of the task is done. To do the second part of the data entry task, you call an ACMS procedure to write that

information to the database. A procedure server contains procedures, or subroutines. You call the procedure in the processing step:

```
GET_RENTAL_INFORMATION:
   EXCHANGE
     TRANSCEIVE FORM RECORD ADD_RESERVE_FORM_REC, ADD_RESERVE_FORM_REC...
WRITE_RESERVATION_INFORMATION:
   PROCESSING
     CALL WRITE_RESERVE_PROC IN RESERVE_SERVER...
  END BLOCK WORK;
END DEFINITION;
```

You use the CALL clause to name the procedure you want to run. In this example, the procedure named WRITE_RESERVE_PROC is the program, entry point, or routine name, such as the Program-ID for a COBOL subprogram; it is not a CDD path name, CDD given name, or file name. *VSI ACMS for OpenVMS Writing Server Procedures* [https://docs.vmssoftware.com/vsi-acms-writing-server-proc/] explains in detail how to write a procedure to run in the processing step of a multiple-step task.

As part of the CALL clause, you also name the server that contains the step procedure to run. When you include the server name, you use the name assigned to the server in the SERVERS ARE clause of the task group definition. This clause is explained in *Chapter 10, "Defining Task Groups"*.

The name of the server that includes the procedure WRITE_RESERVE_PROC is RESERVE_SERVER. If a task contains many processing steps, and they all use the same server, you can define a default server for the task. *VSI ACMS for OpenVMS ADU Reference Manual* [https://docs.vmssoftware.com/vsi-acms-adu-ref-manual/] explains how to define a default server for a task.

2.2.1. Using Workspaces to Pass Data Between Steps

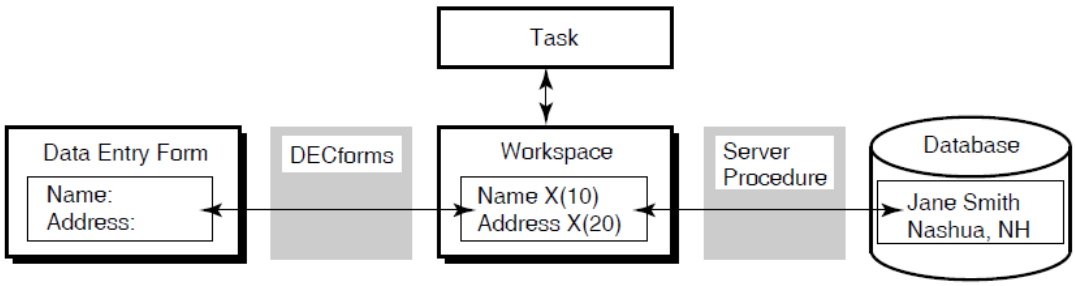
After you get the information from the terminal user, you need to transfer the information to the procedure so that the procedure can write it to the database. This process consists of two steps:

1. You use workspaces to pass the information from the form that collects it.

2. A procedure receives the information from the workspaces and writes it to the database.

A **workspace** is a buffer that the task uses to store information and pass information between steps. Every workspace has a record definition, created by using the Common Dictionary Operator (CDO), that describes the workspace layout. The CDD documentation explains how to use the CDO. In the Add Car Reservation task, you need a workspace to pass the new car rental information from ADD_RESERVE_FORM_REC to the WRITE_RESERVE_PROC procedure. *Figure 2.2, "The Workspace Used to Pass Information"* shows how the form and the procedure use the workspace to pass information.

Figure 2.2. The Workspace Used to Pass Information



Both the form and the procedure in the definition use the workspace: the form stores the information supplied by the terminal user in the workspace, the procedure reads the information from the workspace and writes it to the database. To define a workspace record in CDO, you must define the fields separate from the record. *Example 2.1, "Definition of ADD_RESERVE_WKSP Fields"* shows the definition of the ADD_RESERVE_WKSP workspace.

Example 2.1. Definition of ADD_RESERVE_WKSP Fields

```
Definition of record ADD_RESERVE_WKSP
|   Contains field          CUST_NUMBER
|   |  Datatype                 signed longword
|   Contains field          CUST_NAME
|   |  Datatype                 text size is 30 characters
|   Contains field          CUST_STREET_ADDRESS
|   |  Datatype                 text size is 30 characters
|   Contains field          CUST_CITY
|   |  Datatype                 text size is 20 characters
|   Contains field          CUST_STATE
|   |  Datatype                 text size is 2 characters
|   Contains field          CUST_ZIP
|   |  Datatype                 text size is 5 characters
|   Contains field          CUST_PHONE
|   |  Datatype                 text size is 10 characters
|   Contains field          CAR_TYPE
|   |  Datatype                 text size is 3 characters
|   Contains field          RENTAL_DATE
|   |  Datatype                 text size is 6 characters
|   Contains field          RETURN_DATE
|   |  Datatype                 text size is 6 characters
```

When you define the ADD_RESERVE_FORM form using DECforms, the form data items must correspond to the fields in the ADD_RESERVE_WKSP definition. At run time, DECforms uses the form record to map form data items to workspace fields. The easiest way to define the form record in your IFDL code is to include a statement that instructs DECforms to copy the workspace definition from the CDD dictionary into the IFDL file:

```
COPY
   DISK1:[CDDPLUS]ACMS$DIR.ACMS$EXAMPLES_RMS.ADD_RESERVE_WKSP
         FROM DICTIONARY
END COPY
```

A workspace name can be different from the name of the record in the record definition for that workspace. For example, the record name in *Example 2.1, "Definition of ADD_RESERVE_WKSP Fields"* does not have to be ADD_RESERVE_WKSP. See the *VSI ACMS for OpenVMS ADU Reference Manual* [https://docs.vmssoftware.com/vsi-acms-adu-ref-manual/] for more information on workspace names and resolution of workspace field names.

Because both the exchange step and the processing step use the ADD_RESERVE_WKSP workspace, you must include the workspace name in both the TRANSCEIVE and PROCEDURE clauses in the task definition.

The definition for the record layout of a workspace is stored in the CDD dictionary. When you first declare a workspace in a task definition, you must use the CDD path name of the record definition for that workspace. When referring to the workspace from within the definition, you use the given name or unique name of the workspace specified by the WORKSPACES ARE clause.

Suppose that the full CDD path name of the record description for the workspace is DISK1:[CDDPLUS]ACMS$DIR.ACMS$EXAMPLES_RMS.ADD_RESERVE_WKSP. When you are writing step definitions, you use only the given name of the record description for a workspace, ADD_RESERVE_WKSP. For example:

```
GET_RENTAL_INFORMATION:
   EXCHANGE
     TRANSCEIVE FORM RECORD ADD_RESERVE_FORM_REC, ADD_RESERVE_FORM_REC
      SENDING ADD_RESERVE_WKSP
      RECEIVING ADD_RESERVE_WKSP;
WRITE_RESERVATION_INFORMATION:
   PROCESSING
     CALL WRITE_RESERVE_PROC IN RESERVE_SERVER
      USING ADD_RESERVE_WKSP;
END BLOCK WORK;
```

This definition assigns the ADD_RESERVE_WKSP workspace for use by both the form and the procedure. The definition of the form record ADD_RESERVE_FORM_REC must correspond with the record definition of the ADD_RESERVE_WKSP workspace.

You may want to use the same record definition for more than one workspace and assign different workspace names to that record definition. When you first declare the workspace in the task definition, you can assign a unique name to that workspace. For example:

```
WORKSPACE IS DISK1:[CDDPLUS]ACMS$DIR.ACMS$EXAMPLES_RMS.ADD_RESERVE_WKSP
   WITH NAME ADD_RESERVE_WKSP_1;
```

To refer to this workspace from within the task definition, you use the unique name ADD_RESERVE_WKSP_1:

```
GET_RENTAL_INFORMATION:
   EXCHANGE
     TRANSCEIVE FORM RECORD ADD_RESERVE_FORM_REC, ADD_RESERVE_FORM_REC
      SENDING ADD_RESERVE_WKSP_1
      RECEIVING ADD_RESERVE_WKSP_1;
WRITE_RESERVATION_INFORMATION:
   PROCESSING
     CALL WRITE_RESERVE_PROC IN RESERVE_SERVER USING ADD_RESERVE_WKSP_1;
END BLOCK WORK;
```

The USING keyword names the workspace or workspaces that you want the form and the procedure to use.

2.2.2. Defining the Block Step

When you define more than one step to do work for a task, you group those steps into a **block step**. A block step has four parts:

- Attributes
- Work
- Action

- Exception handler

Chapter 8, "Handling Task Execution Errors" describes how to use exception handlers.

In the attributes part of the block step, you must indicate that the task uses DECforms to interface with the terminal by adding the keywords FORM I/O to the definition.

The work part of a block step consists of the exchange and processing steps you define for a task. The definition for the Add Car Reservation task, including the minimum block step syntax, follows:

```
BLOCK WORK
   WITH FORM I/O
   GET_RENTAL_INFORMATION:
     EXCHANGE
       TRANSCEIVE FORM RECORD ADD_RESERVE_FORM_REC, ADD_RESERVE_FORM_REC
        SENDING ADD_RESERVE_WKSP
        RECEIVING ADD_RESERVE_WKSP;
   WRITE_RESERVATION_INFORMATION:
     PROCESSING
       CALL WRITE_RESERVE_PROC IN RESERVE_SERVER
        USING ADD_RESERVE_WKSP;
   END BLOCK WORK;
```

You use the BLOCK clause to start the work for the task. The END BLOCK WORK keywords indicate the end of that work. Include a semicolon (;) after the END BLOCK WORK keywords.

2.2.3. Defining Characteristics of the Task

A task definition must also describe the task characteristics. You use the task part of a definition to set up characteristics for the block step and for steps within the block step.

The most important characteristic of a simple data entry task is the workspace or workspaces used by the steps in the task. You use the WORKSPACES clause to name the workspace or workspaces used by the steps in the task. You must use the CDD path name of the record description for each workspace you name. The CDD given names of the path names you declare must match the given names of the workspaces referred to by the TRANSCEIVE and CALL clauses in the task definition.

The sample step definition uses only one workspace, ADD_RESERVE_WKSP. Suppose that the definition for this workspace is in the ACMS$EXAMPLES_RMS directory in the ACMS$DIR directory and that the anchor is in DISK1:[CDDPLUS]. The corresponding WORKSPACE clause looks like this:

```
WORKSPACE IS DISK1:[CDDPLUS]ACMS$DIR.ACMS$EXAMPLES_RMS.ADD_RESERVE_WKSP;
```

You must end the clause with a semicolon (;). If you set your CDD default to DISK1:[CDDPLUS]ACMS$DIR.ACMS$EXAMPLES_RMS when you build the definition, you can use just the given name of the record description for the workspace:

```
WORKSPACE IS ADD_RESERVE_WKSP;
```

If many programmers are working on a project, you might want to use the full path name, rather than the given name, because different programmers might use different CDD defaults.

Now the definition looks like this:

```
WORKSPACE IS ADD_RESERVE_WKSP;
 BLOCK
   WORK WITH FORM I/O
```

```
   GET_RENTAL_INFORMATION:
     EXCHANGE
       TRANSCEIVE FORM RECORD ADD_RESERVE_FORM_REC, ADD_RESERVE_FORM_REC
        SENDING ADD_RESERVE_WKSP
        RECEIVING ADD_RESERVE_WKSP;
   WRITE_RESERVATION_INFORMATION;
     PROCESSING
       CALL WRITE_RESERVE_PROC IN RESERVE_SERVER USING ADD_RESERVE_WKSP;
   END BLOCK WORK;
END DEFINITION;
```

You must end every definition with the END DEFINITION keywords and a semicolon (;).

ACMS requires that the name of each workspace be unique within the workspace declarations for each task group. If two or more given names of record path names are identical, you must assign a unique name with the keyword WITH NAME. For example, to assign the unique name ADD to the ADD_RESERVE_WKSP workspace, you use:

```
WORKSPACE IS ADD_RESERVE_WKSP WITH NAME ADD;
```

Use the WITH NAME keyword carefully. In general, if you need to use the same record definition, under different names, in multiple parts of your application, it is easier to maintain the application if you assign unique workspace names than if you use WITH NAME.

2.2.4. Storing a Task Definition in the Dictionary

Once you have written the task definition, you can use either the ADU **CREATE** or ADU **REPLACE** command to store that definition in the dictionary.

When you use the **CREATE** or **REPLACE** command, you include:

- The kind of definition you are creating
- Where in the dictionary you want to store the definition
- The name of the file containing the source definition

For example:

```
ADU>CREATE TASK ADD_CAR_RESERVATION_TASK ADDCAR.TDF
```

This **CREATE** command processes the definition in the file ADDCAR.TDF. If there are no errors in the definition, ADU stores it in the dictionary in the default directory defined by CDD$DEFAULT.

You can insert the **REPLACE** command at the beginning of the source definition file and submit it to ADU as a command file. When you submit definitions as command files, use the **REPLACE** command rather than the **CREATE** command to save yourself the effort of changing the command when you resubmit the file to ADU.

Example 2.2, "Contents of a Source Definition File" shows the contents of the source definition file ADDCAR.TDF. This command file includes the **REPLACE** command and all the clauses used in this chapter to build the source definition.

Example 2.2. Contents of a Source Definition File

```
SET VERIFY
REPLACE TASK ADD_CAR_RESERVATION_TASK
   WORKSPACE IS ADD_RESERVE_WKSP;
```

```
   BLOCK
     WORK WITH FORM I/O

   GET_RENTAL_INFORMATION:
     EXCHANGE
       TRANSCEIVE FORM RECORD ADD_RESERVE_FORM_REC, ADD_RESERVE_FORM_REC
        SENDING ADD_RESERVE_WKSP
        RECEIVING ADD_RESERVE_WKSP;
   WRITE_RESERVATION_INFORMATION:
     PROCESSING
       CALL WRITE_RESERVE_PROC IN RESERVE_SERVER
        USING ADD_RESERVE_WKSP;
   END BLOCK WORK;
END DEFINITION;
```

You can include the **SET VERIFY** command with a source definition you are submitting as a command file. This command displays each line of the definition as it is processed by ADU, letting you see where errors in the definition occur.

To submit the definition to ADU as a command file, use the at sign character (@) followed by the file name:

```
ADU>@ADDCAR.COM
```

When you submit the command file, ADU checks the source definition file and returns any errors to the terminal. If errors occur, you can edit the source definition file to correct the definition and resubmit the file in the same manner.

Note

If you use the **CREATE** or **REPLACE** command at the ADU prompt to process a definition file, the definition must *not* contain the CREATE or REPLACE clause, or else ADU returns an error.

If you insert the CREATE or REPLACE clause in the definition file, you must submit it to ADU as a command file.

2.2.5. Additional Considerations: Error Handling and Ease of Use

So far, the sample task definition does not handle two considerations that you want to address when solving most business problems: handling processing errors and making the task easy to use.

A complete and realistic version of a data entry task makes provision for errors by taking the following actions:

1. Get information.
2. Allow the user to type a key or combination of keys to end the task instead of completing the first form.
3. Write information to the database.
4. If the processing is successful, let the user repeat the same task without going back to the selection menu. If a user-related or recoverable error occurs, save the input data, tell the user about the error, and let the user correct the error. If a nonrecoverable error occurs, cancel the task.

Once you have broken the problem down into these parts, you can begin putting together the definition. To handle errors that might occur in the processing step, you might need to add another exchange step to the task definition. To handle errors and special conditions in both the exchange and processing steps, you also might need to increase the number of workspaces and change some task characteristics.

2.2.5.1. Using ACMS Workspaces

When you want to handle errors in a task, you test the contents of a field in a workspace and take action based on the contents. For example, a procedure can return a value to a field in a workspace. You can use the CONTROL FIELD clause to test the contents of that field. Then you use action clauses to take action based on those contents. In addition to the CONTROL FIELD clause, ACMS provides three other **conditional clauses** for testing workspace fields:

- IF THEN ELSE
- SELECT FIRST
- WHILE DO

The CONTROL FIELD clause can test the contents of a control field in either of two workspaces:

- An ACMS system workspace, especially ACMS$PROCESSING_STATUS
- A workspace you define

You can use any of the ACMS system workspaces with a conditional clause. However, the ACMS$PROCESSING_STATUS system workspace is especially useful for handling results of procedures in processing steps. Workspaces you define are especially useful for handling information passed to a form.

There are three ACMS system workspaces; each workspace handles a different kind of information. The *VSI ACMS for OpenVMS ADU Reference Manual* [https://docs.vmssoftware.com/vsi-acms-adu-ref-manual/] lists all the system workspaces and gives a brief description of each.

The first exchange step in the Add Car Reservation task uses the CONTROL FIELD clause to test a workspace that you define, while the processing step uses the IF THEN ELSE clause to test a system workspace field.

The ACMS$PROCESSING_STATUS system workspace has four fields:

- ACMS$L_STATUS
- ACMS$T_SEVERITY_LEVEL
- ACMS$T_STATUS_TYPE
- ACMS$T_STATUS_MESSAGE_LONG or ACMS$T_STATUS_MESSAGE

All processing work returns a final status value. For example, when a procedure exits, ACMS places its return status in the ACMS$L_STATUS field of the ACMS$PROCESSING_STATUS workspace. ACMS translates that value and performs the following operations:

1. Stores in the ACMS$T_SEVERITY_LEVEL field a single character string indicating the severity level of the error. These characters are:

- S – SUCCESS
- I – INFORMATION
- W – WARNING
- E – ERROR
- F – FATAL

If the error returned by the procedure does not match any of these error severities, ACMS stores a question mark (?) in the ACMS$T_SEVERITY_LEVEL field.

2. Stores in the ACMS$T_STATUS_TYPE field a single character string indicating whether the severity level of the error is good or bad. These characters are:

   - G – GOOD
   - B – BAD

   If the severity level of the return status of the procedure is SUCCESS or INFORMATION, ACMS stores a G in the ACMS$T_STATUS_TYPE field. If the severity level is WARNING, ERROR, or FATAL, ACMS stores a B in that field.

ACMS can also use the return status value in the ACMS$L_STATUS field to get an error message from a message file. By default, it stores that message in the ACMS$T_STATUS_MESSAGE_LONG field. To retrieve the error message, you must use the GET ERROR MESSAGE clause, as explained later in this chapter.

Whether the conditional clause tests a field in a system workspace or in a workspace you define, the values it tests must be literal strings and the datatype of the field must be text. *Table 2.2, "Field and Values Tested by Conditional Clauses"* summarizes the fields and values that conditional clauses can test.

Table 2.2. Field and Values Tested by Conditional Clauses

Workspace	Field	Value
User-defined	Any	Quoted string
ACMS$PROCESSING_STATUS	ACMS$T_SEVERITY_LEVEL	S,I,W,E,F,?
ACMS$PROCESSING_STATUS	ACMS$T_STATUS_TYPE	G,B

The initial value of the ACMS$T_SEVERITY_LEVEL workspace is S. The initial value of the ACMS$T_STATUS_TYPE field is G.

2.2.5.1.1. Using the CONTROL FIELD Clause in an Exchange Step

One special condition you want to allow for is letting the terminal user press a key to stop running the task. The form stores a value associated with that key in a workspace field. You can then use the CONTROL FIELD clause to test the contents of that field.

In the Add Car Reservation task, you want to let the user stop running the task when the form displays a panel asking for rental information. To do this, you need to define a key, such as the PF4 key, as the exit or quit key. DECforms refers to such a key as a **function key**. In your form definition, you must make the function declaration after the LAYOUT specification. In the following example, the function is PF4 and its name is QUIT_KEY.

```
FORM ADD_RESERVE_FORM
  .
  .
  .
    Layout VT_LAYOUT
         .
         .
         .
        Size 24 lines by 80 columns

        Function QUIT_KEY
            is %PF4
        End Function
```

You must also declare a **function response** in the IFDL source file. A function response alters the way DECforms processes the form. In this example, when the terminal user presses **PF4**, the function response directs DECforms to return the value "QUIT" to an ACMS workspace. Place the function response at the beginning of the panel declaration.

```
 Panel ADD_RESERVE_PANEL
     Function Response QUIT_KEY
          Let QUIT_KEY = "QUIT"
          Return Immediate
     End Response
```

For more information about declaring function responses, see *VSI DECforms Guide to Developing an Application* [https://docs.vmssoftware.com/vsi-decforms-for-openvms-guide-to-developing-an-application/].

In your exchange step, you need to identify the workspace where DECforms stores the value "QUIT". In the example below, the QUIT_KEY field is in the QUIT_CTRL_WKSP workspace. The form record used for the receive part of the exchange is a record list. ADD_RESERVE_FORM_REC_LIS contains form records for ADD_RESERVE_WKSP and QUIT_CTRL_WKSP. Then, in the action part of the exchange step, you can use the CONTROL FIELD clause to test the workspace field. For example:

```
EXCHANGE
   TRANSCEIVE FORM RECORD ADD_RESERVE_FORM_REC,
                          ADD_RESERVE_FORM_REC_LIS
    SENDING ADD_RESERVE_WKSP
    RECEIVING ADD_RESERVE_WKSP, QUIT_CTRL_WKSP;
 ACTION IS
    CONTROL FIELD QUIT_CTRL_WKSP.QUIT_KEY
    "QUIT"  :  EXIT TASK;
    END CONTROL FIELD;
```

The CONTROL FIELD clause tests the QUIT_KEY field of QUIT_CTRL_WKSP. When you use the CONTROL FIELD clause, you can name the workspace and the workspace field, or just the workspace field. ACMS checks all of the workspaces defined for the task until it finds the field named in the CONTROL FIELD clause. However, including the workspace name is a good way to keep track of the location of information that the task uses. When you name the workspace in the CONTROL FIELD clause, use a period to separate it from the name of the control field.

If the value "QUIT" is in the QUIT_KEY field, ACMS exits or stops processing the task. Otherwise, ACMS goes on to process the next step in the definition. You must end each action clause, such as EXIT TASK, with a semicolon (;). End the CONTROL FIELD clause with the keywords END CONTROL FIELD and a semicolon (;).

When ACMS processes the EXIT TASK clause, it ends the task and returns the user to a selection menu without returning a message to the user. You can also use the CANCEL TASK clause to end a task. When ACMS processes the CANCEL TASK clause, it records the ending of the task as an abnormal ending or interruption, and returns a message to the user before returning the user to a selection menu.

In general, you use the EXIT TASK clause if the user wants to end the task and if there is no chance that data will be left in an inconsistent state. You use the CANCEL TASK clause if there is an abnormal reason for ending the task.

2.2.5.1.2. Using the IF THEN ELSE Clause in a Processing Step

Suppose the user does not cancel the task in the first step of the task but types information and presses **Return** or **Enter**. In this case, the processing step calls a procedure to write that information to a file. This procedure can encounter errors.

You want the task to take different actions depending on the kind of error encountered. There are two kinds of errors: recoverable errors and nonrecoverable errors. Recoverable errors are those a user can correct, such as typing the wrong customer number. Nonrecoverable errors are those a user cannot correct. For example, "file not found" is a nonrecoverable error in the Add Car Reservation task.

When a procedure encounters a recoverable error, you can tell the user about the error and let the user do something to correct the error. In the case of a nonrecoverable error, you generally want the procedure to cancel the task.

When a procedure runs, it returns a status value to ACMS. ACMS puts this value in the ACMS$L_STATUS field of the ACMS$PROCESSING_STATUS system workspace. ACMS translates the return status value and stores, in the ACMS$T_SEVERITY_LEVEL field, a value indicating the severity level of the error. ACMS also stores in the ACMS$T_STATUS_TYPE field a GOOD or BAD value. You can use the IF THEN ELSE clause to test the contents of the ACMS$T_STATUS_TYPE field.

ACMS can also use the return status value in ACMS$L_STATUS field to retrieve an error message from a message file and then store that message in the ACMS$T_STATUS_MESSAGE field.

Suppose now that WRITE_RESERVE_PROC tries to write a reservation record to a file, but a record for that customer already exists. This is a recoverable error because the user can try a different customer number or enter information for a different customer. In this case, you want to tell the user about the error and let the user try again. Here is the processing step of the Add Car Reservation task:

```
PROCESSING
   CALL WRITE_RESERVE_PROC IN RESERVE_SERVER
     USING ADD_RESERVE_WKSP;
   IF (ACMS$T_STATUS_TYPE EQ "B")
   THEN  GET ERROR MESSAGE;
         MOVE ACMS$T_STATUS_MESSAGE TO MSG_WKSP.MESSAGE_PANEL;
         GOTO STEP ERROR_PROCESS_MSG;
   ELSE  GOTO PREVIOUS EXCHANGE;
   END IF;
```

In this step, WRITE_RESERVE_PROC tries to write the information in the ADD_RESERVE_WKSP workspace to a file. It returns a status value to the ACMS$L_STATUS field of the ACMS$PROCESSING_STATUS workspace. ACMS translates that value and stores a B or G in the ACMS$T_STATUS_TYPE field. If the record already exists, the value in that field is a B.

The IF THEN ELSE clause tests a Boolean expression to determine which course of action to follow. You must enclose the Boolean expression within parentheses. See the *VSI ACMS for OpenVMS ADU*

Reference Manual [https://docs.vmssoftware.com/vsi-acms-adu-ref-manual/] for more information about using Boolean expressions. If the Boolean expression evaluates to true, ACMS performs the actions associated with the THEN keyword. In this example, if the record already exists, ACMS performs the following steps:

1. Retrieves the error message from a message file
2. Stores the error message in the MESSAGE_PANEL field of the MSG_WKSP workspace
3. Goes to the ERROR_PROCESS_MSG exchange step in the task

To display the error message to the terminal user, you need to include the following exchange step in your task definition:

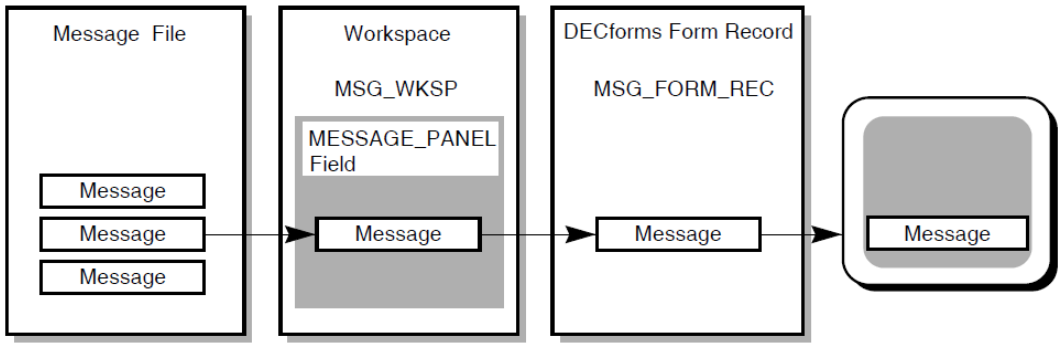
```
ERROR_PROCESS_MSG:
  EXCHANGE WORK
   SEND FORM RECORD MSG_FORM_REC
    SENDING MSG_WKSP;
```

This exchange step sends the error message stored in the MSG_WKSP workspace to the MSG_FORM_REC form record for display. The IFDL file for the ADD_RESERVE_FORM form must include the form record definition for MSG_FORM_REC.

If the Boolean expression evaluates to false, ACMS performs the actions associated with the ELSE keyword. In this example, the GOTO PREVIOUS EXCHANGE clause directs ACMS to repeat the task. You must end the IF THEN ELSE clause with the END IF keywords and a semicolon (;).

Figure 2.3, "Retrieving Messages" shows the process of retrieving and displaying error messages.

Figure 2.3. Retrieving Messages



Although you can use action clauses in any order you want, ACMS always processes them in the same order. For example, ACMS always processes the GET ERROR MESSAGE clause before any sequencing clauses, such as GOTO PREVIOUS EXCHANGE.

For more information on returning status and using the system workspace ACMS$PROCESSING_STATUS, see *VSI ACMS for OpenVMS Writing Server Procedures* [https://docs.vmssoftware.com/vsi-acms-writing-server-proc/].

2.2.5.1.3. Additional Workspace Definitions

Because the error handling described in the preceding sections named new workspaces, you must enter these definitions into the dictionary. In the first exchange step, the CONTROL FIELD clause tests the QUIT_KEY field of the QUIT_CTRL_WKSP workspace.

In the processing step, the IF THEN ELSE clause uses an ACMS system workspace and the MSG_WKSP workspace. You do not need to do anything with the system workspace, but you have to enter the MSG_WKSP definition into the dictionary.

In your task definition, you must include QUIT_CTRL_WKSP and MSG_WKSP in the WORKSPACES clause.

2.2.5.2. Repeating the Task Automatically

In a data entry task, you frequently do not want to redisplay the menu each time the user adds a reservation record or encounters an error. Instead, you want to repeat the task automatically, letting the user return to the menu by pressing a function key such as **PF4**.

You can describe this characteristic by using the REPEAT TASK clause when you define the actions for the block step. You can include the optional ACTION keyword to distinguish the action part of the block definition from the work part of the definition.

Example 2.3, "Complete Definition for the Add Car Reservation Task" shows the complete definition for the new version of the Add Car Reservation task.

Example 2.3. Complete Definition for the Add Car Reservation Task

```
REPLACE TASK ADD_CAR_RESERVATION_TASK
WORKSPACES ARE ADD_RESERVE_WKSP, QUIT_CTRL_WKSP, MSG_WKSP;

BLOCK WORK WITH FORM I/O IS

  GET_RENTAL_INFORMATION:
   EXCHANGE WORK IS
    TRANSCEIVE FORM RECORD ADD_RESERVE_FORM_REC,
                           ADD_RESERVE_FORM_REC_LIS
     SENDING ADD_RESERVE_WKSP
     RECEIVING ADD_RESERVE_WKSP, QUIT_CTRL_WKSP;
   ACTION IS
    CONTROL FIELD IS QUIT_CTRL_WKSP.QUIT_KEY
    "QUIT"  :  EXIT TASK;
    END CONTROL FIELD;

  WRITE_RESERVATION_INFORMATION:
   PROCESSING WORK IS
    CALL WRITE_RESERVE_PROC IN RESERVE_SERVER
    USING ADD_RESERVE_WKSP;
   ACTION IS
    IF (ACMS$T_STATUS_TYPE EQ "B")
    THEN GET ERROR MESSAGE;
         MOVE ACMS$T_STATUS_MESSAGE TO MSG_WKSP.MESSAGE_PANEL;
         GOTO NEXT EXCHANGE;
    ELSE GOTO PREVIOUS EXCHANGE;
    END IF;

  ERROR_PROCESS_MSG:
   EXCHANGE WORK IS
    SEND FORM RECORD MSG_FORM_REC
    SENDING MSG_WKSP;

END BLOCK WORK;
ACTION IS
```

```
  REPEAT TASK;

END DEFINITION;
```

2.3. Defining an Inquiry Task

A task displaying information from a file has many of the same characteristics as one adding information to a file. This section presents an inquiry task, the Review Car Rates task, that displays rental rates for particular types of cars. *Table 2.3, "Inquiry Task"* lists the steps that an inquiry task performs.

Table 2.3. Inquiry Task

Step	Type
Display a form to get information	Exchange step
Read information from a file	Processing step
Display an error message if necessary	Exchange step
Display a form with information	Exchange step

2.3.1. Getting Information from the User

The first step of both data entry and inquiry tasks involves getting information from the user. In the Review Car Rates task, the terminal user types in a code indicating the class of car (compact, midsize, and so on) that the customer wants to rent and presses **Return** or **Enter** to proceed to the next step. Otherwise, the terminal user can press **PF4**to exit from the task and return to the menu.

The first step of the Review History inquiry task looks like this:

```
DETERMINE_RENTAL_CLASS:
   EXCHANGE
    TRANSCEIVE FORM RECORD RENTAL_CLASSES_FORM_REC,
                           RENTAL_CLASSES_FORM_REC_LIS
     SENDING RENTAL_CLASSES_WKSP
     RECEIVING RENTAL_CLASSES_WKSP, QUIT_CTRL_WKSP;
   ACTION
    CONTROL FIELD IS QUIT_CTRL_WKSP.QUIT_KEY
    "QUIT"  :  EXIT TASK;
    END CONTROL FIELD;
```

Because both the DETERMINE_RENTAL_CLASS step and the first step in the data entry task get information from the user, their definitions are almost identical. Both steps do the following:

- Use a TRANSCEIVE clause to display a form asking for information
- Let the user exit directly from the step
- Use workspaces to pass control information and data

The definitions of the workspaces for the data entry and inquiry tasks are similar also. *Example 2.4, "Definition for RENTAL_CLASSES_WKSP"* shows the definition for RENTAL_CLASSES_WKSP that the first step of the Review Car Rates task uses.

Example 2.4. Definition for RENTAL_CLASSES_WKSP

```
Definition of record RENTAL_CLASSES_WKSP
```

```
|   Contains field          COUNTRY_ID
|   |  Datatype                 signed longword
|   Contains field          REQUESTED_RENTAL_CLASS_ID
|   |  Datatype                 text size is 2 characters
|   Contains field          DAY_RENTAL_RATE_AMT
|   |  Datatype                 text size is 5 characters
|   Contains field          WEEK_RENTAL_RATE_AMT
|   |  Datatype                 text size is 7 characters
|   Contains field          MONTH_RENTAL_RATE_AMT
|   |  Datatype                 text size is 7 characters
```

As in the data entry task, you can use the DECforms COPY statement in the RENTAL_CLASSES_FORM IFDL code to create the RENTAL_CLASSES_FORM_REC form record that corresponds to the application workspace.

2.3.2. Retrieving Information from a File

The second part of data entry and inquiry involves interaction with a file. You accomplish this interaction by using a procedure in a processing step. In a data entry task, a procedure writes data to a file; in an inquiry task, a procedure reads data from a file.

You must consider errors that can occur when a procedure tries to read from a file. In the Review Car Rates task, a procedure reads from the Rates file, using the rental class code as a key in RENTAL_CLASSES_WKSP. The procedure can encounter the following errors:

- The record does not exist.
- The record is locked by another user.

Both of these errors are recoverable; the user can type a new number if the record did not exist or can retry the inquiry if the record was locked. Any other errors in reading the record are treated as nonrecoverable, and the procedure cancels the task.

Both recoverable errors cause the procedure to return an error code with a severity level of WARNING or ERROR. Both of these severity levels are handled by the value B (BAD) in the ACMS$T_STATUS_TYPE field in the workspace ACMS$PROCESSING_STATUS. Therefore, you can use the IF THEN ELSE clause to test the contents of that field and handle errors for the task.

Because the second step of an inquiry task and the second step of a data entry task perform processing, both steps are very similar.

```
GET_RENTAL_RATES:
   PROCESSING
     CALL GET_RATES_PROC IN RENTAL_SERVER
       USING RENTAL_CLASSES_WKSP;
   ACTION IS
     IF (ACMS$T_STATUS_TYPE EQ "B")
     THEN GET ERROR MESSAGE;
          MOVE ACMS$T_STATUS_MESSAGE TO MSG_WKSP.MESSAGE_PANEL;
          GOTO NEXT EXCHANGE;
     ELSE GOTO STEP DISPLAY_RENTAL_RATES;
     END IF;
```

Both steps do the following:

- Call a procedure to read from or write to a file

- Use information stored in a workspace
- Test the ACMS$T_STATUS_TYPE field of the ACMS$PROCESSING_STATUS system workspace for the severity level of errors returned by the procedure
- Go to an exchange step that displays an error message, if the value of ACMS$T_STATUS_TYPE is B

If ACMS evaluates the Boolean expression in the IF THEN ELSE clause to be false, it performs the action associated with the ELSE keyword. In this case, it passes control to the DISPLAY_RENTAL_RATES exchange step.

For the processing step of the Review Car Rates task, the procedure GET_RATES_PROC in RENTAL_SERVER uses the rental class code stored in RENTAL_CLASSES_WKSP to read a record from the Rates file. The procedure returns a status value to the ACMS$PROCESSING_STATUS system workspace. ACMS uses the severity level of the status to return either a B or G to the ACMS$T_STATUS_TYPE field. If the value in that field is B, ACMS retrieves an error message from an error message file and passes control to the NO_RC_MSG exchange step, which displays the error message on the terminal screen.

```
NO_RC_MSG:
   EXCHANGE
     TRANSCEIVE FORM RECORD MSG_FORM_REC, QUIT_CTRL_FORM_REC
      SENDING MSG_WKSP
      RECEIVING QUIT_CTRL_WKSP;
   ACTION IS
     IF (QUIT_CTRL_WKSP.QUIT_KEY EQ "QUIT")
     THEN EXIT TASK;
     ELSE GOTO STEP DETERMINE_RENTAL_CLASS;
     END IF;
```

As in the data entry task, you must define the additional workspaces that you use for error handling, QUIT_CTRL_WKSP and MSG_WKSP. Because the NO_RC_MSG exchange step sends the error message from MSG_WKSP to the form, you also need to include the form record definition for MSG_FORM_REC in the IFDL code. QUIT_CTRL_FORM_REC is the form record that corresponds to the QUIT_CTRL_WKSP workspace.

The NO_RC_MSG exchange step uses an IF THEN ELSE clause in the action part of the step to test whether or not the terminal user wants to exit from the task. If the user does not want to exit, ACMS passes control to the DETERMINE_RENTAL_CLASS exchange step so that the user can enter another rental class code.

2.3.3. Displaying Information

In a data entry task, once a procedure has written the information to a file, the task is complete. For an inquiry task, however, once the procedure has read information from a file, you must display that information to the terminal user.

As in the first exchange step of both the data entry and inquiry tasks, you can let the user press a function key to exit the task.

```
DISPLAY_RENTAL_RATES:
   EXCHANGE WORK IS
    TRANSCEIVE FORM RECORD RENTAL_CLASSES_FORM_REC, QUIT_CTRL_FORM_REC
     SENDING RENTAL_CLASSES_WKSP
     RECEIVING QUIT_CTRL_WKSP;
```

```
   ACTION
    CONTROL FIELD QUIT_CTRL_WKSP.QUIT_KEY
    "QUIT"   :   EXIT TASK;
    END CONTROL FIELD;
```

This step:

- Displays the rates for cars in the requested rental class
- Ends the task, if the user presses a function key

Once the DISPLAY_RENTAL_RATES step has displayed the car rates, ACMS returns control to the block step part of the definition.

2.3.4. Completing the Task Definition

Like the Add Car Reservation task, the Review Car Rates task uses DECforms to communicate with the terminal user. Therefore, you need to assign the FORM I/O attribute to the block step.

You define the start of the block step with the BLOCK WORK keywords. You use the END BLOCK WORK keywords to signal the end of the work done in the block and to separate the block work from the block action.

For inquiry tasks, terminal users are likely to want to look at more than one record without having to reselect the task from a menu. You can let them do this by using the REPEAT TASK clause in the action part of the block step definition.

Finally, you must identify the workspaces used by steps in the task. Here is the task part of the definition for the Review Car Rates task:

```
WORKSPACES ARE RENTAL_CLASSES_WKSP, QUIT_CTRL_WKSP, MSG_WKSP;
```

The task automatically uses the ACMS$PROCESSING_STATUS workspace; do not name that workspace in the WORKSPACES clause. *Example 2.5, "Complete Definition of the Review Car Rates Task"* shows the complete definition for the Review Car Rates task.

Example 2.5. Complete Definition of the Review Car Rates Task

```
REPLACE TASK REVIEW_CAR_RATES_TASK /LIST=CARRTRV.LIS

WORKSPACES ARE RENTAL_CLASSES_WKSP, QUIT_CTRL_WKSP, MSG_WKSP;
BLOCK WORK WITH FORM I/O

DETERMINE_RENTAL_CLASS:
   EXCHANGE
    TRANSCEIVE FORM RECORD RENTAL_CLASSES_FORM_REC,
                           RENTAL_CLASSES_FORM_REC_LIS
     SENDING RENTAL_CLASSES_WKSP
     RECEIVING RENTAL_CLASSES_WKSP, QUIT_CTRL_WKSP;
   ACTION
    CONTROL FIELD IS QUIT_CTRL_WKSP.QUIT_KEY
    "QUIT"  :  EXIT TASK;
    END CONTROL FIELD;

GET_RENTAL_RATES:
   PROCESSING
```

```
     CALL GET_RATES_PROC IN RENTAL_SERVER
       USING RENTAL_CLASSES_WKSP;
   ACTION IS
     IF (ACMS$T_STATUS_TYPE EQ "B")
     THEN GET ERROR MESSAGE;
          MOVE ACMS$T_STATUS_MESSAGE TO MSG_WKSP.MESSAGE_PANEL;
          GOTO NEXT EXCHANGE;
     ELSE GOTO STEP DISPLAY_RENTAL_RATES;
     END IF;

NO_RC_MSG:
   EXCHANGE
     TRANSCEIVE FORM RECORD MSG_FORM_REC, QUIT_CTRL_FORM_REC
      SENDING MSG_WKSP
      RECEIVING QUIT_CTRL_WKSP;
   ACTION IS
     IF (QUIT_CTRL_WKSP.QUIT_KEY EQ "QUIT")
     THEN EXIT TASK;
     ELSE GOTO STEP DETERMINE_RENTAL_CLASS;
     END IF;

DISPLAY_RENTAL_RATES:
   EXCHANGE WORK IS
    TRANSCEIVE FORM RECORD RENTAL_CLASSES_FORM_REC, QUIT_CTRL_FROM_REC
     SENDING RENTAL_CLASSES_WKSP
     RECEIVING QUIT_CTRL_WKSP;
   ACTION
    CONTROL FIELD QUIT_CTRL_WKSP.QUIT_KEY
    "QUIT"   :   EXIT TASK;
    END CONTROL FIELD;
END BLOCK WORK;

ACTION
   REPEAT TASK;

END DEFINITION;
```

When making changes to the definition, use the **REPLACE** command on the first line in the file to minimize the changes that you have to make to the file later. If you want to produce a listing file when submitting the file to ADU, you must include the **/LIST** qualifier in the file itself, because you cannot use it with the at sign character (@). The listing file shows you the source definition file and errors encountered by ADU when that definition was processed with the **CREATE** or **REPLACE** command.

To submit the definition file REVIEW_CAR_RATES_TASK.COM to ADU, type:

```
ADU> @REVIEW_CAR_RATES_TASK.COM
```

If there are no syntax errors in the definition, ADU stores it in the dictionary. If there are errors in the definition, you can edit the source definition file and resubmit it to ADU as a command file.

2.3.5. Additional Considerations: Displaying Multiple Records

The previous sections show how to define a simple inquiry task, Review Car Rates. That task displayed a single record from a file. However, you might want to define a task that lets a user display many records at the same time. Also, all the records requested by the user might not fit on a single screen.

For example, a car rental agency that handles many corporate car rentals needs to be able to review all the current reservation records for a particular company's employees. The Review Reservation task lets a terminal user display the reservation records for employees in a particular company. The task displays only five records at a time. However, it lets the terminal user display five more records without redisplaying the initial panel.

Because the Review Reservation task is still an inquiry task, its basic structure is the same as before. However, although the basic parts of the problem are the same, the action taken at the end of the second and fourth steps is affected by the following:

- Whether or not there are more records available for display
- Whether or not the terminal user wants to see more records

The steps for the Review Reservation task follow:

1. Display a panel requesting the identification number of the company whose employee reservations the terminal user wants to see.
2. Read information from the Reservation file.
3. Display error message if necessary.
4. Display records for the terminal user. If the user wants to see more records, repeat the second and fourth parts. Otherwise, end the task.

2.3.5.1. Getting Information from the User

The exchange step you define to get information is virtually the same in the Review Reservation task as in the Review Car Rates task:

```
GET_COMPANY_ID:
   EXCHANGE
     TRANSCEIVE FROM RECORD CO_RESERVE_FORM_REC,
                            CO_RESERVE_FORM_REC_LIS
      SENDING CO_RESERVE_WKSP
      RECEIVING CO_RESERVE_WKSP, QUIT_CTRL_WKSP;
   ACTION IS
     CONTROL FIELD IS QUIT_CTRL_WKSP.QUIT_KEY
       "QUIT"     : EXIT TASK;
     END CONTROL FIELD;
```

Like the Review Car Rates task, the Review Reservation task uses one workspace for storing data returned by the form and another workspace for testing whether or not the terminal user wants to exit from the task. *Example 2.6, "Record Description for REVIEW_RESERVATION_WORKSPACE"* shows the record description of CO_RESERVE_WKSP.

Example 2.6. Record Description for REVIEW_RESERVATION_WORKSPACE

```
Definition of record CO_RESERVE_WKSP
|   Contains field          COMP_ID
|   |  Datatype                 text size is 5 characters
|   Contains field          COMP_NAME
|   |  Datatype                 text size is 20 characters
|   Contains field          WK_ERR_MSG
|   |  Datatype                 text size is 4 characters
|   Contains field          WK_SAVE_NUMBER
```

```
|   |  Datatype                 signed longword
|   Contains record          EMPL
|   |  Row_major array           1:5
|   |  Contains field            EMPL_NAME
|   |  |  Datatype                   text size is 30 characters
|   |  Contains field            EMPL_PHONE
|   |  |  Datatype                   text size is 10 characters
|   |  Contains field            CAR_TYPE
|   |  |  Datatype                   text size is 3 characters
|   |  Contains field            RENTAL_DATE
|   |  |  Datatype                   text size is 6 characters
|   |  Contains field            RETURN_DATE
|   |  |  Datatype                   text size is 6 characters
```

2.3.5.2. Retrieving Information

The definition of the processing step for the Review Reservation inquiry task is very similar to that for the Review Car Rates inquiry task. It calls a procedure and handles recoverable errors.

```
GET_FIVE_RESERVATIONS:
   PROCESSING
     CALL REVIEW_RESERVATION_PROC IN RESERVATION_SERVER
       USING CO_RESERVE_WKSP;
   ACTION IS
     IF (ACMS$STATUS_TYPE EQ "B")
     THEN GET ERROR MESSAGE;
          MOVE ACMS$T_STATUS_MESSAGE TO MSG_WKSP.MESSAGE_PANEL;
          GOTO NEXT EXCHANGE;
     ELSE GOTO STEP DISPLAY_RESERVATIONS;
     END IF;
```

However, the REVIEW_RESERVATION_PROC procedure, after reading as many records as fit on the terminal screen, must tell the form whether or not there are more records available for display.

The REVIEW_RESERVATION_PROC procedure writes either MORE or STOP to the WK_ERR_MSG field of CO_RESERVE_WKSP. You must code your IFDL form file to test that field to tell the terminal user whether or not there are more records to see.

As in the previous task definitions, you need to add code to handle possible processing errors. The processing step tests the ACMS$T_STATUS_TYPE field, and if the field indicates that an error occurred, ACMS moves the error message to the MSG_WKSP workspace and passes control to the DISPLAY_ERROR exchange step.

```
DISPLAY_ERROR:
    EXCHANGE
     TRANSCEIVE FORM RECORD MSG_FORM_REC, QUIT_CTRL_FORM_REC
      SENDING MSG_WKSP
      RECEIVING QUIT_CTRL_WKSP.QUIT_KEY;
     ACTION
      IF (QUIT_CTRL_WKSP.QUIT_KEY EQ "QUIT")
      THEN EXIT TASK;
      ELSE GOTO STEP GET_COMPANY_ID;
      END IF;
```

This exchange step is identical to that used in the Review Car Rates task. After it displays the error message to the terminal user, it checks to see whether or not the user wants to exit from the task. If the user does not want to exit, ACMS returns control to the first exchange step.

2.3.5.3. Displaying Information to the User

As in any exchange step, when you use a form to display information, you can let the terminal user press a function key to exit the task. In this type of inquiry task, you also want to do the following:

- Use the form to tell the user whether or not there are more records available for display
- Let the user press a function key to see more records
- Retrieve and display the next five records if the user wants to see more records

Here is the definition of the final exchange step of the Review Reservation task:

```
DISPLAY_RESERVATION:
   EXCHANGE
    TRANSCEIVE FORM RECORD CO_RESERVE_FORM_REC, QUIT_CTRL_FORM_REC
     SENDING CO_RESERVE_WKSP
     RECEIVING QUIT_CTRL_WKSP;
    ACTION IS
     CONTROL FIELD IS QUIT_CTRL_WKSP.QUIT_KEY
     "QUIT"   :   EXIT TASK;
     "MORE"   :   GOTO PREVIOUS PROCESSING;
     END CONTROL FIELD;
```

Just as you declared the PF4 function key to be the QUIT_KEY in the Add Car Reservation task, you must declare a function key and function response in your IFDL file so that the terminal user can press that key to see five more records. When the terminal user presses that key, DECforms passes the "MORE" value into the QUIT_KEY field of the QUIT_CTRL_WKSP workspace. When ACMS processes the DISPLAY_RESERVATION step, if that field contains "MORE", ACMS repeats the GET_FIVE_RESERVATIONS processing step. The user avoids returning to the first step to retype the company identification number.

2.3.5.4. Completing the Task Definition

As with the Review Car Rates task, the block step in the Review Reservation task uses DECforms to interface with the terminal user. Therefore, you need to assign the FORM I/O attribute to the block.

Although the Review Reservation task has just one processing step, that step can be repeated many times. You need to consider whether or not the task needs to have the same server process each time it runs that processing step; you decide to retain or release **server context**.

Suppose the REVIEW_RESERVATION_PROC procedure reads the first five records, and the form displays those records. For the user to look at the next five records, REVIEW_RESERVATION_PROC must keep a pointer in the file to keep track of which record was last read. You need some way to save this pointer so that the procedure can use it if the user wants to see more records.

You can choose one of two ways to retain pointers between steps in a task:

- Retain the pointers in a workspace
- Retain server context

To retain pointers in a workspace, the procedure must write those pointers to the workspace. Pointers are part of the context associated with a server process. Therefore, if you retain server context, you retain the file pointers and do not have to write those pointers to a workspace. However, in a task such as Review

Reservation, retaining server context means retaining a process until the user looks at all the records the user wants to see.

The least expensive choice in terms of system resources is to write the file pointer to a workspace rather than retain server context. In the Review Reservation task, the procedure writes this file pointer to the WK_SAVE_NUMBER field of CO_RESERVE_WKSP and releases server context. *VSI ACMS for OpenVMS Concepts and Design Guidelines* [https://docs.vmssoftware.com/vsi-acms-concepts-and-design-guide/] explains server context in more detail.

In addition to considering the general characteristics of the block step, consider the actions you want to take as a result of the work done in that block. Suppose that you want to take the same actions as those defined in the action part of the block step for the Review Car Rates task: repeat the task unless the user presses the **PF4** key in the final exchange step. You use the REPEAT TASK clause in the action part of the block step definition.

```
  .
  .
  .
END BLOCK WORK;

ACTION
   REPEAT TASK;
```

Finally, use the WORKSPACE clause to name the workspaces that you define (CO_RESERVE_WKSP, QUIT_CTRL_WKSP, and MSG_WKSP) for the Review Reservation task. *Example 2.7, "Complete Definition of Review Reservation Task"* shows the complete definition for the Review Reservation task.

Example 2.7. Complete Definition of Review Reservation Task

```
REPLACE TASK REVIEW_RESERVATION_TASK /LIST=RVRSV.LIS
   WORKSPACES ARE CO_RESERVE_WKSP, QUIT_CTRL_WKSP, MSG_WKSP;

BLOCK WORK WITH FORM I/O

   GET_COMPANY_ID:
   EXCHANGE
     TRANSCEIVE FROM RECORD CO_RESERVE_FORM_REC,
                            CO_RESERVE_FORM_REC_LIS
      SENDING CO_RESERVE_WKSP
      RECEIVING CO_RESERVE_WKSP, QUIT_CTRL_WKSP;
   ACTION IS
     CONTROL FIELD IS QUIT_CTRL_WKSP.QUIT_KEY
       "QUIT"     : EXIT TASK;
     END CONTROL FIELD;

   GET_FIVE_RESERVATIONS:
     PROCESSING
       CALL REVIEW_RESERVATION_PROC IN RESERVATION_SERVER
         USING CO_RESERVE_WKSP;
     ACTION IS
       IF (ACMS$STATUS_TYPE EQ "B")
       THEN GET ERROR MESSAGE;
            MOVE ACMS$T_STATUS_MESSAGE TO MSG_WKSP.MESSAGE_PANEL;
            GOTO NEXT EXCHANGE;
       ELSE GOTO STEP DISPLAY_RESERVATIONS;
       END IF;
```

```
   DISPLAY_ERROR:
    EXCHANGE
     TRANSCEIVE FORM RECORD MSG_FORM_REC, QUIT_CTRL_FORM_REC
      SENDING MSG_WKSP
      RECEIVING QUIT_CTRL_WKSP;
     ACTION
      IF (QUIT_CTRL_WKSP.QUIT_KEY EQ "QUIT")
      THEN EXIT TASK;
      ELSE GOTO STEP GET_COMPANY_ID;
      END IF;

   DISPLAY_RESERVATION:
   EXCHANGE
    TRANSCEIVE FORM RECORD CO_RESERVE_FORM_REC, QUIT_CTRL_FORM_REC
     SENDING CO_RESERVE_WKSP
     RECEIVING QUIT_CTRL_WKSP;
    ACTION IS
     CONTROL FIELD IS QUIT_CTRL_WKSP.QUIT_KEY
     "QUIT"   :   EXIT TASK;
     "MORE"   :   GOTO PREVIOUS PROCESSING;
     END CONTROL FIELD;

END BLOCK WORK;
ACTION
   REPEAT TASK;
END DEFINITION;
```

2.4. Defining an Update Task

This section presents an update task, the Review Update task, that lets the terminal user review an existing reservation record and change information in the record. *Table 2.4, "Update Task"* describes the steps of an update task.

Table 2.4. Update Task

Step	Type
Display a panel to get information	Exchange step
Read information from a file	Processing step
Display an error message, if necessary	Exchange step
Display a panel with information	Exchange step
Update file and display message, if necessary	Nested Block step

The steps of an update task apply to the Review Update task in the following manner:

1. Display a panel requesting the name of the customer whose reservation record the terminal user wants to update.

2. Read information about the reservation from the Reservation file.

3. If an error occurs during the processing step, display the error message on the screen and return to the first step.

4. Display reservation review information, allowing the terminal user to update the data.

5. Check to see whether or not the user has changed the reservation record; if yes, write the new data to the Reservation file and display a message to the user.

2.4.1. Getting Information from the User

To get information from the terminal user, you display a panel. Here is the first exchange step of the Review Update task:

```
GET_CUSTOMER_NAME:
   EXCHANGE
    TRANSCEIVE FORM RECORD RESERVE_FORM_REC, RESERVE_FORM_REC_LIS
     SENDING RESERVE_WKSP
     RECEIVING RESERVE_WKSP, QUIT_CTRL_WKSP;
   ACTION
     CONTROL FIELD IS QUIT_CTRL_WKSP.QUIT_KEY
     "QUIT"   :   EXIT TASK;
     END CONTROL FIELD;
```

This step is very similar to the first step in the inquiry task, Review Reservation. Both steps do the following:

- Call a form record to display a panel asking for a record key; in this case the record key is the customer's name.
- Let the terminal user end the task.
- Use an optional label to assign a name to the step.

As a result, the record definition for the workspace that you use to pass reservation data between the application and the form is very similar to the one in the Review Reservation task. *Example 2.8, "Definition for RESERVE_WKSP Workspace"* shows the record description of RESERVE_WKSP.

Example 2.8. Definition for RESERVE_WKSP Workspace

```
Definition of record RESERVE_WKSP
|   Contains field          CUST_NAME
|   |  Datatype                 text size is 30 characters
|   Contains field          CUST_STREET_ADDRESS
|   |  Datatype                 text size is 30 characters

|   Contains field          CUST_CITY
|   |  Datatype                 text size is 20 characters

|   Contains field          CUST_STATE
|   |  Datatype                 text size is 2 characters
|   Contains field          CUST_ZIP

|   |  Datatype                 text size is 5 characters
|   Contains field          CUST_PHONE
|   |  Datatype                 text size is 10 characters
|   Contains field          CAR_TYPE

|   |  Datatype                 text size is 3 characters
|   Contains field          RENTAL_DATE
|   |  Datatype                 text size is 6 characters
|   Contains field          RETURN_DATE
```

```
|   |  Datatype                  text size is 6 characters
```

2.4.2. Retrieving Information from a File

As shown in the data entry and inquiry tasks, reading from or writing to a file requires a processing step. Here is the processing step for the Review Update task:

```
FIND_RESERVATION:
   PROCESSING
    CALL FIND_RESERVE_PROC IN RESERVE_SERVER
     USING RESERVE_WKSP;
   ACTION
    IF (ACMS$T_STATUS_TYPE EQ "B")
    THEN GET ERROR MESSAGE;
         MOVE ACMS$T_STATUS_MESSAGE TO MSG_WKSP.MESSAGE_PANEL;
         GOTO NEXT EXCHANGE;
    ELSE GOTO STEP DISPLAY_RESERVATION;
    END IF;
```

When you update information, it is important that the contents of a record do not change from the time those contents are displayed to the user for update until the time you write the changed record back to the file. This is especially important in a multiuser application in which another user might update the record while it is being displayed on the terminal screen for the first user.

One way of ensuring the integrity of the record you are updating is to have the procedure lock the record and then retain server context between processing steps. However, it is much more efficient to release server context between processing steps. When you update a record, to ensure the integrity of the record without locking the record and retaining server context, you need to check whether the record was updated by someone else before writing the first user's changes to the file. See *VSI ACMS for OpenVMS Writing Server Procedures* [https://docs.vmssoftware.com/vsi-acms-writing-server-proc/] for further discussion on releasing server context.

As in the inquiry and data entry tasks, you must consider errors that can occur when the procedure tries to read from the file. If the reservation record for the customer name provided by the terminal user does not exist or is locked by another user, ACMS stores the value B in the ACMS$T_STATUS_TYPE field of the ACMS$PROCESSING_STATUS workspace. The processing step checks that field and, if the field contains B, retrieves the error message and stores it in the MSG_WKSP workspace. ACMS then passes control to the DISPLAY_ERROR exchange step:

```
DISPLAY_ERROR:
   EXCHANGE
    TRANSCEIVE FORM RECORD MSG_FORM_REC, QUIT_CTRL_FORM_REC
     SENDING MSG_WKSP
     RECEIVING QUIT_CTRL_WKSP;
    ACTION
     IF (QUIT_CTRL_WKSP.QUIT_KEY EQ "QUIT")
     THEN EXIT TASK;
     ELSE GOTO STEP GET_CUSTOMER_NAME;
     END IF;
```

This exchange step sends the error message stored in MSG_WKSP to the MSG_FORM_REC form record for display to the terminal screen. The action part of this step tests the QUIT_KEY field of the QUIT_CTRL_WKSP workspace to see whether or not the terminal user wants to exit from the task. If the user presses the **PF4** key, DECforms returns the value "QUIT" to the workspace, and ACMS ends the task and returns the user to the menu. Otherwise, ACMS passes control to the first step in the task definition.

2.4.3. Letting the User Update the Information

Once a procedure has read information from a file, you can display that information to the terminal user. The user can then supply information, in the form of changes, to be written back to the file. You use an exchange step to call a form record to display information to the user and to accept changes.

Here is the exchange step that displays information for the Review Update task:

```
DISPLAY_RESERVATION:
   EXCHANGE
    TRANSCEIVE FORM RECORD RESERVE_FORM_REC, RESERVE_FORM_REC_LIS
     SENDING RESERVE_WKSP
     RECEIVING RESERVE_WKSP, QUIT_CTRL_WKSP
               SHADOW IS RESERVE_SHADOW_WKSP;
   ACTION
    CONTROL FIELD IS QUIT_CTRL_WKSP.QUIT_KEY
    "QUIT"  :  EXIT TASK;
    END CONTROL FIELD;
```

For an inquiry task, such as Review Reservation, this second exchange step is the final step in the task. However, in an update task, the user can make changes to the information displayed.

The DISPLAY_RESERVATION step uses RESERVE_WKSP to send the information read by the FIND_RESERVE_PROC procedure to the form for display. The step also uses RESERVE_WKSP to store the information that the terminal user returns.

Unlike exchange steps in the data entry and inquiry tasks, this step uses a **shadow workspace** to determine whether or not the terminal user changed any information in the reservation record. When you declare a shadow workspace, DECforms returns a value to that workspace that indicates whether or not the user changed any data in the record. In this exchange step, RESERVE_SHADOW_WKSP is the shadow workspace for the RESERVE_WKSP workspace.

The record description for RESERVE_SHADOW_WKSP is the same as for RESERVE_WKSP, except that you must add a field to store the value that DECforms returns. *Example 2.9, "Record Description for RESERVE_SHADOW_WKSP Workspace"* shows the definition.

Example 2.9. Record Description for RESERVE_SHADOW_WKSP Workspace

```
Definition of record RESERVE_SHADOW_WKSP
|   Contains field          REC_STATUS
|   |  Datatype                 text size is 1 character
|   Contains field          CUST_NAME_SHADOW
|   |  Datatype                 text size is 1 character
|   Contains field          CUST_STREET_ADDRESS_SHADOW
|   |  Datatype                 text size is 1 character
|   Contains field          CUST_CITY_SHADOW
|   |  Datatype                 text size is 1 character
|   Contains field          CUST_STATE_SHADOW
|   |  Datatype                 text size is 1 character
|   Contains field          CUST_ZIP_SHADOW
|   |  Datatype                 text size is 1 character
|   Contains field          CUST_PHONE_SHADOW
|   |  Datatype                 text size is 1 character
|   Contains field          CAR_TYPE_SHADOW
|   |  Datatype                 text size is 1 character
|   Contains field          RENTAL_DATE_SHADOW
```

```
|   |  Datatype                text size is 1 character
|   Contains field          RETURN_DATE_SHADOW
|   |  Datatype                text size is 1 character
```

If the terminal user changed any data in RESERVE_WKSP, ACMS stores a 1 in the REC_STATUS field of RESERVE_SHADOW_WKSP when the transceive operation completes. The status field in the shadow workspace must be the first field in your record definition. In your DECforms IFDL code, you must assign the TRACKED attribute to the fields of shadow workspaces. See the DECforms documentation for information on using this attribute.

After the DISPLAY_RESERVATION step ends, you need to check the shadow workspace to see whether or not the reservation record has changed. If it has, write the new record to the Reservation file and display a message on the terminal screen confirming the update.

2.4.4. Writing the New Information to the File

The CHECK_RESERVE_CHANGES step is a **nested block step** that includes the following parts:

- Block conditional clause
- Processing step
- Exchange step

By using a nested block step, you can group processing and exchange steps and have ACMS process them only if a certain condition is met. You introduce a nested block the same way you introduce a block, with the BLOCK WORK keywords. As with other steps, you can assign a label to a nested block and refer to it from elsewhere in the task definition.

A block conditional clause is one of the four ADU conditional clauses (CONTROL FIELD, IF THEN ELSE, SELECT FIRST, or WHILE DO) used at the block step level. You can use it to start an exchange step, a processing step, or another block step. However, you can use a block conditional clause only at the start of a block step. You cannot use it between steps within the block. Here is the definition for the CHECK_RESERVE_CHANGES nested block step:

```
CHECK_RESERVE_CHANGES:
   BLOCK WORK
    IF (RESERVE_SHADOW_WKSP.REC_STATUS EQ "1")
    THEN
      PROCESSING
       CALL WRITE_RESERVE_PROC IN RESERVE_SERVER
        USING RESERVE_WKSP;
      ACTION
       MOVE "RESERVATION RECORD UPDATED" TO MSG_WKSP.MESSAGE_PANEL;

      EXCHANGE
       SEND FORM RECORD MSG_FORM_REC
        SENDING MSG_WKSP;
    END IF;
   END BLOCK;
```

In the CHECK_RESERVE_CHANGES step, an IF THEN ELSE block conditional clause tests the shadow workspace. If the REC_STATUS field equals 1, ACMS processes the processing and exchange steps that follow the THEN keyword. The processing step calls the WRITE_RESERVE_PROC procedure, which writes the new reservation record to the Reservation file.

The action part of the processing step moves the message "RESERVATION RECORD UPDATED" to the MSG_WKSP workspace, and the exchange step then sends that message to DECforms for display.

If the shadow record indicates that the terminal user did not change any data in the reservation record, ACMS does not perform the processing and exchange steps in the CHECK_RESERVE_CHANGES block step. Note that you do not need to include the ELSE keyword in the IF THEN ELSE clause. ACMS passes control to the clause following the END IF keywords. Be sure to end the nested block with the END BLOCK keywords.

2.4.5. Completing the Task Definition

The Review Update task uses DECforms to interface with the terminal; therefore, you need to assign the FORM I/O attribute to the block. Nested blocks inherit the attributes that you assign to their parent blocks; so, you do not need to include the FORM I/O keywords with the CHECK_RESERVE_CHANGES step.

Because you want to repeat this task automatically rather than make the terminal user choose the task again from the menu, include the REPEAT TASK clause in the action part of the parent block.

Finally, you must name the workspaces used by the steps in the task. In the Review Update task these are RESERVE_WKSP, RESERVE_SHADOW_WKSP, QUIT_CTRL_WKSP, and MSG_WKSP. Declare these workspaces using the WORKSPACES ARE clause in the task definition.

```
WORKSPACES ARE RESERVE_WKSP, RESERVE_SHADOW_WKSP, QUIT_CTRL_WKSP, MSG_WKSP;
```

Example 2.10, "Complete Definition of Review Update Task" shows the complete definition for the Review Update task.

Example 2.10. Complete Definition of Review Update Task

```
REPLACE TASK REVIEW_UPDATE_TASK /LIST=RVSCHED.LIS
   WORKSPACES ARE RESERVE_WKSP, QUIT_CTRL_WKSP, MSG_WKSP,
                  RESERVE_SHADOW_WKSP;

BLOCK WORK WITH FORM I/O

GET_CUSTOMER_NAME:
   EXCHANGE
    TRANSCEIVE FORM RECORD RESERVE_FORM_REC, RESERVE_FORM_REC_LIS
     SENDING RESERVE_WKSP
     RECEIVING RESERVE_WKSP, QUIT_CTRL_WKSP;
   ACTION
     CONTROL FIELD IS QUIT_CTRL_WKSP.QUIT_KEY
     "QUIT"   :   EXIT TASK;
     END CONTROL FIELD;

FIND_RESERVATION:
   PROCESSING
    CALL FIND_RESERVE_PROC IN RESERVE_SERVER
     USING RESERVE_WKSP;
   ACTION
    IF (ACMS$T_STATUS_TYPE EQ "B")
    THEN GET ERROR MESSAGE;
         MOVE ACMS$T_STATUS_MESSAGE TO MSG_WKSP.MESSAGE_PANEL;
         GOTO NEXT EXCHANGE;
    ELSE GOTO STEP DISPLAY_RESERVATION;
```

```
    END IF;

DISPLAY_ERROR:
   EXCHANGE
    TRANSCEIVE FORM RECORD MSG_FORM_REC, QUIT_CTRL_FORM_REC
     SENDING MSG_WKSP
     RECEIVING QUIT_CTRL_WKSP;
    ACTION
     IF (QUIT_CTRL_WKSP.QUIT_KEY EQ "QUIT")
     THEN EXIT TASK;
     ELSE GOTO STEP GET_CUSTOMER_NAME;
     END IF;

DISPLAY_RESERVATION:
   EXCHANGE
    TRANSCEIVE FORM RECORD RESERVE_FORM_REC, RESERVE_FORM_REC_LIS
     SENDING RESERVE_WKSP
     RECEIVING RESERVE_WKSP, QUIT_CTRL_WKSP
               SHADOW IS RESERVE_SHADOW_WKSP;
   ACTION
    CONTROL FIELD IS QUIT_CTRL_WKSP.QUIT_KEY
    "QUIT"  :  EXIT TASK;
    END CONTROL FIELD;

CHECK_RESERVE_CHANGES:
   BLOCK WORK
    IF (RESERVE_SHADOW_WKSP.REC_STATUS EQ "1")
    THEN
      PROCESSING
       CALL WRITE_RESERVE_PROC IN RESERVE_SERVER
        USING RESERVE_WKSP;
      ACTION
       MOVE "RESERVATION RECORD UPDATED" TO MSG_WKSP.MESSAGE_PANEL;

      EXCHANGE
       SEND FORM RECORD MSG_FORM_REC
        SENDING MSG_WKSP;
    END IF;
   END BLOCK;
END BLOCK;
ACTION
   REPEAT TASK;
END DEFINITION;
```

Chapter 3. Using VSI DECforms with ACMS

Chapter 3, "Using VSI DECforms with ACMS" describes how the implementation of DECforms affects ACMS task and application definitions. Separate sections in this chapter explain the ACMS interface to DECforms, making calls to DECforms external requests, new DECforms user interface features, writing and compiling DECforms escape units, and a comparison of DECforms and TDMS.

3.1. ACMS Interface to VSI DECforms

The ACMS interface to DECforms is made up of six calls. These calls enable, disable, or cancel a form as well as send, receive, and transceive (a combination of send and receive) information to and from the form. ACMS makes these calls to DECforms **external requests**, which are requests called from *outside* the form.

The next section explains how external requests are called by ACMS. The second section outlines how DECforms processes external requests.

3.1.1. Calls to External Requests

ACMS automatically makes calls to three external requests:

- ENABLE – When a form is accessed in a task for the first time
- DISABLE – When you exit from ACMS
- CANCEL – When you use the DCL commands **`ACMS/CANCEL USER`**, **`ACMS/CANCEL TASK`**, or to cancel a task

You make calls to the remaining three external requests in the EXCHANGE step of an ACMS task definition:

- SEND

  Use the SEND call in a task definition when you want to send data to the form.

- RECEIVE

  Use the RECEIVE call in a task definition to receive data from the form.

- TRANSCEIVE

  Use the TRANSCEIVE call to send data to and receive data from the form.

3.1.2. Processing External Requests

In processing external requests, DECforms follows a specific order of steps, called **phases**. See *VSI DECforms Programmer's Reference Manual* [https://docs.vmssoftware.com/vsi-decforms-for-openvms-programmer-s-reference-manual/] for more details on how phases proceed with DECforms.

3.1.3. Responses to External Requests

With DECforms, use **responses** to control the operation of forms processing. You can think of a DECforms response as a way of directing how DECforms **responds** or behaves. By declaring responses, you can determine much of what DECforms does in its interaction with the user's terminal, the form, and the ACMS application.

The following list describes DECforms responses that you can use to direct the actions of forms processing. The descriptions include references to the DECforms documentation for additional information.

- Control text responses

  You can use control text responses to send information to DECforms (such as display directions) or to collect information from DECforms (such as status information about the completion of a SEND, RECEIVE, or TRANSCEIVE operation).

  Refer to *VSI DECforms IFDL Reference Manual* [https://docs.vmssoftware.com/vsi-decforms-for-openvms-ifdl-reference-manual/] for more information about control text responses.

- External responses

  After it processes control text responses, DECforms processes other responses that you enter in the form source (IFDL) file. If you do not declare responses in the IFDL file, the DECforms performs certain default actions for each call to an external request. Default actions are detailed in *VSI DECforms Programmer's Reference Manual* [https://docs.vmssoftware.com/vsi-decforms-for-openvms-programmer-s-reference-manual/].

  To alter the default actions of DECforms or to direct DECforms to take certain actions if there are no defaults, you must enter external responses in the form source IFDL file.

  Refer to *VSI DECforms IFDL Reference Manual* [https://docs.vmssoftware.com/vsi-decforms-for-openvms-ifdl-reference-manual/] for more information on declaring external responses.

- Response steps

  A response step is an instruction to DECforms to alter the processing of an external request. Enter response steps in the form source IFDL file.

  VSI DECforms Guide to Commands and Utilities [https://docs.vmssoftware.com/vsi-decforms-for-openvms-guide-to-commands-and-utilities/] contains a table of response steps, and *VSI DECforms Programmer's Reference Manual* [https://docs.vmssoftware.com/vsi-decforms-for-openvms-programmer-s-reference-manual/] describes each response step in detail.

  An example of a response step is DISPLAY, which causes the named panels on the terminal screen to display.

- Accept responses

  Enter accept responses in the form source IFDL file to determine what DECforms does during user input. The four types of accept responses are ENTRY, EXIT, FUNCTION, and VALIDATION.

  Refer to *VSI DECforms Guide to Commands and Utilities* [https://docs.vmssoftware.com/vsi-decforms-for-openvms-guide-to-commands-and-utilities/] for more information about accept responses.

- Internal responses

  An internal response is one that is called from another response. You can use internal responses for any action that is called for repeatedly in a form. For example, you might want to direct DECforms to display a message each time a user enters the last item in a list.

3.2. Writing and Compiling VSI DECforms Escape Units

In DECforms, you can use escapes to call subroutines from the form. These subroutines, which are called **escape units**, perform actions outside the form, such as data validation or database lookup. Escape units are similar to UARs in FMS.

Note

Many agents, including the CP agent supplied by ACMS, perform work on behalf of multiple users. These agents are multithreaded or asynchronous. However, DECforms escape units provide only a synchronous interface. When you use an escape unit in a multithreaded or asynchronous agent, be careful not to do anything that might delay the CP agent in its processing other users' requests. For example, do not perform terminal or disk I/O from an escape unit executing in a multithreaded or asynchronous agent.

Using DECforms escape units is a two-step procedure. You must write the escape unit, and you must also edit the form IFDL source file to call the escape unit from the form. The following sections explain the steps necessary to write and implement DECforms escape units.

3.2.1. Writing an Escape Unit

You can write an escape unit in COBOL or in any other high-level programming language that supports the OpenVMS calling standard. *Example 3.1, "Example of an Escape Unit"* shows a COBOL program that counts the number of employees added to the EMPLOYEE_INFO_RECORD_1.

Example 3.1. Example of an Escape Unit

```
************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. EMPLOYEE_COUNT.

************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.          VAX-11.
OBJECT-COMPUTER.          VAX-11.

************************************************
DATA DIVISION.
WORKING-STORAGE SECTION.

LINKAGE SECTION.
01 EMPL_COUNT                      PIC S9(9) COMP.
```

```
************************************************
PROCEDURE DIVISION USING COUNTER.

00-DO-ADD.

        ADD 1 TO COUNTER.
        END PROGRAM EMPLOYEE_COUNT.
```

One use for the COUNTER value is to display its value on a panel after a message such as "Total number of employees is."

After you write an escape unit, you create an object (.OBJ) file by compiling the escape unit. For example:

```
$ COBOL employee_count
```

3.2.2. Calling an Escape Unit in a Form Source IFDL File

To use a DECforms escape unit, you must also enter a response step in the form source IFDL file to call that escape unit. The function of the escape unit created in *Example 3.1, "Example of an Escape Unit"* is to count employees as they are added to the database. You need to place the call to the escape unit within an external response, which directs DECforms to vary its processing of an external request. For example:

```
RECEIVE RESPONSE EMPLOYEE_INFO_FORM
     DISPLAY EMPLOYEE_INFO_PANEL_1
     CALL 'EMPLOYEE_COUNT' USING EMPL_NUMBER
END RESPONSE
```

In the previous example, the CALL response step calls the escape unit EMPLOYEE_COUNT and passes the form data item EMPL_NUMBER to the escape unit. For each EMPL_NUMBER sent to it, the escape unit adds 1 to the COUNTER variable and returns the new value to the form. When the escape unit finishes, DECforms ends the response.

3.3. Linking VSI DECforms Form Objects

After you have edited your form's IFDL source file and translated it into a binary .FORM file, you need to create an object module and a shareable image of the form to use in your application. First, use the DECforms **EXTRACT OBJECT** command as follows:

```
$ FORMS EXTRACT OBJECT EMPLOYEE_INFO_FORM.FORM
```

This command creates a form object module, or .OBJ file. Then link the object into a shareable image:

```
$ LINK/SHARE EMPLOYEE_INFO_FORM.OBJ
```

The result of the **LINK/SHARE** command is an image (.EXE) of the file, which can be shared by multiple users.

3.4. Linking Escape Units

The two methods for linking the escape units used in an application are:

- Link escape unit objects with form objects. For example:

```
$ LINK/SHARE EMPLOYEE_INFO_FORM.OBJ, EMPLOYEE_ESC_UNIT.OBJ
```

- Link escape unit objects into a separate image. For example:

```
$ LINK/SHARE=SHARE_ESC_UNIT.EXE EMPLOYEE_ESC_UNIT1.OBJ, -
_$ EMPLOYEE_ESC_UNIT2.OBJ, ESC_UNIT.OPT/OPT
```

When you link escape units into a separate image, you must use an options file to declare all escape unit names as universal. For example:

```
UNIVERSAL = EMPLOYEE_ESC_UNIT1
UNIVERSAL = EMPLOYEE_ESC_UNIT2
```

If you link escape units separately, you can place them in a write-protected directory for security. However, ACMS does not automatically cache escape unit images that are stored separately. Instead, the system manager must manually distribute the escape unit image files. *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/] explains caching ACMS and DECforms files in more detail.

Note

When you use DECforms, form objects and escape unit objects cannot be linked with agents.

The following list describes the advantages and disadvantages of each method for linking escape units:

- Linking escape units with the form objects

  You can make escape units available to the Command Process (CP) agent by linking them into the same image as the form. An advantage of this method is that, if your system uses distributed processing, the escape units are automatically cached, along with the form files, to the remote submitter node.

  By using this method, however, you allow application code from a remote system to execute in a privileged environment on your system. But if the remote application node and the local submitter node are in a common security domain, this may not be a concern.

  To avoid the possible security problem of executing escape units that are linked with the forms shareable image, ACMS, by default, does not execute these escape units. To execute them, you must define the following system-level logical:

  ```
  ACMS$ESC_RTN_IN_FORM
  ```

  If the value of this logical is set to T, t, Y, y, or l, then ACMS allows these escape units to run. If this logical is not defined or has another value, ACMS does not execute the escape units.

- Linking escape units in a separate image

  You can also link escape units in their own shareable image. You then need to make that image available to the CP agent by defining the following logical name:

  ```
  FORMS$IMAGE
  ```

  If there are more than one escape unit image, you can make the logical a search list of all images. For the CP agent supplied by ACMS, this logical must be in either the system group name table or the

system name table. Refer to the DECforms documentation for more information about defining this logical.

If you use the FORMS$IMAGE logical to specify escape unit images, these escape units can be shared by all of the applications that the CP agent references. However, sometimes the different applications may have the same escape unit name for different functions. To settle this possible naming conflict, ACMS provides an additional system-level logical name:

```
ACMS$ESC_RTN_<node>_<application>
```

You can use this logical to define an escape unit image for a particular application on a particular node. If the application uses more than one image, this logical can be a search list. When you use this logical name on a system where the NODE_NAME parameter is not defined in ACMSGEN, you must omit the `<node>` part from the logical name. Therefore, the system-level logical name looks like:

```
ACMS$ESC_RTN_<application>
```

If you do not define either of these logicals, or if the specified logical is not on the search list, DECforms then uses the FORMS$IMAGE logical.

3.4.1. Managing Escape Unit Files

Escape units are executed in the context of the agent process. Consequently, the application and system managers must consider the following:

- Making escape units available to the CP agent
- Protecting privileged agents that execute escape units

Certain agents, such as the CP agent that ACMS supplies, run in a privileged environment. Any escape units that execute in a privileged agent also execute in the same privileged environment. Be sure to develop escape units to execute in a privileged agent in the same way as you develop any other privileged code. Carefully review the code for any possible security violations before you run it live on the system.

Place the escape unit images in a write-protected directory; also write-protect the images themselves. After you place the file in a protected directory, you can define the logical names that make the escape units available to the agent process.

You may not, however, need to place the escape units executed by a nonprivileged agent in a protected directory. In some cases, such as debugging, you may want the person running the CP agent to be able to change the escape units executed. Use the methods described in this section to make the escape unit images available to the CP agent.

3.4.2. Replacing Form Files

If you need to change a DECforms file in binary format (.FORM), stop the application and restart it before using the new file. This procedure ensures that EXC has the correct file information for caching purposes.

DECforms files in image format (.EXE) require different treatment. Before using a new form image file for DECforms files in image format, stop the terminal subsystem so that the CP can map the new version of the form image. This action is necessary because OpenVMS maps these images into the processes that are using them, and there is no way to unmap the images.

Form image files are recommended for production environments, because they require fewer resources in multithreaded environments than files in binary format. The following procedure is suggested for customers who need to replace form image files without stopping the agent process.

The ACMS task group definition must specify the form file with a logical name rather than a file specification. Define the logical name outside ACMS in a logical name table accessed by both the EXC process and the application manager.

To replace a form file:

- Rename the new form file.
- Change the definition of the logical name for the form file to point to the new file specification.
- Stop and restart the application.

When you follow this procedure, the application picks up the new name of the form file and passes it to the agent. The agent then asks OpenVMS to map the renamed file. Because the OpenVMS image activation code sees the new name of the form file, it considers the file to be different from any files that are already mapped in. The renamed form file is mapped in, and the agent process begins to use it.

Note that the OpenVMS image activation code uses only the file name portion of the file specification to determine whether or not it is a new image. Changing other parts of the file specification (for example, device, directory, file extension, or version number) has no effect. Each image can be activated only once in a process. If an image file has been activated, then a different image file with the same name is not activated.

3.5. Creating Forms Trace Files

To create DECforms forms trace files, define the DECforms logical names FORMS$TRACE and FORMS$TRACE_FILE. If you are using the CP, define the two DECforms logical names in the the system name table.

If you define FORMS$TRACE but not FORMS$TRACE_FILE, DECforms writes the trace file to the CP default directory. In this case, you must ensure that the CP default directory exists. Otherwise, the CP does not enable forms because it cannot create trace files for the forms. If this happens, ACMS does not let users sign in. This situation occurs even if you define the ACMS$DEFAULT_MENU_FORMS_PRODUCT logical name to be TDMS.

3.6. Naming Forms Image Files

If you use DECforms forms image files, adhere to the following restrictions when you name the image files:

- The file name of the image must be unique. The OpenVMS image activator keeps track of images by file name only. Therefore, forms image file names must be unique, even if they are located in different directories or on different devices.
- Avoid defining the file name of a forms image file as a logical. The OpenVMS image activator attempts to perform a logical translation of the file name. However, it is recommended that you do not define the file name of a forms image file to be a logical because this can cause problems with the ACMS management of the forms file. If you do define the file name of a forms image file to be a logical, the logical name must refer to a valid form file. Otherwise, ACMS generates the following error message when a task tries to access the forms image file:

```
An error was returned from a DECforms request
%FORMS-F-INVFHDSIZ, invalid form header size.
```

The file name of the ACMS menu is ACMS_MENU. If you define ACMS_MENU as a logical, it must refer to a forms image file that contains a valid form for the ACMS menu. Otherwise, the CP does not let users sign in to ACMS.

3.7. User Interface Features with VSI DECforms

DECforms user interface features include the following:

- **Ctrl/Z** or **F10** to exit from a DECforms menu

  As part of DECforms design, **Ctrl/Z** can be defined so that it functions the same as the **EXIT** command does. When a user presses **Ctrl/Z** or **F10** for a DECforms menu selection, the CP handles it like an **EXIT** command, and the user exits out of ACMS.

- Cursor repositioning

  In ACMS, a user can press **PF2**and then the **UP/DOWN** arrow keys to move the cursor up and down among the entries. If the user presses **Return**, the CP handles it just as if the user had typed the number of the item.

- Line recall selection

  In ACMS, a user can press the **UP/DOWN** arrow keys to recall a selected line backward or forward just like the OpenVMS command line recall function.

3.8. Comparison of VSI DECforms and TDMS

Although there are not always exact equivalents between DECforms and TDMS, *Table 3.1, "VSI DECforms and TDMS Terminology"* is useful in understanding the differences and the similarities of these two forms products.

Table 3.1. VSI DECforms and TDMS Terminology

VSI DECforms Term	TDMS Equivalent
Form	Request
Form file	Request library file
Form record	CDD record definition
IFDL source file	Request definition file
Panel	Form
Panel field	Form field
Panel Editor	Form editor (Layout Phase)
Layout	(No equivalent)
Viewport	(No equivalent)
Request response	(No equivalent)

VSI DECforms Term	TDMS Equivalent
Request calls	TDMS programming calls
Literals	Background text
Function keys	Program request keys and redefined keys

Refer to *DECforms Guide to Converting VAX TDMS Applications* for more information about TDMS and DECforms terminology.

Note

You can use the TDMS Converter to convert TDMS forms or TDMS requests, or both, to a DECforms IFDL source file. There are many differences between the two forms products, and ACMS continues to support TDMS forms. Therefore, it is recommended that you use DECforms to develop new forms, but that you continue to use TDMS forms for existing ACMS applications. See *DECforms Guide to Converting VAX TDMS Applications* for more information on conversions.

Chapter 4. Defining Workspaces

Chapter 2, "Defining Tasks" briefly introduces the concept of ACMS workspaces and shows how to use them to pass data between parts of an application. *Chapter 4, "Defining Workspaces"* describes workspaces in more detail. Specifically, this chapter describes the three types of ACMS workspaces and explains how to use them to perform the following functions:

- Using a workspace that you define, rather than a system workspace, to do error handling in a definition
- Using data that the user types in response to the Selection: prompt on a menu as input to a task definition
- Using special kinds of workspaces to pass information needed for more than one instance of a task

VSI ACMS for OpenVMS Writing Server Procedures [https://docs.vmssoftware.com/vsi-acms-writing-server-proc/] explains how to write the procedures required for the task definitions explained in this chapter.

4.1. Understanding the Types of ACMS Workspaces

The three types of workspaces you can use in your task definitions are:

- Task
- User
- Group

Task is the default and most frequently used workspace type. Use task workspaces to store database records; to pass data between parts of your application, such as the form and exchange steps; and to perform error handling. Because ACMS retains task workspaces only for the life of a task instance, task workspaces do not use a lot of memory and CPU time. A task instance is one iteration of a task. Therefore, unless your application has certain requirements that can be met only by using user or group workspaces, use task workspaces.

A system workspace is one of three ACMS-supplied task workspaces. The ACMS$PROCESSING_STATUS workspace stores status information returned from procedures. The ACMS$SELECTION_STRING workspace stores text strings passed by the terminal user when the user selects a task from the menu. The ACMS$TASK_INFORMATION workspace stores task execution information. Because system workspaces are task workspaces, they do not use much memory and CPU time.

User workspaces let you store information used by a single user in many tasks or many task instances. User workspaces are helpful when the user needs to use the same information as a key for multiple task selections. For example, in a personnel application, a user might need to retrieve various information about an employee by choosing several tasks. Instead of requiring the user to type in the employee's number for each task, you can store the employee number in a user workspace so that it is available for all tasks that the user selects during that particular ACMS session.

You might also use user workspaces to store user-specific information that changes during the user's session. For example, you might want to keep a log of the type of work done by the user. Because ACMS

retains user workspaces for the duration of the user's session, user workspaces require more memory and CPU time than task workspaces.

Group workspaces let you store information used by many users in many tasks or instances of the same task. You typically use group workspaces to store static information. For example, an accounting application might include several tasks that use the current interest rate. Instead of requiring users to repeatedly enter the interest rate, you can store it in a group workspace. Because ACMS retains group workspaces as long as the application is started, group workspaces require more memory and CPU time than task workspaces.

Table 4.1, "Summary of ACMS Workspaces" briefly summarizes the availability and purpose of the three types of workspaces.

Table 4.1. Summary of ACMS Workspaces

Type	Available	Purpose
Task	For duration of task instance	Passing information between: • Steps in a task • Exchange steps and forms • Processing steps and servers • Parent tasks and called tasks
User	For user's ACMS session	Storing user-specific information
Group	As long as the application is started	Storing static information required by many tasks in a group

4.2. Handling Errors with a Task Workspace

You want to trap errors that a task encounters and tell the user what happened whenever a task encountered them. You use workspaces to pass error-related information among the procedure, the definition, and the form in task definitions.

There are two kinds of workspaces you may want to use for error handling:

- The ACMS system workspace ACMS$PROCESSING_STATUS
- A task workspace that you define

The task definitions explained in *Chapter 2, "Defining Tasks"* used the ACMS$PROCESSING_STATUS system workspace for error handling. When you use the ACMS$PROCESSING_STATUS workspace with the GET ERROR MESSAGE clause, ACMS:

1. Checks the status value returned by a procedure and placed, by ACMS, in the ACMS$L_STATUS field of the ACMS$PROCESSING_STATUS workspace
2. Gets a message from a file named by the MESSAGE FILE clause
3. Stores the message in the ACMS$T_STATUS_MESSAGE_LONG field of the ACMS$PROCESSING_STATUS workspace

To handle messages for a task, you may want to use a workspace that you define rather than the ACMS$PROCESSING_STATUS system workspace if:

- You want to return to the user information not related to the return status of a procedure
- You want to store literal message text instead of using message files
- You have other needs that are not completely met by the system workspace

When using workspaces that you define to handle errors, pass the workspace to the form or procedure and move values to a field in the workspace. You then test the workspace in the action part of the step to determine what to do next. This is how the tasks explained in *Chapter 2, "Defining Tasks"* use the QUIT_KEY field of a workspace to handle actions in exchange steps. You can handle actions in processing steps in the same way.

For example, you can add a field for storing information returned by the WRITE_RESERVE_PROC procedure to the record definition for ADD_RESERVE_WKSP used in the Add Car Reservation task in *Chapter 2, "Defining Tasks"*. *Example 4.1, "Record Definition for ADD_RESERVE_WKSP"* shows the new CDD definition for ADD_RESERVE_WKSP.

Example 4.1. Record Definition for ADD_RESERVE_WKSP

```
Definition of record ADD_RESERVE_WKSP
|   Contains field          WK_CONTROL
|   |  Datatype                 text size is 7 characters
|   Contains field          CUST_NUMBER

|   |  Datatype                 signed longword
|   Contains field          CUST_NAME
|   |  Datatype                 text size is 30 characters
|   Contains field          CUST_STREET_ADDRESS
|   |  Datatype                 text size is 30 characters

|   Contains field          CUST_CITY
|   |  Datatype                 text size is 20 characters
|   Contains field          CUST_STATE
|   |  Datatype                 text size is 2 characters

|   Contains field          CUST_ZIP
|   |  Datatype                 text size is 5 characters
|   Contains field          CUST_PHONE
|   |  Datatype                 text size is 10 characters
|   Contains field          CAR_TYPE
|   |  Datatype                 text size is 3 characters
|   Contains field          RENTAL_DATE
|   |  Datatype                 text size is 6 characters

|   Contains field          RETURN_DATE
|   |  Datatype                 text size is 6 characters
```

The WRITE_RESERVE_PROC procedure returns a value to WK_CONTROL, the new field you have added. Because the WK_CONTROL field is a 7-character text field, the WRITE_RESERVE_PROC procedure can return a literal string describing the success or failure of writing a new record to the Reservation file.

Now you add a workspace that you define for error handling to the USING part of the CALL clause in the processing step of the Add Car Reservation task:

```
PROCESSING
   CALL WRITE_RESERVE_PROC IN RESERVE_SERVER
     USING ADD_RESERVE_WKSP;
   CONTROL FIELD WK_CONTROL
     "FAILURE"  : GOTO PREVIOUS EXCHANGE;
   END CONTROL FIELD;
```

The WRITE_RESERVE_PROC procedure stores a value in the WK_CONTROL field of ADD_RESERVE_WKSP, and you use the CONTROL FIELD clause to test that value. Then you can return a message to the user. There are three ways to return messages to the user:

- Return error message text from the procedure code. In this case, the procedure moves the error message text into a workspace that is passed to the form. The form displays the message text like any other output.
- Use the GET ERROR MESSAGE clause in the task definition to move a message from a message file into a workspace. You can name the message symbol used for getting the message in the GET ERROR MESSAGE clause, or it can be the return status value of the procedure, placed by ACMS in the ACMS$PROCESSING_STATUS workspace by ACMS. See the *VSI ACMS for OpenVMS ADU Reference Manual* [https://docs.vmssoftware.com/vsi-acms-adu-ref-manual/]> for more information on the GET ERROR MESSAGE clause.
- Include literal error message text in the form, using an IF THEN ELSE response statement to determine whether or not to display the text or what text to display.

In this case, if the WRITE_RESERVE_PROC procedure returns the string FAILURE to the WK_CONTROL field, GOTO PREVIOUS EXCHANGE returns ACMS to the previous exchange. You can use the form in the previous exchange to display a message telling the user that the new car reservation record was not added to the file. The "FAILURE" in this example is literal error message text that displays when the WRITE_RESERVE_PROC procedure returns the string FAILURE to the WK_CONTROL field.

Example 4.2, "Task Definition for Add Car Reservation Task" shows the complete definition for the Add Car Reservation task using a workspace that you define for error handling.

Example 4.2. Task Definition for Add Car Reservation Task

```
REPLACE TASK ADD_CAR_RESERVATION_TASK
   WORKSPACES ARE ADD_RESERVE_WKSP, QUIT_CTRL_WKSP;

BLOCK WORK WITH FORM I/O IS

  GET_RENTAL_INFORMATION:
   EXCHANGE WORK IS
    RECEIVE FORM RECORD ADD_RESERVE_FORM_REC_LIS
     RECEIVING ADD_RESERVE_WKSP, QUIT_CTRL_WKSP;
   ACTION IS
    CONTROL FIELD IS QUIT_CTRL_WKSP.QUIT_KEY
    "QUIT"  :  EXIT TASK;
    END CONTROL FIELD;

  WRITE_RESERVATION_INFORMATION:
   PROCESSING WORK IS
    CALL WRITE_RESERVE_PROC IN RESERVE_SERVER
    USING ADD_RESERVE_WKSP;
```

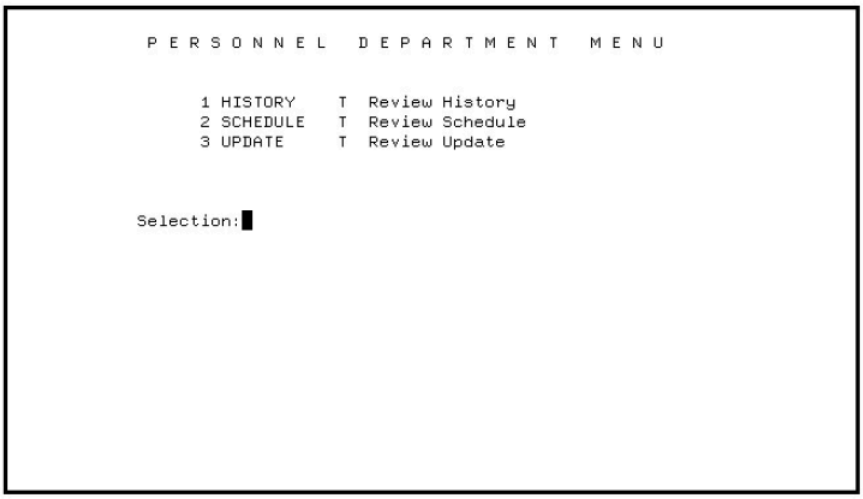
```
   ACTION IS
    CONTROL FIELD IS WK_CONTROL
    "FAILURE"  :  GOTO PREVIOUS EXCHANGE;
    END CONTROL FIELD;
END BLOCK WORK;
ACTION IS
   REPEAT STEP;
END DEFINITION;
```

4.3. Using Data Supplied at the Menu

You can use information that the user supplies when selecting a task from a menu as input to a task.

Users often must type the number or keyword of the task they want to run when selecting a task from a menu like the one shown in *Figure 4.1, "A Selection Menu"*.

Figure 4.1. A Selection Menu



To display the review history of an employee from the menu in *Figure 4.1, "A Selection Menu"*, the user types HISTORY in response to the Selection: prompt.

When a user selects a task from a menu, the first step of the task is often an exchange step asking the user to supply a key, usually something like an employee number or department number. Instead of asking the user for the key after making the selection, however, you can let the user supply that information *when making a selection*. ACMS stores this selection string in the system workspace, ACMS$SELECTION_STRING.

*Example 4.3, "Definition for the ACMS$SELECTION_STRING Workspace"* shows the record definition of the ACMS$SELECTION_STRING workspace.

Example 4.3. Definition for the ACMS$SELECTION_STRING Workspace

```
DEFINE RECORD DISK1:[CDDPLUS]ACMS$DIR.ACMS$WORKSPACES.ACMS
$SELECTION_STRING.
   ACMS$SELECTION_STRING STRUCTURE.
     ACMS$T_SELECTION_STRING   DATATYPE TEXT 255.
   END STRUCTURE.
END RECORD.
```

Like the ACMS$PROCESSING_STATUS system workspace, the ACMS$SELECTION_STRING workspace is automatically available to all tasks. In the ACMS$SELECTION_STRING workspace, ACMS stores any text containing up to 255 characters that the user types after typing the number or keyword of a task.

If you use the ACMS$SELECTION_STRING workspace to let the user type in a record key, the first exchange step of the task can check for that key in the ACMS$SELECTION_STRING workspace. The exchange step displays a DECforms panel asking for the key only if the workspace field is empty. Any validation work that the form normally performs is bypassed. If the field contains a key, ACMS does not do any work in the first exchange step of the task and goes directly to the second step.

Include the ACMS$SELECTION_STRING workspace in the first exchange step of the Display Basic task so that the user can type the number of the employee when selecting the task from a menu.

```
VALIDATE_EMPLOYEE:
   EXCHANGE
     CONTROL FIELD ACMS$T_SELECTION_STRING
       ""          :  RECEIVE FORM RECORD DISPLAY_NUMBER_FORM_REC
                       RECEIVING ACMS$SELECTION_STRING;
       NOMATCH     : NO EXCHANGE;
     END CONTROL FIELD;
```

You can also use the CONTROL FIELD clause to take conditional actions based on the contents of a workspace field. ACMS checks the field ACMS$T_SELECTION_STRING field in the VALIDATE_EMPLOYEE step. If the field is empty, the user did not supply an employee number when selecting the Display Basic task, and DISPLAY_BASIC_FORM displays a panel asking for the employee number. The form stores this number in the ACMS$SELECTION_STRING workspace.

At the end of the previously shown exchange step, the ACMS$SELECTION_STRING workspace always contains an employee number, typed in either as a selection string or in response to the initial panel. Because the processing step in the Display Basic task needs the employee number to know what record to get, you must pass the contents of the selection string workspace to the procedure in the processing step.

```
PROCESSING
   CALL DISPLAY_BASIC_GET
     USING ACMS$SELECTION_STRING, HIST_RECORD, PERS_RECORD;
   CONTROL FIELD ACMS$T_STATUS_TYPE
     "G"         : GOTO NEXT STEP;
     "B"         : GET ERROR MESSAGE;
                   GOTO PREVIOUS EXCHANGE;
   END CONTROL FIELD;
```

The DISPLAY_BASIC_GET procedure uses the employee number stored in the ACMS$SELECTION_STRING workspace to read basic employee information from both the History and Personnel files. You name the ACMS$SELECTION_STRING workspace in the USING part of the CALL clause, if the processing step uses information stored in the workspace.

The error handling in this processing step is the same as that in other tasks explained in this manual. If the DISPLAY_BASIC_GET procedure returns a recoverable error, ACMS:

1. Stores the value B in the ACMS$T_STATUS_TYPE field of the ACMS$PROCESSING_STATUS workspace

2. Repeats the first exchange, where the form uses the contents of the ACMS$T_STATUS_TYPE field to display an error message to the user, asking for another employee number

The DISPLAY_BASIC_GET procedure has to test and clear the ACMS$T_SELECTION_STRING workspace before using the form. If the workspace had contained a value when tested in the first exchange step, this would have put the task into an infinite loop.

Example 4.4, "Definition for Display Basic Task" shows the complete definition for the Display Basic task.

Example 4.4. Definition for Display Basic Task

```
REPLACE TASK DISPLAY_BASIC
   USE WORKSPACES ADMIN_WORKSPACE, HIST_RECORD, PERS_RECORD,
                  QUIT_CTRL_WKSP;
   DEFAULT SERVER IS ADMINISTRATION_SERVER;
   DEFAULT FORM IS DISPLAY_BASIC_FORM
   BLOCK
     WORK WITH NO SERVER CONTEXT FORM I/O

     VALIDATE_EMPLOYEE:
       EXCHANGE
         CONTROL FIELD ACMS$T_SELECTION_STRING
           ""          : RECEIVE FORM RECORD DISPLAY_NUMBER_FORM_REC
                           RECEIVING ACMS$SELECTION_STRING;
           NOMATCH     : NO EXCHANGE;
         END CONTROL FIELD;

       PROCESSING
         CALL DISPLAY_BASIC_GET
           USING ACMS$SELECTION_STRING, HIST_RECORD, PERS_RECORD;
         CONTROL FIELD ACMS$T_STATUS_TYPE
           "G"         : GOTO NEXT STEP;
           "B"         : GET ERROR MESSAGE;
                         GOTO PREVIOUS EXCHANGE;
         END CONTROL FIELD;

     DISPLAY_BASIC_DATA:
       EXCHANGE
         TRANSCEIVE FORM RECORD DISPLAY_BASIC_FORM_REC,
                                QUIT_CTRL_FORM_REC
           SENDING ADMIN_WORKSPACE
           RECEIVING QUIT_CTRL_WKSP;
         CONTROL FIELD IS QUIT_CTRL_WKSP.QUIT_KEY
           "QUIT"      : EXIT TASK;
         END CONTROL FIELD;
     END BLOCK WORK;
     ACTION
      REPEAT STEP;
  END DEFINITION;
```

When using the ACMS$SELECTION_STRING workspace to let the user supply a record key when selecting a task, you:

1. Name the ACMS$SELECTION_STRING workspace in the form definitions and in the steps using that workspace

2. Use the CONTROL FIELD clause in the first exchange step to do conditional work based on the value in the ACMS$T_SELECTION_STRING field

3. Do *not* name the ACMS$SELECTION_STRING workspace or any other ACMS system workspace in the WORKSPACES or USE WORKSPACES clause of task or task group definitions

When you use the contents of the ACMS$SELECTION_STRING workspace as input to a task, validation of that information, normally handled by the form, must be handled in the procedure code of the task.

4.4. Using Group Workspaces

Often there is some information that many tasks in a task group use. For example, in an accounting application there may be several tasks in a task group that always require the current interest rate. You can store this shared information in a **group workspace**.

Each task instance using a group workspace gets its own copy of the workspace. ACMS discards the copy of the workspace belonging to each task instance, but unless the workspace is being updated, ACMS keeps the original contents of a group workspace when a task instance is finished, as it does with the contents of a task workspace. This procedure allows many tasks or different instances of the same task to use the contents of a group workspace many times.

A Labor Reporting task shows one example of how to use a group workspace. This task lets the user type information about the work that an employee did on different projects. Part of the information used by the task each time it runs is the week-ending date. Rather than require the user to supply the date whenever typing information for an employee, you can store that date in a group workspace. The task can then access that workspace each time it runs.

Example 4.5, "Record Definition for GROUP_WORKSPACE" shows the record definition for LABOR_GROUP_WORKSPACE.

Example 4.5. Record Definition for GROUP_WORKSPACE

```
DEFINE RECORD
     DISK1:[CDDPLUS]ACMS$DIR.ACMS$EXAMPLES_RMS.LABOR_GROUP_WORKSPACE.
   LABOR_GROUP_WORKSPACE STRUCTURE.
     WK_ENDING_DATE    DATATYPE TEXT 6.
                       INITIAL VALUE "".
   END LABOR_GROUP_WORKSPACE STRUCTURE.
 END LABOR_GROUP_WORKSPACE.
```

Because the task is going to take a value from the workspace, you need a way to put the value into the workspace. There are several ways to initialize the contents of a group workspace:

- Run a task to load a value into the workspace when the application using the workspace is started.
- Check the contents of the workspace in the initial step of each task using the workspace. If the workspace is empty, either run a procedure or call a form in that step to initialize the workspace. However, when many tasks use the same group workspace, those tasks are not likely to have update access to that workspace.
- Check the contents of the workspace in the initial step of each task using that workspace. If the workspace is empty, use the MOVE clause in the action part of the step. You may use MOVE to initialize signed longword fields and text fields. If you want to initialize other data types, use one of the other methods discussed here.
- Check the contents of the workspace in the initial step of each task using the workspace. If the workspace is empty, invoke another task that initializes the workspace.

The Labor Reporting task uses the last of these methods to set the initial contents of the workspace. The first step of the task uses a CONTROL FIELD clause to test the contents of the workspace and, if the workspace does not contain the week ending date, runs another task, Get Initial Value, that initializes the workspace. You can run another task without returning the user to a selection menu by using the task-call-task feature, described in detail in *Chapter 5, "Using the Task-Call-Task Feature"*.

The Get Initial Value task consists of a single processing step that runs a procedure to initialize the group workspace.

The Labor Reporting task is the only task using the workspace named LABOR_GROUP_WORKSPACE, so you can declare this group workspace at either the group or task level. However, by declaring workspaces at the group level, you ensure that the same workspace name is used consistently by all tasks. You use the USE WORKSPACES clause in the task definition to specify that workspaces to be used in a task are declared at the group level:

```
USE WORKSPACES LABOR_GROUP_WORKSPACE, LABOR_RECORD, LABOR_WORKSPACE;
```

Here is the first step of the Labor Reporting task:

```
PROCESSING
   NO PROCESSING;
   CONTROL FIELD LABOR_GROUP_WORKSPACE.WK_ENDING_DATE
     ""          : GOTO TASK GET_INITIAL_VALUE_TASK;
     NOMATCH     : GOTO NEXT STEP;
   END CONTROL FIELD;
```

The WK_ENDING_DATE field must be defined to have an initial value of " ". If you do not define an initial value for the field, ACMS initializes it with zeros. If the CONTROL FIELD clause tests the field for " " and finds zeros, it always takes the action associated with the NOMATCH keyword, because the field is not empty.

If the WK_ENDING_DATE field is empty, ACMS runs INITIAL_VALUE_TASK. *Example 4.6, "Complete Definition for the Get Initial Value Task"* shows the definition for that task.

Example 4.6. Complete Definition for the Get Initial Value Task

```
REPLACE TASK GET_INITIAL_VALUE_TASK
   USE WORKSPACE LABOR_GROUP_WORKSPACE WITH UPDATE LOCK;
       PROCESSING
         CALL LABOR_GET_INITIAL_VALUE IN WORKSPACE_SERVER
           USING LABOR_GROUP_WORKSPACE;
         CONTROL FIELD ACMS$T_STATUS_TYPE
           "G"         : GOTO TASK LABOR_DATA_ENTRY_TASK;
           "B"         : CANCEL TASK;
         END CONTROL FIELD;
END DEFINITION;
```

The LABOR_GET_INITIAL_VALUE procedure gets the week-ending date from a file and stores it in the WK_ENDING_DATE field of the workspace LABOR_GROUP_WORKSPACE. When the week-ending date needs to be changed, the contents of the file must be changed, and the application or applications using that workspace must be stopped and restarted.

If the LABOR_GET_INITIAL_VALUE procedure is unsuccessful, ACMS cancels the task, returning the user to the menu. If the procedure is successful, ACMS processes the rest of the Labor Reporting task. When the initial processing step of that task runs, the LABOR_GROUP_WORKSPACE contains the week-ending date. The next step in the task can:

- Display a panel by using the contents of the workspace
- Do processing by using the contents of the workspace

In the Labor Reporting task, once the first processing step has checked the contents of the group workspace, an exchange step uses a form to display an initial panel. Here is the first exchange step of the Labor Reporting task:

```
EXCHANGE
   TRANSCEIVE FORM RECORD LABOR_GROUP_FORM_REC, EMPLOYEE_FORM_REC_LIS
    SENDING LABOR_GROUP_WKSP
    RECEIVING EMPLOYEE_RECORD_WKSP, QUIT_CTRL_WKSP;
ACTION IS
   CONTROL FIELD IS QUIT_CTRL_WKSP.QUIT_KEY
   "QUIT"  :  EXIT TASK;
   END CONTROL FIELD;
```

When this step is complete, the task runs like any other update task. The user supplies a record key to a form in this exchange step. A procedure in a processing step uses that key to read information from a file or database. A form in an exchange step displays that information for the user. The user can then type additional information. In the Labor Reporting task, the user types project information for an employee.

Example 4.7, "Definition for Labor Reporting Task" shows the complete definition for the Labor Reporting task.

Example 4.7. Definition for Labor Reporting Task

```
REPLACE TASK LABOR_DATA_ENTRY_TASK
   USE WORKSPACES
     EMPLOYEE_RECORD_WKSP, LABOR_GROUP_WKSP, LABOR_RECORD_WKSP,
     QUIT_CTRL_WKSP;
   DEFAULT SERVER IS LABOR_SERVER;
   DEFAULT FORM IS LABOR_FORM
   BLOCK
     WORK WITH FORM I/O

       PROCESSING
         NO PROCESSING;
          CONTROL FIELD LABOR_GROUP_WKSP.WK_ENDING_DATE
           ""          : GOTO TASK GET_INITIAL_VALUE_TASK;
            NOMATCH    : GOTO NEXT STEP;
         END CONTROL FIELD;


       EXCHANGE
         TRANSCEIVE FORM RECORD LABOR_GROUP_FORM_REC,
                                EMPLOYEE_FORM_REC_LIS
          SENDING LABOR_GROUP_WKSP
          RECEIVING EMPLOYEE_RECORD_WKSP, QUIT_CTRL_WKSP;
       ACTION IS
         CONTROL FIELD IS QUIT_CTRL_WKSP.QUIT_KEY
         "QUIT"  :  EXIT TASK;
         END CONTROL FIELD;


      PROCESSING
         CALL LABOR_EMPLOYEE_GET USING EMPLOYEE_RECORD_WKSP;
         CONTROL FIELD ACMS$T_STATUS_TYPE
```

```
          "B"        : GET ERROR MESSAGE;
                       GOTO PREVIOUS EXCHANGE;
          "G"        : GOTO NEXT STEP;
        END CONTROL FIELD;

    GET_PROJECT_DATA:
      EXCHANGE
        TRANSCEIVE FORM RECORD LABOR_FORM_REC, LABOR_FORM_REC_LIS
          SENDING LABOR_RECORD_WKSP
          RECEIVING LABOR_RECORD_WKSP, QUIT_CTRL_WKSP;
      ACTION IS
        CONTROL FIELD IS QUIT_CTRL_WKSP.QUIT_KEY
        "QUIT"  :  EXIT TASK;
        END CONTROL FIELD;

      PROCESSING
        CALL LABOR_PROJECT_PUT
          USING LABOR_GROUP_WORKSPACE, LABOR_RECORD_WKSP;
        CONTROL FIELD ACMS$T_STATUS_TYPE
          "B"        : GET ERROR MESSAGE;
                       GOTO PREVIOUS EXCHANGE;
        END CONTROL FIELD;

    END BLOCK WORK;
    ACTION
      REPEAT STEP;
END DEFINITION;
```

4.5. Using User Workspaces

Just as group workspaces let you keep information for many users in many tasks or instances of the same task, **user workspaces** let you store information used by a single user in many tasks or many task instances.

Because a user workspace is specific to a single user rather than to tasks in an application, you do not run a task at application startup to initialize a user workspace (as you would for a group workspace). Instead, you can include an initial processing step that tests the contents of the workspace. If the workspace is empty, a procedure or form gets information and writes it to the workspace.

The user may need to see several kinds of information about an employee, for example. The user must select a separate task to see each kind of information. To see information about the employee's education, the user selects the Display Education task. To see family information, the user selects the Display Family task.

In each of these tasks, the user begins by typing in the employee number. Because all the information displayed by different tasks is about a single employee in this case, you do not want the user to have to supply the employee number for each task selection. Instead you can use the contents of a user workspace in the initial step of each task. Displaying those contents allows the user either to change the employee number or to press **Return** to see information about the employee.

Example 4.8, "Record Definition for DISPLAY_USER_WKSP" shows the record definition for DISPLAY_USER_WKSP.

Example 4.8. Record Definition for DISPLAY_USER_WKSP

```
DEFINE RECORD
```

```
        DISK1:[CDDPLUS]ACMS$DIR.ACMS$EXAMPLES_RMS.DISPLAY_USER_WKSP.
   DISPLAY_USER_WKSP STRUCTURE.
     EMP_NUMBER    DATATYPE SIGNED LONGWORD.
   END DISPLAY_USER_WKSP STRUCTURE.
END DISPLAY_USER_WKSP.
```

This workspace has a single field for the number of the employee whose records the user wants to see.

The first exchange step of each of the display tasks can use this workspace to display the current employee number. Here is the first exchange step of the Display Basic task:

```
VALIDATE_EMPLOYEE:
   EXCHANGE
     TRANSCEIVE FORM RECORD DISPLAY_NUMBER_FORM_REC,
                            DISPLAY_NUMBER_FORM_REC_LIS
       SENDING DISPLAY_USER_WKSP
       RECEIVING DISPLAY_USER_WKSP, QUIT_CTRL_WKSP;
     CONTROL FIELD QUIT_CTRL_WKSP.QUIT_KEY
       "QUIT"  :  EXIT TASK;
     END CONTROL FIELD;
```

This exchange step displays the current employee number, and lets the terminal user change the employee number to see information about a different employee. The processing step then uses the DISPLAY_BASIC_GET procedure to retrieve records for the employee whose number is in DISPLAY_USER_WKSP.

Example 4.9, "Definition for Display Basic Task with User Workspace" shows the complete definition for the Display Basic task using a user workspace.

Example 4.9. Definition for Display Basic Task with User Workspace

```
REPLACE TASK DISPLAY_BASIC
   USE WORKSPACES
     ADMIN_WORKSPACE,
     DISPLAY_USER_WKSP WITH ACCESS UPDATE,
     HIST_RECORD, PERS_RECORD;
   DEFAULT SERVER IS ADMINISTRATION_SERVER;
   DEFAULT FORM IS DISPLAY_BASIC_FORM;
   BLOCK
     WORK WITH NO SERVER CONTEXT FORM I/O

     VALIDATE_EMPLOYEE:
       EXCHANGE
     TRANSCEIVE FORM RECORD DISPLAY_NUMBER_FORM_REC,
                            DISPLAY_NUMBER_FORM_REC_LIS
       SENDING DISPLAY_USER_WKSP
       RECEIVING DISPLAY_USER_WKSP, QUIT_CTRL_WKSP;
     ACTION IS
       CONTROL FIELD QUIT_CTRL_WKSP.QUIT_KEY
         "QUIT"  :  EXIT TASK;
       END CONTROL FIELD;

       PROCESSING
         CALL DISPLAY_BASIC_GET
           USING ADMIN_WORKSPACE, DISPLAY_USER_WKSP,
                 HIST_RECORD, PERS_RECORD;
         CONTROL FIELD ACMS$T_STATUS_TYPE
           "G"        : GOTO NEXT STEP;
```

```
          "B"        : GET ERROR MESSAGE;
                       GOTO PREVIOUS EXCHANGE;
        END CONTROL FIELD;

    DISPLAY_BASIC_DATA:
      EXCHANGE
        TRANSCEIVE FORM RECORD DISPLAY_BASIC_FORM_REC, QUIT_CTRL_FORM_REC
          SENDING ADMIN_WORKSPACE
          RECEIVING QUIT_CTRL_WKSP;
      ACTION IS
        CONTROL FIELD QUIT_CTRL_WKSP.QUIT_KEY
          "QUIT"  :  EXIT TASK;
        END CONTROL FIELD;
    END BLOCK WORK;
    ACTION
      REPEAT STEP;
END DEFINITION;
```

You can test the contents of the user workspace, just as you would the contents of a group workspace. For example, in the Display Basic task, you can test the contents of the workspace in the initial exchange step. If the workspace is empty, you can use a form to display a panel asking for an employee number.

In the final steps of the task, you can use function keys to let the user clear or save the contents of the user workspace. To test the EMP_NUMBER field, you must be careful to redefine it as a text field, or you get a "datatype mismatch" error when you build the task group. If you change the data type of EMP_NUMBER to text, the procedure must then convert that information to a numeric datatype before moving the key to the personnel record.

You might want to let the user:

- Return to the menu with no value in the user workspace
- Return to the menu, keeping the current value in the user workspace
- Repeat the task, keeping the current value in the user workspace
- Repeat the task with no value in the user workspace

The following example shows the final exchange steps:

```
DISPLAY_BASIC_DATA:
   EXCHANGE
         TRANSCEIVE FORM RECORD DISPLAY_BASIC_FORM_REC,
                                QUIT_CTRL_FORM_REC
           SENDING ADMIN_WORKSPACE
           RECEIVING QUIT_CTRL_WKSP;
         CONTROL FIELD  QUIT_CTRL_WKSP.CTRL_KEY
           "NEXT"   :  GOTO NEXT STEP;
           "QUIT"   :  EXIT TASK;
           "REPET"   :  EXIT BLOCK;
           NOMATCH   :  GOTO NEXT STEP;
         END CONTROL FIELD;
   EXCHANGE
         RECEIVE FORM RECORD CLEAR_WKSP_FORM_REC_LIS
           RECEIVING DISPLAY_USER_WKSP, QUIT_CTRL_WKSP;
         CONTROL FIELD  QUIT_CTRL_WKSP.CTRL_KEY
           "QUIT"   :  EXIT TASK;
           NOMATCH   :  EXIT BLOCK;
```

```
         END CONTROL FIELD;
ACTION
   REPEAT TASK;
```

In the first of these exchange steps, the user presses either one of three function keys or **Return**.

To return to the menu and discard the current employee number, the user presses GOLD-E, which returns the value "NEXT" to the CTRL_KEY field. The CLEAR_WKSP_FORM_REC in the final exchange step clears the employee number from DISPLAY_USER_WKSP.

To return to the menu and keep the same employee number, the user presses GOLD-D, which returns the value "QUIT" to the CTRL_KEY field. ACMS exits the task and returns the user to the menu, but ACMS does not clear the employee number from DISPLAY_USER_WKSP.

To repeat the task by using the same employee number, the user presses GOLD-R, which returns the value "REPET" to the CTRL_KEY field. ACMS processes the block action, REPEAT STEP, without clearing the employee number from DISPLAY_USER_WKSP.

To repeat the task with a new employee number, the user presses **Return**. The CLEAR_WKSP_FORM_REC clears the employee number from DISPLAY_USER_WKSP, and ACMS processes the block action REPEAT STEP.

Keeping a value such as an employee number in a user workspace when a task instance is finished lets you write other task definitions that can use that value.

4.6. Moving Data to a Workspace Field

One method of moving data into a workspace field is by using the ADU MOVE clause. The MOVE clause specifies that ADU moves one of the following elements into a workspace field:

- Number
- Numeric value of a global symbol
- Workspace field
- Quoted string

Note

ACMS does not let you pass arrays to workspace fields.

Example 4.10, "Moving Data to a Workspace Field" demonstrates the use of the MOVE clause.

Example 4.10. Moving Data to a Workspace Field

```
SET VERIFY
REPLACE TASK TASK01
WORKSPACES ARE
   WKSP_1, WKSP_2;
PROCESSING IS
  CALL NULL_TRANSACTION   IN TEST_SERVER
   USING WKSP_1;
  MOVE   RMS$_EOF INTO WKSP_1.L_MESSAGE,
         -2 INTO WKSP_1.L_NUMBER,
```

```
          "GOOD"    INTO   ( WKSP_1.T_TEXT,WKSP_1.T_STRING ),
          WKSP_2.V_BASIC    INTO WKSP_1.V_EMPL ;
END DEFINITION;

BUILD GROUP  TEST_GROUP /SYSSHR -
                         /SYSLIB
```

Example 4.10, "Moving Data to a Workspace Field" uses the MOVE clause to move information from various sources to fields in the workspace called WKSP_1. The workspace fields in this example receive data in the following form:

- Numeric value of a global symbol, RMS$_EOF
- Number –2
- Quoted string "GOOD"
- Workspace field WKSP_2.V_BASIC

The following restrictions apply to MOVE clauses:

- You must use the keywords INTO or TO.
- If the source is a global symbol, the target must be of type signed longword.
- You cannot use two MOVE clauses in the same ACTION step.

You can also use the MOVE clause in CONTROL FIELD and SELECT FIRST clauses. For example:

```
SELECT FIRST TRUE
       (WK_VALUE EQL 1): MOVE 2 TO NUMBER;
       (WK_VALUE EQL 2): MOVE 3 TO NUMBER;
```

4.7. Passing Data with User-Written Agents

Another method of passing information to an ACMS task is by using a Systems Interface (SI) agent or another task. An SI agent can call an ACMS task and optionally pass workspaces to that task for read, write, or modify purposes. The called task can return the modified workspace back to the agent at the end of the task instance. The agent can then use the contents of this workspace in subsequent task executions. The agent passes workspaces in an argument list on either the ACMS$CALL or ACMS$START_CALL services. The agent can pass task workspaces only.

See *VSI ACMS for OpenVMS Systems Interface Programming* [https://docs.vmssoftware.com/vsi-acms-sys-interface-prog/] for information about passing arguments from an SI agent to an ACMS task.

4.8. Using External Global Symbols in Task Definitions

Task definitions can reference external global symbols in place of a workspace field reference in the following instances (RMS$_EOF and RMS$_OK_DUP represent external global symbols):

- SELECT FIRST clause. For example:

  ```
  SELECT FIRST (RMS$_EOF = numeric_workspace_field):
  ```

```
(numeric_workspace_field = RMS$_EOF):
```

- CANCEL TASK RETURNING clause:

  ```
  CANCEL TASK RETURNING RMS$_EOF;
  ```

- EXIT TASK RETURNING clause:

  ```
  EXIT TASK RETURNING RMS$_OK_DUP;
  ```

- MOVE clause:

  ```
  MOVE RMS$_EOF TO numeric_workspace_field;
  ```

- GET ERROR MESSAGE NUMBER clause:

  ```
  GET ERROR MESSAGE NUMBER RMS$_EOF;
  ```

Global symbols have a signed longword data type.

ACMS resolves global symbols during **BUILD GROUP** processing, where ADU searches for a matching workspace field reference. If it does not find a matching reference, ADU then searches the object modules and libraries that you specified on the **BUILD GROUP** command line with the **/SYSLIB**, **/SYSSHR**, **/OBJECT**, and **/USERLIBRARY** qualifiers and takes the following actions:

- If it finds a matching symbol, ADU stores the numeric value of the global symbol in the .TDB file.
- If it does not find a numeric workspace field or a global symbol, ADU issues an error message stating that there is an unresolved field reference.

ADU does *not* use the MESSAGE FILES ARE clause to resolve global symbols at build time.

Chapter 5. Using the Task-Call-Task Feature

In processing steps, you can call another task in the same task group, instead of calling a procedure in a server. This capability is called the **task-call-task** feature.

Task calling extends the capabilities of an ACMS task definition. Applications of the task-call-task feature include:

- Customized menus

  You can create menus that list only the tasks a user can select, allow a user to select a series of tasks at once, or provide additional choices without returning to a main menu.

- Enhanced security checking

  You can call tasks to perform specialized security checking in addition to default ACMS security checks.

- Common library tasks

  You can define commonly used tasks once and then call them from many tasks, like a subroutine library.

- Branching to another task

  You can define a function key to branch to an inquiry task during an update task, suspending the update task until the inquiry task completes.

See *VSI ACMS for OpenVMS Concepts and Design Guidelines* [https://docs.vmssoftware.com/vsi-acms-concepts-and-design-guide/] for further information on these applications of the task-call-task feature.

5.1. Calling Another Task

The task-call-task feature is based on the subroutine call-and-return model. You call another task in basically the same way that you make a subroutine call to a server procedure. Control passes to the task you call (along with whatever workspaces you name as parameters) and then returns to the place from which you made the call.

A calling task is referred to in this section as a **parent task**. A called task can become a parent task if it calls another task in turn.

A called task can return a status value to a parent task similarly to the way a routine in a procedure server can return a status value to a task. A parent task can use this value, or the contents of a task workspace field modified by the called task, to control subsequent task execution.

You must pay particular attention to the rules for passing workspaces. See *Section 5.1.3, "Passing Workspaces"* for further information on passing workspaces in task calling.

There is no limit to the number of task calls you can make other than that imposed by the maximum number of task instances specified in the application definition, and resource limits such as the task workspace pool and virtual memory inside the Application Execution Controller (EXC) process.

5.1.1. Defining a Task Call

To create a task call, you specify a task name instead of a server procedure. For example:

```
REPLACE TASK ENTER_ORDER
.
.
.
   PROCESSING
     CALL TASK ORDER_LINES USING
       CUSTOMER_INFO_RECORD,
       CUSTOMER_ACCOUNT_RECORD,
       ORDER_DATA_RECORD;
.
.
.
```

In this example, the processing step calls the ORDER_LINES task, passing the CUSTOMER_INFO_RECORD, CUSTOMER_ACCOUNT_RECORD, and ORDER_DATA_RECORD workspaces. The task name specified in a CALL TASK clause must be a group task name, not an application task name.

You must also specify the workspaces that the called task will receive as arguments.

```
REPLACE TASK ORDER_LINES

WORKSPACES ARE
     CUSTOMER_INFO_RECORD,
     CUSTOMER_ACCOUNT_RECORD,
     ORDER_DATA_RECORD,
     TASK_CONTROL_RECORD;

TASK ARGUMENTS ARE
     CUSTOMER_INFO_RECORD WITH ACCESS READ,
     CUSTOMER_ACCOUNT_RECORD WITH ACCESS MODIFY,
     ORDER_DATA_RECORD WITH ACCESS WRITE;
```

The ENTER_ORDER task controls the entry of general information about the order (such as customer name and address, shipping address, type of order, sales representative, and so on) and then passes control to the ORDER_LINES task. The ORDER_LINES task receives customer information in the CUSTOMER_INFO_RECORD, updates the customer's accounts information in the CUSTOMER_ACCOUNT_RECORD, and returns line totals in the ORDER_DATA_RECORD.

Workspaces you use as task call arguments must be defined as task workspaces (the default) in the called task. Workspaces received by the called task must be listed in the TASK ARGUMENTS statement in the order in which the calling task passes them. You can specify MODIFY, READ, or WRITE access to those workspaces for the calling task. Use MODIFY to pass and return data, READ to pass data only, and WRITE to return data only.

You must be careful not to unintentionally overwrite data when accessing group and user workspaces from both the parent task and the called task. See *Section 5.1.3, "Passing Workspaces"* for further information on using workspaces.

5.1.2. Task Call Example

This section presents a complete example of how to use the task calling feature. The example shows a simple order-entry transaction. The example is limited to ACMS task definition syntax, with one exception: a BASIC program the order-detail task uses.

Example 5.1, "Task ENTER_ORDER" shows the ACMS task definition syntax that defines the order-header portion of the order-entry transaction.

Example 5.1. Task ENTER_ORDER

```
REPLACE TASK ENTER_ORDER

WORKSPACES ARE
     HEADER_DATA_WSP,
     MSG_WKSP,
     ORDER_DATA_RECORD,
     TASK_CTL_WSP;

DEFAULT SERVER IS ORDER_SERVER;
DEFAULT FORM IS ORDER_FORM;
GLOBAL;

BLOCK WORK WITH FORM I/O

     EXCHANGE
         RECEIVE FORM RECORD HEADER_DATA_FORM_REC_LIS
           RECEIVING HEADER_DATA_WSP, TASK_CTL_WSP;
         ACTION IS
           CONTROL FIELD TASK_CTL_WSP.TASK_CTL_FIELD
           "QUIT"  :  EXIT TASK;
           END CONTROL FIELD;

     PROCESSING
         CALL WRITE_ORDER_HEADER_DATA USING HEADER_DATA_WSP;

DO_ORDER_LINE:
     PROCESSING
         CALL TASK PROCESS_ORDER_LINE USING
             ORDER_DATA_RECORD,
             TASK_CTL_WSP, MSG_WKSP;
         ACTION IS
           CONTROL FIELD TASK_CTL_WSP.TASK_CTL_FIELD
           "QUIT"  :  MOVE "Order completed successfully" TO
                       MSG_WKSP.MESSAGE_PANEL;
                       GOTO STEP END_ORDER;
           END CONTROL FIELD;

     PROCESSING
         CALL INCREMENT_ORDER_TOTALS USING
             ORDER_DATA_RECORD,
             HEADER_DATA_WSP;
         ACTION IS
           GOTO STEP DO_ORDER_LINE;

END_ORDER:
     EXCHANGE
```

```
        SEND FORM RECORD MSG_WKSP_FORM_REC
          SENDING MSG_WKSP;

END BLOCK WORK;
```

The first exchange step in *Example 5.1, "Task ENTER_ORDER"* displays the order entry panel and uses the HEADER_DATA_WSP to store the data the user types in. If the TASK_CTL_FIELD contains the word "QUIT" (which the user can place there by using a DECforms-defined function key), the task exits. Otherwise, the first processing step calls a server procedure to write the contents of HEADER_DATA_WSP to a file.

The second processing step calls the PROCESS_ORDER_LINE task, passing the ORDER_DATA_RECORD and TASK_CTL_WSP workspaces. Control passes from this point in the task definition to the PROCESS_ORDER_LINE task (see *Example 5.2, "Task PROCESS_ORDER_LINE"*).

When control returns to the ENTER_ORDER task, the TASK_CTL_FIELD is again checked for the word "QUIT". If the user did not press the QUIT function key, the third processing step in the ENTER_ORDER task increments the order totals and passes control back to the DO_ORDER_LINE step, which calls the PROCESS_ORDER_LINE task to input the next line of the order.

This sequence continues until the user presses the QUIT function key and control passes to the second, and final, exchange step.

Note

Whenever the user presses the QUIT key, all tests for the word "QUIT" in both task definitions evaluate to TRUE. This is because the PROCESS_ORDER_LINE task accepts the TASK_CTL_WSP workspace using WRITE access; when control returns to the ENTER_ORDER task, ACMS writes the contents of that workspace into the parent task's workspace.

Example 5.2, "Task PROCESS_ORDER_LINE" shows the ACMS task definition syntax for the order-detail portion of the order-entry transaction.

Example 5.2. Task PROCESS_ORDER_LINE

```
REPLACE TASK PROCESS_ORDER_LINE

WORKSPACES ARE
     DATA_RECORD,
     MSG_WKSP,
     ORDER_DATA_RECORD,
     TASK_CTL_WSP;

DEFAULT SERVER ORDER_SERVER;
DEFAULT FORM ORDER_FORM;

LOCAL;

TASK ARGUMENTS ARE
     ORDER_DATA_RECORD WITH ACCESS MODIFY,
     TASK_CTL_WSP WITH ACCESS WRITE;
     MSG_WKSP WITH ACCESS WRITE;
```

```
BLOCK WORK WITH FORM I/O

     EXCHANGE
         RECEIVE FORM RECORD INPUT_DATA_FORM_REC_LIS
           RECEIVING DATA_RECORD, TASK_CTL_WSP;
         ACTION IS
             CONTROL FIELD TASK_CTL_WSP.TASK_CTL_FIELD
             "QUIT"  :  EXIT TASK;
             END CONTROL FIELD;

     PROCESSING
         CALL WRITE_ORDER_LINE USING
             DATA_RECORD,
             ORDER_DATA_RECORD,
             TASK_CTL_WSP;
         ACTION IS
             SELECT FIRST TRUE
             ( TASK_CTL_FIELD = "DUPL" ): GOTO PREVIOUS EXCHANGE;
             NOMATCH  : MOVE "Order line record successfully
                         added to database" TO MSG_WKSP.MESSAGE_PANEL;
             END SELECT;

     EXCHANGE
         SEND FORM RECORD MSG_WKSP_FORM_REC
           SENDING MSG_WKSP;
         ACTION IS
             EXIT TASK;

END BLOCK WORK;
```

The PROCESS_ORDER_LINE task is a LOCAL task, which means that it can be called only from another task. It cannot be initiated by using a menu selection. You assign the LOCAL attribute in the application definition.

The TASK ARGUMENTS clause lists the workspaces the PROCESS_ORDER_LINE task can receive when called from another task. ORDER_DATA_RECORD is defined as a modify-access workspace; that is, any data written to it by the PROCESS_ORDER_LINE task can be modified, and is returned to the calling task. TASK_CTL_WSP and MSG_WKSP are defined as write-only workspaces; that is, no data are passed into them from the calling task, but any data written to them by the PROCESS_ORDER_LINE task is returned to the calling task. This way of passing workspaces is more efficient than using MODIFY (or read-write) access.

The PROCESS_ORDER_LINE task calls a DECforms form to display the order-lines panel and accept data that the user types in. After accepting data from the user, the task calls the WRITE_ORDER_LINE procedure (see *Example 5.3, "Procedure WRITE_ORDER_LINE (in BASIC)"*) by using data from the order header as well as data from the order line.

The WRITE_ORDER_LINE procedure checks for a duplicate key value and returns to the first exchange step if it finds one. If the order line is written successfully to the file, a second exchange step writes a success message, the task exits, and control returns to the ENTER_ORDER task.

When the user presses the QUIT key, the task exits, and control returns immediately to the ENTER_ORDER task, without calling the server procedure or the second exchange step.

Example 5.3, "Procedure WRITE_ORDER_LINE (in BASIC)" shows the BASIC program that writes the order-line information to a file.

Example 5.3. Procedure WRITE_ORDER_LINE (in BASIC)

```
10 SUB WRITE_ORDER_LINE( DATA_RECORD DATA_REC,            &
                         ORDER_DATA_RECORD ORD_VALUE_REC, &
                         TASK_CTL_WSP TASK_CTL )
    %INCLUDE %FROM %CDD "DATA_RECORD"
    %INCLUDE %FROM %CDD "ORD_VALUE_REC"
    %INCLUDE %FROM %CDD "TASK_CTL_WSP"
    MAP (DATA_FILE) DATA_RECORD FILE_REC

    FILE_REC = DATA_REC             ! Transfer data to file buffer
    ON ERROR GOTO 20                ! Set up an error trap
    PUT #1%                         ! Store the data
    ORD_VALUE_REC::ORD_LINE_TOTAL = DATA_REC::PRICE * DATA_REC::QTY
                                    ! Calculate order line total
    TASK_CTL::TASK_CTL_FIELD = "OK" ! It all worked so return success
    EXIT SUB                        ! And exit

20 IF ERR = 134% THEN              ! If duplicate key detected then
      TASK_CTL::TASK_CTL_FIELD = "DUPL" ! Return failure indicator
      RESUME 99                     !     And exit

    ELSE                            ! Otherwise
      CALL ACMSAD$REQ_CANCEL        !     Something serious occurred
    END IF                          !     so cancel the task.

99 END SUB
```

The procedure shown in *Example 5.3, "Procedure WRITE_ORDER_LINE (in BASIC)"* moves the data from the workspace to the file buffer and then writes a record to the file. If the write operation succeeds, the procedure returns a success code in a task control workspace. If it detects a duplicate key, which indicates that a duplicate item exists in the same order, the procedure returns a failure code to the task. Any other error causes the procedure to cancel the task.

If there are any unexpected errors (such as a device-full RMS error) or the server process terminates unexpectedly while the WRITE_ORDER_LINE procedure is running, ACMS automatically cancels the PROCESS_ORDER_LINE task. When a called task is canceled, ACMS by default also cancels the parent task.

In this example, ACMS automatically cancels the ENTER_ORDER task under the following conditions:

- If any unexpected errors occur, such as a device-full RMS error
- If the server process terminates unexpectedly while the INCREMENT_ORDER_TOTALS procedure is running
- If the server process terminates unexpectedly while the PROCESS_ORDER_LINE task is executing and has called the WRITE_ORDER_LINE_ROUTINE procedure

The ENTER_ORDER task does not need to check the status from the call to the PROCESS_ORDER_LINE task to determine whether the task has failed. The ENTER_ORDER task does not expect the called task to return anything other than a success status and uses the contents of the TASK_CTL_FIELD workspace field to control task execution. If the PROCESS_ORDER_LINE task fails, the ENTER_ORDER task expects to be canceled. For example, the ENTER_ORDER task expects to be canceled if:

- A user presses **Ctrl/Y** while entering order line data.

- An operator uses an **ACMS/CANCEL TASK** command to cancel the PROCESS_ORDER_LINE task.

When a called task is canceled, by default the calling task also is canceled. For information on how to override this default so that ACMS does not automatically cancel the calling task when the called task is canceled, see *Chapter 8, "Handling Task Execution Errors"*.

5.1.3. Passing Workspaces

This section presents the rules for passing ACMS workspaces to and from a task you call from another task.

ACMS allows a called task to accept arguments in the form of task workspaces. User-written agents or tasks may pass workspaces to other tasks.

Note

The rules for passing workspaces when you call the task from an agent are the same as the rules for passing workspaces when you call the task from another task. See *VSI ACMS for OpenVMS Systems Interface Programming* [https://docs.vmssoftware.com/vsi-acms-sys-interface-prog/] for information on writing an agent program.

A task passes workspaces to a called task by position. That is, the contents of the first workspace identified by the USING phrase in the parent task is moved to the first workspace named in TASK ARGUMENTS clause in the called task, and so on, for each workspace named in the USING phrase. This is the same as the method for passing workspaces to routines in procedure servers or to DECforms forms.

When calling a task, ACMS compares the length of each workspace to be passed with the length of the corresponding workspace in the called task. If any do not match, ACMS cancels both the parent and the called task. However, if you transpose two workspaces of the same length, ACMS does not flag this reversal.

Warning

When the task-call-task feature is used, it is possible to see the following error in SWL:

```
ACMSWSP-E-NONESUCH, INTERNAL ERROR: SPECIFIED BLOCK DOES NOT EXIST
```

This error could be the result of declaring a workspace in the child task, but not using it. If the workspace is not used, remove the declaration from the child task and remove the workspace from the workspace list in the task call in the parent task.

A called task can accept arguments using task workspaces only. You cannot pass arguments into group, user, or system workspaces. Special rules apply for accessing group, user, and system workspaces in called tasks.

You can omit a workspace when you call a task. If you do, ACMS initializes the workspace in the called task with its default contents from the CDD workspace definition stored in the task database (.TDB file). If there are no default contents, ACMS initializes the workspace with zeros.

You can specify READ, WRITE, or MODIFY access for each workspace you include in a TASK ARGUMENT clause. Use MODIFY for passing and returning data, READ for passing data, and WRITE for returning data:

- MODIFY access (the default)

  ACMS passes data supplied by the parent task. If the parent task does not pass any data, ACMS initializes the workspace with its default contents or zeros. Data is returned to the parent task when the called task completes.

- READ access

  ACMS passes data supplied by the parent task. If the parent task does not pass any data, ACMS initializes the workspace with its default contents or zeros. The called task can modify the contents of the workspace; however, no data is returned to the parent task when the called task completes.

- WRITE access

  ACMS initializes the workspace defined with its default contents or zeros. Data is returned to the parent task when the called task completes.

Specifying READ access on task workspace arguments can provide performance benefits for tasks calling other tasks, because ACMS does not have to update the workspace in the parent task when the called task completes.

In creating workspaces for task calls, bear in mind the trade-off between workspace size and the various access types. For large workspaces, it is more efficient to pass data using a READ workspace and return data using a WRITE workspace than it is to pass and return data in a single MODIFY workspace. Conversely, it is more efficient to pass a single MODIFY workspace containing a small amount of data than it is to pass several separate READ and WRITE workspaces.

You may specify any workspace type (task, user, group, or system) as an argument and pass it to the called task with the USING phrase. However, note that the parent task may supply only workspaces to task workspaces in the called task. See *Section 5.1.3.2, "Handling User and Group Workspaces"* for further information on accessing group and user workspaces.

5.1.3.1. Using System Workspaces

Each task instance owns its own copy of each of the three ACMS system workspaces. Once a called task instance starts, these copies are totally independent of a parent task.

Normally, you do not pass a system workspace from a parent task to a called task. Instead, move the data you need from the system workspace into a task workspace, and then pass it. You can move data by using either a MOVE clause or a processing step.

If you find you must pass a system workspace to a called task, specify READ access in the TASK ARGUMENT clause. This protects the contents of the system workspace from accidental modification.

The following rules apply to the three ACMS system workspaces when they are passed by default to a called task:

- ACMS initializes the ACMS$TASK_INFORMATION system workspace with information about the task instance and the task submitter.

  The ACMS$L_CALL_SEQUENCE_NUMBER field contains the sequence number of the current called task. When an agent (ACMS menu or user-written) calls a task, ACMS initializes this field to zero. Each time a task calls another task, ACMS increments this field by 1. Therefore, the first called task has a call sequence number of 1. If that task calls another task, or if that task returns and the

original task calls another task, the sequence number increments again and the new called task has a call sequence number of 2. This sequence number resets to zero each time an agent calls a task.

- ACMS always initializes the ACMS$PROCESSING_STATUS system workspace to reflect a success status and to contain a blank status message field.
- ACMS copies the selection string argument from the ACMS$START_CALL argument list to the ACMS$SELECTION_STRING system workspace. If no selection string is supplied, ACMS initializes the workspace with spaces.

  If a task calls another task, ACMS initializes the selection string system workspace with the contents of the selection string system workspace from the new parent task.

  ACMS does not return the contents of the ACMS$SELECTION_STRING system workspace to the parent task when the called task completes. If you need to return the contents of the ACMS$SELECTION_STRING system workspace to a parent task, move it to a task workspace and transfer it from the task workspace into the ACMS$SELECTION_STRING workspace after control returns to the parent task.

5.1.3.2. Handling User and Group Workspaces

A called task can access the same group and user workspaces that a parent task can access. You must be careful when updating group and user workspaces shared by a parent and called task, because it is possible to overwrite data and store incorrect results.

If both parent and called tasks require only read access to a user or a group workspace, then you can write both tasks as you normally do. However, because ACMS updates only the master copies of user and group workspaces at the end of a task instance, you must consider the case where either the parent task or the called task must update a user or group workspace that both tasks reference and use.

Note

If a parent and called task access the same workspace for update by using the WITH LOCK clause, ACMS cancels the called task.

When a parent and a called task both attempt to update the contents of a user or group workspace, the following occurs:

1. The called task completes and ACMS updates the master copy of the workspace.
2. The parent task completes and ACMS updates the master copy of the workspace, thereby overwriting the previous contents stored by the called task.

Data from the called task can be lost this way.

The best method of handling a group or user workspace is to pass the workspace from a parent task into a task workspace of the same layout in the called task (see *Example 5.5, "Passing User Workspaces to Menu Tasks"*). This way, when the parent task completes, the master copy of the user workspace updates correctly. This method is also more efficient than having each called task individually access the user workspace.

5.1.3.3. Tasks You Also Select from a Menu

You must give special consideration to accessing group and user workspaces in tasks that can be selected from a menu and that can also be called by other tasks.

You can test whether the task is being initiated by a menu or called from another task. If it is being called from another task, move the user or group workspace you want to use into a task workspace of the same layout. *Example 5.5, "Passing User Workspaces to Menu Tasks"* illustrates a technique that you can use to implement this functionality.

When a task is initiated by a menu selection, ACMS initializes any TASK ARGUMENT clause workspaces with their initial CDD contents or zeros. In order for both a menu and another task to call it, the called task should check the selection method and then transfer the contents from the appropriate workspace (see *Example 5.5, "Passing User Workspaces to Menu Tasks"*).

Another method is to write a task that is selected from the menu and that calls the other task. *Example 5.4, "Updating User and Group Workspaces"* illustrates this technique. You write a dummy task, the only function of which is to call the other task from the menu, thereby avoiding the problem.

5.1.3.4. Example of Updating Group and User Workspaces

Example 5.4, "Updating User and Group Workspaces" and *Example 5.5, "Passing User Workspaces to Menu Tasks"* illustrate how to use group and user workspaces with called tasks when the parent, the called task, or both need to update a group or user workspace.

Following is an excerpt from the task group definition used in *Example 5.4, "Updating User and Group Workspaces"* and *Example 5.5, "Passing User Workspaces to Menu Tasks"*:

```
WORKSPACES ARE
     TASK_WSP1,
     TASK_WSP2,
     USER_WSP WITH TYPE USER;



TASKS ARE
     PARENT_TASK: TASK DEFINITION IS PARENT_TASK;
     CALLED_TASK: TASK DEFINITION IS CALLED_TASK;
END TASKS;
```

If a parent task requires that a called task update a user or group workspace, pass the workspace into a workspace in the called task defined as a TASK ARGUMENT WITH ACCESS MODIFY. When the called task completes, ACMS updates the workspace in the parent task. When the parent task completes, ACMS updates the master copy of the user workspace.

Example 5.4. Updating User and Group Workspaces

```
DEFINE TASK PARENT_TASK

USE WORKSPACES
     TASK_WSP1,
     USER_WSP WITH UPDATE LOCK;
     .
     .
     .

PROCESSING
     CALL TASK CALLED_TASK USING
         TASK_WSP1,
         USER_WSP;
     .
```

```
     .
     .

END DEFINITION;


DEFINE TASK CALLED_TASK

USE WORKSPACES
     TASK_WSP1,
     TASK_WSP2,
WORKSPACE IS
     USER_WSP;
TASK ARGUMENTS ARE
     TASK_WSP1 WITH ACCESS MODIFY,
     USER_WSP WITH ACCESS MODIFY;
     .
     .
     .

PROCESSING
     CALL UPDATE_PROCEDURE USING
         TASK_WSP1,
         TASK_WSP2,
         USER_WSP;
     .
     .
     .

END DEFINITION;
```

Note that the called task defines the workspace named USER_WSP as a task-owned, task instance workspace and not as a user workspace. This definition allows the called task to use the same record layout as that of the workspace when it is defined in the group as a user workspace.

If you need to select CALLED_TASK (shown in *Example 5.5, "Passing User Workspaces to Menu Tasks"*) from an ACMS menu as well as to call CALLED_TASK from PARENT_TASK, the technique of passing the user workspace as a TASK ARGUMENT does not work. Because the task is selected from a menu, ACMS initializes the TASK ARGUMENT workspace with its CDD initial contents or with zeros. For both a menu and another task to call it, the called task should check the selection method and then transfer the contents from the appropriate source.

Example 5.5. Passing User Workspaces to Menu Tasks

```
USE WORKSPACES
     TASK_WSP1,
     TASK_WSP2,
     USER_WSP;
WORKSPACE IS
     USER_WSP WITH NAME USER_WSP_COPY;
TASK ARGUMENTS ARE
     TASK_WSP1,
     USER_WSP_COPY;

BLOCK WORK
PROCESSING
     NO PROCESSING;
```

```
     SELECT FIRST TRUE
         ( ACMS$L_CALL_SEQUENCE_NUMBER <> 0 ):
             MOVE USER_WSP_COPY TO USER_WSP;
     END SELECT;



PROCESSING
     CALL update_procedure USING
         TASK_WSP1,
         TASK_WSP2,
         USER_WSP;



END BLOCK WORK;
ACTION IS
     SELECT FIRST TRUE
         ( ACMS$L_CALL_SEQUENCE_NUMBER <> 0 ):
             MOVE USER_WSP TO USER_WSP_COPY;
     END SELECT;
```

In *Example 5.5, "Passing User Workspaces to Menu Tasks"*, when the task is selected from a menu, the task uses the contents of the user workspace. If another task calls the task, the first processing step copies the user workspace data passed by the parent task into the user workspace declared in this task. At the end of the block, the ACTION clause employs a MOVE phrase to update the copy of the user workspace that the parent task passed as a task workspace argument.

If the ACMS$L_CALL_SEQUENCE_NUMBER is not 0, you know that the task was called by another task, and not initiated using a menu selection.

5.1.4. Controlling Called Tasks

This section describes various mechanisms for controlling tasks that are called by other tasks. You can pass control information in a workspace that you define, use the EXIT task clause, or test ACMS system workspace fields.

5.1.4.1. Passing Control Information in User-Defined Workspaces

In some cases, a called task needs to know information about the parent task. For example, a called task may need to record the name of each task that calls it, or it may need to perform different processing steps, depending on the task that called it. Suppose that you want to pass only a few fields from a system workspace to a called task. The most efficient method is to use the MOVE clause to transfer the data from the system workspace to one or more fields in a task workspace supplied to the called task.

Example 5.6, "Passing Data to a Called Task" illustrates how a parent task can pass workspaces to a called task.

Example 5.6. Passing Data to a Called Task

```
WORKSPACES ARE
     DATA_WSP1,
     DATA_WSP2,
     DATA_WSP3;

BLOCK WORK
```

```
PROCESSING
     CALL PROC_1 USING
         DATA_WSP1,
         DATA_WSP2;
     MOVE ACMS$T_TASK_NAME TO PARENT_TASK_NAME,
         "BATCH_JOB_1.COM" TO ACMS$T_SELECTION_STRING;



PROCESSING
     CALL TASK B USING
         DATA_WSP1,
         DATA_WSP2,
         DATA_WSP3;



PROCESSING
     CALL PROC_2 USING
         DATA_WSP2,
         DATA_WSP3;

END BLOCK WORK;



WORKSPACES ARE
     DATA_WSP1,
     DATA_WSP2,
     DATA_WSP3;

TASK ARGUMENTS ARE
     DATA_WSP1 WITH ACCESS READ,
     DATA_WSP2 WITH ACCESS MODIFY,
     DATA_WSP3 WITH ACCESS WRITE;

BLOCK WORK



PROCESSING
     CALL PROC_3 IN SVR1 USING
         DATA_WSP1,
         DATA_WSP2,
         DATA_WSP3;



PROCESSING WITH NO IO
     DCL COMMAND IS "SUBMIT 'P1" IN SVR2;



END BLOCK WORK;
```

In this example, Task B:

1. Accepts one set of data items for ACCESS READ into DATA_WSP1

2. Accepts another set of data items for ACCESS MODIFY into DATA_WSP2

3. Returns a group of data items in the workspace DATA_WSP3, defined with ACCESS WRITE.

The name of the parent task is passed to the called task in a field named PARENT_TASK_NAME in the DATA_WSP1 workspace. Task A copies the name of the current task into that field with the MOVE clause. It uses the DATA_WSP1 workspace, because that workspace is defined in the TASK ARGUMENTS clause for READ access only.

This example also illustrates one method of loading data into the workspace for use by the called task. In this case, Task B uses the ACMS$SELECTION_STRING workspace to submit a batch job using a DCL server. ACMS always passes the contents of the ACMS$SELECTION_STRING system workspace from the parent task to the called task. ACMS parses each element of the ACMS$SELECTION_STRING workspace and assigns the elements to the DCL parameter. The first element is assigned to P1, the second element is assigned to P2, and so on.

The technique of moving one or two fields from a system workspace into a task instance workspace is more efficient than supplying the entire ACMS$TASK_INFORMATION workspace to the called task. Not only do you pass one less workspace to the called task, but the called task does not require as much space in the task instance workspace pool.

5.1.4.2. Ending a Called Task

A task can end by exiting normally or by canceling itself. In either case, you can return one of the default ACMS task status codes or a status code that you define.

To exit from a task normally, use the EXIT TASK clause. The EXIT TASK syntax allows a task to complete with a success status value other than that which produces the default "Task completed normally" message (ACMS$_TASK_COMPLETE). If the EXIT TASK clause returns a status, it must be a success status.

You can use the CANCEL TASK clause to cancel a task from within a task definition. If the CANCEL TASK clause returns a status, it must be a failure status. You can specify a failure status value other than that which produces the default "Cancel results from a step action in the task definition" message (ACMS$_TASK_DEF). See *Example 5.7, "Returning Your Own Exit and Cancel Status Values"*.

Example 5.7. Returning Your Own Exit and Cancel Status Values

```
PROCESSING
   CALL STORE_NEW_CUSTOMER USING
       CUST_RECORD;
   SELECT FIRST TRUE
        (TASK_STATUS = "OK"):
          EXIT TASK RETURNING APPLMSG_CUST_ADDED;
        (TASK_STATUS = "DUPL"):
          CANCEL TASK RETURNING APPLMSG_DUPL_CUST;
   END SELECT;
```

In this example, APPLMSG_CUST_ADDED returns a success status value that you define, and APPLMSG_DUPL_CUST returns a cancel error status that you define.

When a called task exits, ACMS commits any active recovery unit (if there is no recovery-unit command) and returns the final status value to the parent task, as well as the contents of any workspaces

defined as TASK ARGUMENTS for MODIFY or WRITE access. A called task can use this method to return information to the parent task to subsequently control the execution of that task.

When a called task is canceled, ACMS rolls back any active recovery unit (if there is no recovery-unit command). If the canceled task is retaining context in a server and you have defined a cancel routine for the server in the task group, ACMS runs the cancel routine. If the parent task BLOCK STEP has a CANCEL ACTION clause, ACMS executes that command. If a task cancels for any reason, only the cancel reason status code is returned to the parent task. The contents of any workspaces defined as TASK ARGUMENTS for MODIFY or WRITE access are not returned.

5.1.4.3. Controlling Parent Tasks

Once you have called a task, you can control subsequent execution of the parent task by using either the fields in task workspaces modified by the called task or the called task's final completion status. You can easily obtain the called task's final completion status by using the symbolic message code support in ADU.

For example:

```
PROCESSING
   CALL TASK ENTER_ORDER;
   ACTION IS
     SELECT FIRST TRUE OF
        ( ACMS$L_STATUS = ACMS$_CALL_CANCELLED ):
            GOTO STEP SUBMITTER_CANCEL;
        ( ACMS$L_STATUS = ACMS$_OPR_CANCELLED ):
            GOTO STEP OPERATOR_CANCEL;
     END SELECT;
```

In this example, a called task returns a bad status to the parent task. Instead of canceling the task, ACMS evaluates the SELECT FIRST clause in the ACTION step to determine whether the task was canceled by the user or by an operator.

5.1.5. Defining Local Tasks

You can define a task as LOCAL or GLOBAL. GLOBAL tasks can be either selected from a menu or called from another task. LOCAL tasks can only be called from another task.

GLOBAL is the default task type. You can select a GLOBAL task from a menu or with the **SELECT** command in CP, or you can call or chain to a GLOBAL task from another task. However, you can access LOCAL tasks only by calling or chaining to them from another task.

Define a LOCAL task when you want to:

- Extend task security

  You can implement your own task selection security that does not rely on the ACMS task access control list mechanism. Because you cannot select LOCAL tasks from an ACMS menu or user-written agent, you can provide additional security checking in a task that calls the LOCAL task. This method gives the parent task complete control over which users can select which tasks. Because the parent task is the only GLOBAL task in the application, it is, therefore, the only task that a Command Process (CP) or an agent can select.

- Control access to commonly used tasks

Define as LOCAL tasks any tasks that you wish to use as subroutine or library tasks. Users should not be able to select them from a menu or with the **SELECT** command, because you want to tailor them for this purpose.

You cannot override the LOCAL/GLOBAL task attribute by using a task group definition, if you define the task in a separate task definition.

5.1.6. Mixing I/O Methods in Parent and Called Tasks

ACMS allows a called task to use a different I/O method than that of the parent task. For example, a task using DECforms I/O can call tasks that use terminal I/O in DCL servers or ACMS stream I/O as well as other DECforms I/O tasks.

In order for a task using DECforms or terminal I/O to call a task using stream I/O, the user-written agent must associate a stream ID with a submitter ID. Note that the ACMS CP performs this association automatically. For more information on agents, streams, and submitted IDs, see *VSI ACMS for OpenVMS Systems Interface Programming* [https://docs.vmssoftware.com/vsi-acms-sys-interface-prog/].

Although you can associate a stream ID with a submitter ID, you cannot associate a terminal device specification with a stream ID. Therefore, if an agent calls a task defined to use stream I/O, the called task can only call or chain to other stream I/O tasks or tasks that do not perform I/O. If an agent calls a task defined to use stream I/O, that task *cannot* call a task that performs local requests, remote requests, or terminal I/O from a server.

A similar rule also applies to tasks that do not perform I/O and are, therefore, defined WITH NO I/O. Because no I/O information is passed to the task when an agent calls it, a NO I/O task may only call or chain to other NO I/O tasks.

5.1.7. Form and Server Context in Task Calling

You cannot retain server context when calling a task from another task.

If a task that is retaining context in a server process attempts to call another task, ACMS unconditionally cancels the calling task.

TDMS does not support multiple sessions to the same device and limits form context to a request using the same form as the previous request. Therefore, called tasks using TDMS I/O always share form context with the parent task.

5.1.8. Using the NOT CANCELABLE BY TASK SUBMITTER Clause

Depending on the type of task, you might sometimes want to prevent a task from being canceled by the terminal user. To accomplish this, ACMS lets you specify the NOT CANCELABLE BY TASK SUBMITTER clause in your task definition. The NOT CANCELABLE BY TASK SUBMITTER clause tells ACMS to ignore a submitter cancel request generated in situations such as when a terminal user presses **Ctrl/Y**. This clause affects a task only when it is currently executing.

In *Example 5.8, "MAIN_MENU Task"*, if the user presses **Ctrl/Y** while the MAIN_MENU task is running, ACMS ignores the request to cancel the task. However, when the MAIN_MENU task calls the ENTER_ORDER task, **Ctrl/Y** cancels the called task.

Example 5.8. MAIN_MENU Task

```
WORKSPACE IS
     TASK_CTL_WSP;

 NOT CANCELABLE BY TASK SUBMITTER;

BLOCK WORK

     EXCHANGE
         RECEIVE FORM RECORD GET_SELECTION_FORM_REC_LIS
           RECEIVING TASK_CTL_WSP, TASK_CTL_WSP;
         ACTION IS
           CONTROL FIELD TASK_CTL_WSP.TASK_CTL_FIELD
           "QUIT"  :  EXIT TASK;
           END CONTROL FIELD;

     PROCESSING
         SELECT FIRST TRUE
             ( TASK_NUMBER = 1 ): CALL TASK ADD_CUSTOMER;
             ( TASK_NUMBER = 2 ): CALL TASK MODIFY_CUSTOMER;
             ( TASK_NUMBER = 3 ): CALL TASK ENTER_ORDER;
             ( TASK_NUMBER = 4 ): CALL TASK MODIFY_ORDER;
             ( TASK_NUMBER = 5 ): CALL TASK DISPATCH_ORDERS;
         END SELECT;

         ACTION IS
             CONTROL FIELD IS ACMS$T_STATUS_TYPE
                 "T":
                     GOTO PREVIOUS EXCHANGE;
                 NOMATCH:
                     GET ERROR MESSAGE;
                     GOTO PREVIOUS EXCHANGE;
             END CONTROL FIELD;

END BLOCK WORK;
```

The MAIN_MENU task:

1. Prompts the user for a task selection

2. Calls a task based on that selection

If a called task returns a success status, the menu task prompts for another task selection. If a called task returns a failure status, the MAIN_MENU task uses the GET MESSAGE phrase to retrieve the error text associated with the returned status value, so that the task selection form can display the message for the user. The NOT CANCELABLE BY TASK SUBMITTER clause prevents the user from canceling the menu task itself by typing **Ctrl/Y**.

5.1.9. Auditing and Operation

This section describes how to audit and operate task-calling applications.

5.1.9.1. Task Auditing and Security

ACMS writes an audit record to the audit trail log whenever a task calls another task defined with the AUDIT attribute.

ACMS always checks the task access control list (ACL) to determine whether you can execute a task, even when the task is called from another task.

To audit instances of tasks called by other tasks, ACMS appends the call sequence number to the Task ID field. The ACMS$TASK_INFORMATION system workspace describes the call sequence number.

5.1.9.2. Using ACMS Operator SHOW and CANCEL Commands

The ACMS operator command **ACMS/SHOW USER/FULL** displays the following types of task selections:

- Tasks selected by a user from an ACMS menu
- Tasks selected by a user with the **SELECT** command
- Tasks invoked by a user agent

The **ACMS/SHOW USER/FULL** command does not display any tasks called by another task. For example:

```
$ ACMS/SHOW USER TESTU1/FULL
ACMS V5.0       CURRENT USERS                    Time: 19-DEC-2005
 12:40:52.53


User Name:   TESTU1             Submitter ID: GROW::00010025
   Agent PID: 21C0043F           Device:       RTA3:

   Task Name:    VR_CHECKIN_TASK
     Task ID:    GROW::00010025-00000005
     Appl Name:  VR_APPL
     Appl Node:  GROW::
```

To display all the task instances in a sequence of task calls, use the ACMS operator command **ACMS/SHOW TASKS**. For example:

```
$ ACMS/SHOW TASKS
ACMS V5.0       CURRENT TASKS                    Time: 19-DEC-2005
 12:41:27.60
Task Name: VR_CHECKIN_TASK
   State:        Active; Executing PROCESSING step WORK
   Step Label:   GET_RESV
   Task ID:      GROW::00010025-00000005
   Application:  VR_APPL                                User Name: TESTU1
   Submitter ID: GROW::00010025                         Device:    RTA3:
   Called Task:  VR_GETRESV_TASK
     State:      Active; Executing EXCHANGE step WORK
     Step Label: CHANGE_INFO
     Task ID:    GROW::00010025-00000005-00000001
```

ACMS indents called tasks under the header of the parent task. This example illustrates that task VR_CHECKIN_TASK has called task VR_GETRESV_TASK. The report appends the task call sequence number for the called tasks to the end of the task ID.

The ACMS operator command **ACMS/CANCEL USER** cancels:

- The user's session

- Any task that the user selected from a menu
- Any tasks that the task called

The ACMS operator command **ACMS/CANCEL TASK** accepts an optional third hexadecimal number that indicates the task call sequence number of the task instance to be canceled. If the operator omits the third number, ACMS cancels all the task instances in the task call sequence. For example, the following command cancels both of the tasks displayed in the previous example:

```
$ ACMS/CANCEL TASK/IDENTIFIER=GROW::00010025-00000005
```

The following command cancels VR_GETRESV_TASK. You can omit leading zeros from each number in the task ID.

```
$ ACMS/CANCEL TASK/IDENTIFIER=GROW::10025-5-1
```

Chapter 6. Using the Detached Task Feature

This chapter describes how to use detached tasks in ACMS applications. For information about ACMS operator commands, refer to *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/].

6.1. Overview of Detached Tasks

This section provides an overview of detached tasks, describes an application that effectively implements the interactive and deferred processing modes of task execution, and lists the characteristics of detached tasks.

6.1.1. Detached Tasks

ACMS typically does interactive processing of the tasks in your applications, but certain tasks capture data that does not require immediate processing. Detached tasks provide ACMS applications with a mechanism to start a task with an ACMS operator command, while allowing the task to execute in an unattended, noninteractive manner once it has started.

6.1.2. Designing Your Application to Use Detached Tasks

ACMS supports two basic modes of task execution:

- Interactive work flow
- Detached processing

In the interactive work flow mode, ACMS tasks collect requests from a user and process the requests while the user waits for a response. Users interact with the executing task through a terminal or device. A typical task has an exchange step to accept data from a user, followed by a processing step to process the data. The next exchange step returns control to the terminal to complete the task, or to continue to request input from the terminal.

The detached processing mode is similar to batch-style processing, where no user interaction occurs through a terminal or device, from a task executing within the Application Execution Controller (EXC). A task that executes in this mode is called a **detached task**.

To illustrate the two modes of task execution, consider the following example involving a manufacturing and distribution company. The company has its headquarters and manufacturing facility based in one location, and multiple distribution centers located remotely. At its headquarters, the company maintains a master database, using Rdb, that holds information about its entire operation. In addition, smaller Rdb databases, located at each distribution center, hold information specific to that segment of the business.

During the day, changes are made to the distribution center databases. These changes necessitate updates to the master database. As the changes are made to the distribution center database, records are queued to a queue database on the distribution center's system. The records are then forwarded to the central system. To update the distribution center databases, an employee executes a task through a terminal device. This is an example of the interactive work flow mode.

Subsequently, a task running continuously in the background without any human interaction is executed to update the master database at the central system. This task is a detached task, and shows how an application uses the detached processing mode.

6.1.3. Characteristics of Detached Tasks

Detached tasks have the following characteristics:

- When you write a detached task, the task cannot perform any exchange I/O and you must define the task with the NO I/O phrase.
- Before starting a detached task, you must start the application within which the detached task executes. The name of the detached task must exist in the specified application database file.
- For every detached task started, the ACMS$SIGN_IN service signs in a task submitter. By default, ACMS uses the user name of the application (EXC) as the task submitter.
- A detached task can have a string passed to it through the ACMS$SELECTION_STRING system workspace. When a detached task is started, the selection string can optionally be specified.
- A detached task can be retried automatically after a task failure. When a detached task is started, both a retry limit and retry wait timer can be specified. The retry limit parameter specifies the number of times ACMS retries the detached task after a failure. The retry wait timer defines how long ACMS waits before retrying the detached task after a task failure.

6.2. Managing Detached Tasks

The following sections describe:

- Starting a detached task
- Setting the retry limit
- Setting the retry wait timer
- Showing the status for detached tasks
- Stopping a detached task
- Forcing a detached task not to retry
- Enabling broadcast of detached task notification messages

6.2.1. Starting a Detached Task

To start a detached task, use the **ACMS/START TASK** command. When starting a detached task, specify the task name and the application name where the task executes. You can also specify a retry limit, retry wait timer, user name, and a selection string to pass workspace data.

Before starting a detached task, start the application within which the detached task executes. Use the following command to start a detached task:

```
$ ACMS/START TASK task-name application-name
```

Example 6.1, "Audit Message for Starting a Detached Task" shows an audit message for starting a detached task.

Example 6.1. Audit Message for Starting a Detached Task

```
Type     : TASK            Time : 19-DEC-2005 12:41:27.60
Appl     : DIST_APPL
Task     : DEQUEUE_TASK
User     : SMITH
ID       : MYNODE::00020013-00000001-B985D5E0-009576B2
Sub      : MYNODE::00020013-00000000-B985D5E0-009576B2
Text     : Detached task started
```

See *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/] for more information about the **ACMS/START TASK** operator command.

6.2.2. Setting the Retry Limit

The retry limit is the maximum number of times ACMS retries the detached task after a failure. The following command starts a detached task and specifies that the task can be retried a maximum of 10 times after a task failure:

```
$ ACMS/START TASK timer_task dist_appl/RETRY_LIMIT=10
```

If the detached task retry limit is exceeded, then an audit message of type ERROR is written to the audit log. *Example 6.2, "Audit Message for Exceeding the Retry Limit"* shows an audit message that results when the task exceeds the retry limit.

Example 6.2. Audit Message for Exceeding the Retry Limit

```
Type     : ERROR            Time : 19-DEC-2005 12:41:27.60
Appl     : DIST_APPL
Text     : Detached Task DEQUEUE_TASK terminated due to retry limit being
             exceeded
```

6.2.3. Setting the Retry Wait Timer

The retry wait timer indicates the number of seconds ACMS waits before retrying the detached task after a task failure. The default value is 5 seconds. The minimum value is 1 second, and the maximum is 65,535 seconds. If the detached task is started with a retry limit of 0, or if the task completes successfully, the retry wait timer is not used.

6.2.4. Showing the Status of Detached Tasks

The following sections describe two operator commands that allow you to determine if a detached task is active, and whether the task has started and has not completed.

6.2.4.1. Using the ACMS/SHOW TASK Command

The **ACMS/SHOW TASK** command displays whether an active task is a detached task. Use the following command to display the status of an active detached task:

```
$ ACMS/SHOW TASK task-name
```

If a detached task is active, then ACMS displays a message similar to *Example 6.3, "ACMS/SHOW TASK Message for Detached Tasks"*.

Example 6.3. ACMS/SHOW TASK Message for Detached Tasks

```
ACMS V5.0      CURRENT TASKS            Time:19-DEC-2005 12:42:36.02

Task Name:     DEQUEUE_TASK    ***Detached Task***
   State:         Active; Executing PROCESSING step WORK
   Step Label:    $STEP_3
   Task ID:       MYNODE::00020016-00000001
   Application:   DIST_APPL                              User Name:  JONES
   Submitter ID: MYNODE::00020016                        Device: NL:
```

6.2.4.2. Using the ACMS/SHOW APPLICATION/DETACHED_TASKS Command

While the **ACMS/SHOW TASK** command shows only detached tasks that are active, the **ACMS/SHOW APPLICATION/DETACHED_TASKS** command shows all detached tasks that have started and have not completed.

Use the following command to display the status of all detached tasks:

```
$ ACMS/SHOW APPLICATION/DETACHED_TASKS
```

ACMS displays the message shown in *Example 6.4, "ACMS/SHOW APPLICATION/DETACHED_TASKS Message"*.

Example 6.4. ACMS/SHOW APPLICATION/DETACHED_TASKS Message

```
ACMS V5.0        CURRENT APPLICATIONS          Time: 19-DEC-2005 12:43:01.62

Application Name: DIST_APPL                              State:  STARTED

   Task Name:  DEQUEUE_TASK
     Task State:       Active
     Submitter ID:     MYNODE::00020016              Username:    JONES
     Retry Limit:      10                            Retry Count: 2
     Retry Wait Timer: 60

   Task Name:  UPDATE_CENTRAL_TASK
     Task State:       Waiting to Retry
     Submitter ID:     MYNODE::00020017              Username:    SMITH
     Retry Limit:      10                            Retry Count: 0
     Retry Wait Timer: 60
```

See *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/] for more information about the **ACMS/SHOW APPLICATION/DETACHED_TASKS** operator command.

6.2.5. Stopping a Detached Task

Use the **ACMS/CANCEL TASK** or **ACMS/CANCEL USER** operator commands to stop a detached task when you need to cancel and not retry the detached task. Both operator commands cancel a detached task and the associated submitter.

ACMS treats task failures generated as a result of **ACMS/CANCEL TASK** and **ACMS/CANCEL USER** operator commands as special cases and does not retry the task. For example, you can use the **ACMS/CANCEL USER** operator command to stop a task accessing a queue file, so the queue file can

be backed up. Once the queue file is backed up, you can start the detached task again. This is particularly important for continuous operations.

In addition, a detached task is stopped whenever the application or the ACMS system is stopped with the **/CANCEL** qualifier.

Example 6.5, "Audit Messages for a Detached Task Canceled by a System Operator" shows an audit message for a canceled detached task.

Example 6.5. Audit Messages for a Detached Task Canceled by a System Operator

```
Type   : TASK      Time   : 19-DEC-2005 12:43:32.84
Appl   : DIST_APPL
Task   : DEQUEUE_TASK
User   : SMITH
ID     : MYNODE::00020013-00000001-B985D5E0-009576B2
Sub    : MYNODE::00020013-00000001-B985D5E0-009576B2
Text   : Detached task end
Task completion status: Task canceled by system operator
************************************************************
Type   : OTHER     Time   : 19-DEC-2005 12:43:41.42
Text   : Detached task terminated in application DIST_APPL
Detached task DEQUEUE_TASK canceled by system operator
```

6.2.6. Forcing a Detached Task to Not Retry

To force a detached task to not retry, a task can return the value ACMS$_DETTASK_NORETRY as the completion status. The ACMS$_DETTASK_NORETRY status directs the detached task to not retry even if the retry limit has not been exceeded. Use the following task definition syntax to return the completion status ACMS$_DETTASK_NORETRY:

```
CANCEL TASK
     RETURNING ACMS$_DETTASK_NORETRY;
```

6.2.6.1. Task Failures that Cause ACMS Not to Retry a Task

Table 6.1, "Task Failures" lists some of the failures that cause ACMS not to retry a detached task.

Table 6.1. Task Failures

Name	Description
ACMS$_DETTASK_NORETRY	Task explicitly specified as NO RETRY.
ACMS$_SECURITY_CHECK_FAILED	Submitter not authorized.
ACMS$_APPL_NOT_STARTED	Application is stopping.
ACMS$_OPR_CANCELED	Operator canceled task.
ACMS$_SUB_CANCELED	Operator canceled submitter.
ACMS$_INTERNAL	Internal ACMS error.
ACMS$_TASKNOTCOMP	Child task is not composable.
ACMS$_NEED_IOID	Child task is defined with I/O.
ACMS$_NEED_DEVORRR	Child task requires device name or RR server.
ACMS$_NEED_DEVICE	Child task requires device name.

Example 6.6, "Audit Message for a Task Failure that Is Not Retried" shows an audit message for a task failure that is not retried.

Example 6.6. Audit Message for a Task Failure that Is Not Retried

```
Type:   : ERROR     Time   : 18-DEC-2005 12:43:41.42
Text:     Error in application DIST_APPL
Detached task DEQUEUE_TASK terminated due to a task failure that
cannot be retried.
Task is not composable
```

6.2.7. Broadcasting Detached Task Messages

ACMS operator terminals can receive notification when a detached task terminates under the following conditions:

- The retry limit is exceeded.
- A task failure occurs that ACMS cannot retry.
- The detached task is canceled by an operator command.
- An unexpected internal failure occurs.

ACMS operator terminals receive notification when a detached task is waiting to retry, and also the reason why the task is retrying.

To enable an ACMS operator terminal to receive these broadcasts, use one of the following commands:

```
$ ACMS/SET SYSTEM/PROCESS/OPERATOR
```

The **ACMS/SET SYSTEM/PROCESS/OPERATOR** command enables your terminal as an ACMS operator terminal for the duration of your process or until ACMS stops.

```
$ ACMS/SET SYSTEM/TERMINAL=TTE2/OPERATOR
```

The **ACMS/SET SYSTEM/TERMINAL/OPERATOR** command enables terminal TTE2 as an ACMS operator terminal for as long as ACMS is active.

Example 6.7, "Broadcast Message for a Detached Task that Exceeded the Retry Limit" shows a detached task broadcast message for a detached task that exceeded the retry limit.

Example 6.7. Broadcast Message for a Detached Task that Exceeded the Retry Limit

```
%ACMS, 20-DEC-2005 12:43:41.42, Detached task DEQUEUE_TASK terminated due
 to
retry limit being exceeded
```

6.3. Using Group Workspaces in a Detached Task

Workspaces that are declared as group workspaces and are used by detached tasks that execute for long periods of time (for example, several days) have the following limitations:

- If the detached task accesses a group workspace for modify access, then the detached task prevents other tasks from updating the workspace until the detached task is completed.

- If the detached task accesses a group workspace for read access, then the detached task gets the contents of the workspace at the time the detached task starts, and uses the contents until the detached task is completed.

To effectively use group workspaces, a detached task that executes for long periods of time must call a task that executes for a short period of time (short-lived) to update the contents of a group workspace. The detached task must also call a short-lived task to read the contents of a group workspace when the contents are needed.

6.4. Concurrent-Use Licensing with Detached Tasks

For every detached task started, the ACMS$SIGN_IN service is used to sign in a task submitter. If your ACMS system has a concurrent-use license, a set of license units is allocated for each detached task that is started. These license units remain allocated until the task submitter signs out and the detached task stops.

The submitter for a detached task is signed out under the following conditions:

- The task completes successfully.
- The retry limit for the detached task is exceeded.
- The task is canceled as a result of an **ACMS/CANCEL TASK** or an **ACMS/CANCEL USER** operator command.
- The application within which the detached task is executing is stopped.
- The task completes with a task failure that ACMS cannot retry.
- An unexpected internal failure occurs; for example, the retry wait timer could not start.

Chapter 7. Defining Distributed Transactions

The previous chapters show how to write task definitions that perform add, delete, update, and inquiry operations. For the sake of simplicity, the examples assume that only one data source is being accessed. However, there might be times when you need to combine updates to multiple files and/or databases into one operation, or transaction. This chapter describes how to define tasks that use distributed transactions. Specifically, this chapter describes:

- Why you might want to use distributed transactions
- How to include distributed transactions syntax in your task definition
- How to include multiple resource managers in a distributed transaction
- How to include a called task in a distributed transaction
- How distributed transactions affect server context
- How to exclude a processing step from participating in a distributed transaction
- Why transactions can fail

7.1. Why Use Distributed Transactions

Often, a business function involves operations on several databases or files. For example, a simple funds transfer function involves debiting one account and crediting another account. For the function to be successful, both operations must complete successfully. Because it is important to treat both operations as one unit of work, include them in an atomic transaction. An **atomic transaction** is a set of one or more operations where either all the operations take effect or none of them take effect. If any of the operations cannot complete successfully, the transaction is **aborted**. This means that all the operations are **rolled back**, or undone. The transaction is said to be **committed** if all the operations complete successfully and are made permanent.

In the funds transfer example, it is essential that the operations be included in an atomic transaction. Otherwise, the consistency of the database is in jeopardy. For example, if the system fails after the debit operation but before the credit operation, the database is not accurate.

To access information stored in databases and files, you use resource managers. A **resource manager** controls shared access to a set of recoverable resources. The most common type of resource manager is a database system. Rdb, DBMS, and RMS are resource managers. On the OpenVMS operating system, a resource manager can ensure that a set of database operations involving one database is atomic. An atomic set of operations, such as debits and credits, to a database is a **database transaction**. In the funds transfer example, if the account to be debited and the account to be credited reside in the same Rdb database, Rdb can ensure that both operations complete successfully or both operations are undone. If both operations are successful, Rdb commits the database transaction. If one of the operations fails, Rdb rolls back the database transaction.

The funds transfer example represents a very simple business function with one database. However, your application might include multiple databases. The OpenVMS operating system provides a set of transaction services, DECdtm services, that ACMS uses to control a group of database transactions involving multiple databases on one or more nodes. The DECdtm services ensure the atomicity of a set of database transactions included in a distributed transaction. A **distributed transaction** is an

atomic transaction that consists of a set of operations involving multiple resources on one or more nodes. Suppose the two accounts in the funds transfer example reside in separate Rdb databases. A distributed transaction would include two database transactions. If the debit and credit operations complete successfully, the DECdtm services end the distributed transaction by instructing the Rdb resource manager to prepare the database transactions to be committed. If Rdb successfully prepares both database transactions, the DECdtm services instruct Rdb to commit both database transactions. If one of the operations fails, the DECdtm services abort the distributed transaction and instruct Rdb to roll back the database transactions.

The resource managers need not be the same type to participate in a distributed transaction. For example, a distributed transaction could include a procedure that updates an Rdb database and a second procedure that writes to an RMS file.

Because the ACMS queuing system uses RMS, you can treat the queuing system as another type of resource manager by setting queue files for journaling. You can coordinate the removal of queued task elements from the task queue with updates to a database. Likewise, you can coordinate the insertion of queued task elements to a task queue with updates to a database. See *Chapter 9, "Queuing ACMS Tasks"* for details on how to include queuing operations in a distributed transaction.

See *VSI ACMS for OpenVMS Concepts and Design Guidelines* [https://docs.vmssoftware.com/vsi-acms-concepts-and-design-guide/] for more information on designing distributed transactions.

7.2. Including Distributed Transactions Syntax in the Task Definition

ACMS provides syntax that lets you control distributed transactions in the task definition. You can start a distributed transaction on a root block step, nested block step, root processing step, or a processing step within a block by specifying the phrase DISTRIBUTED TRANSACTION in the attributes part of the step. The DISTRIBUTED keyword is optional. A distributed transaction must end in the action part of the same step on which it started. Therefore, a distributed transaction can span multiple processing steps only if you specify TRANSACTION on the block that contains those processing steps.

To explicitly end a distributed transaction, specify either the COMMIT TRANSACTION action clause or the ROLLBACK TRANSACTION action clause. COMMIT TRANSACTION instructs ACMS to call the transaction services to commit the transaction. If the resource managers participating in the transaction can successfully complete their operations, DECdtm instructs each resource manager to commit its operations, making all the changes permanent. However, if any of the resource managers is unable to complete its operations, DECdtm instructs all resource managers to roll back their updates. ROLLBACK TRANSACTION instructs ACMS to call the DECdtm services to abort the transaction; DECdtm then instructs each resource manager to roll back any updates made during the transaction.

Example 7.1, "Distributed Transaction on a Nested Block Step" shows the structure of a task definition that contains a distributed transaction.

Example 7.1. Distributed Transaction on a Nested Block Step

```
BLOCK WORK
    EXCHANGE
        .
        .
    BLOCK WORK WITH DISTRIBUTED TRANSACTION
       PROCESSING
            .
            .
```

```
        PROCESSING
            .
            .
    END BLOCK;
    COMMIT TRANSACTION;

    EXCHANGE
        .
        .
END BLOCK;
```

In this example, the distributed transaction spans the two processing steps within the nested block. Because the distributed transaction starts on the nested block, you must specify the COMMIT TRANSACTION clause in the action part of the same block step. The exchange steps before and after the nested block are not part of the distributed transaction.

If you do not explicitly end a distributed transaction in the action part of the step on which it started, ACMS provides a default of COMMIT TRANSACTION unless the action part of the step contains either the CANCEL TASK or RAISE EXCEPTION sequencing action clause, in which case ACMS provides a default of ROLLBACK TRANSACTION.

You can also start and end distributed transactions in procedures by using the $START_TRANS, $END_TRANS, and $ABORT_TRANS system services. See *VSI ACMS for OpenVMS Systems Interface Programming* [https://docs.vmssoftware.com/vsi-acms-sys-interface-prog/] for information on including these services in agent programs.

7.3. Including Multiple Resource Managers in a Distributed Transaction

The AVERTZ sample car rental application includes a reservation task, VR_RESERVE_TASK, which gathers customer and reservation information from the terminal operator and updates several Rdb databases. The task uses several distributed transactions to ensure the integrity of the databases.

Example 7.2, "Multiple Database Updates in a Distributed Transaction" shows a part of the VR_RESERVE_TASK that includes updates to two Rdb databases in a distributed transaction. Although the resources in this example use the same type of resource manager, Rdb, you can include different types of resource managers in a distributed transaction. For example, a distributed transaction might include updates to a DBMS database and an RMS file.

Example 7.2. Multiple Database Updates in a Distributed Transaction

```
CANCEL_RES:

BLOCK WITH TRANSACTION

     PROCESSING
         CALL PROCEDURE  VR_CANCEL_RS_PROC
         IN      VR_UPDATE_SERVER
         USING   VR_RESERVATIONS_WKSP,
                 VR_CONTROL_WKSP;

     ACTION IS
         GET MESSAGE INTO VR_CONTROL_WKSP.MESSAGEPANEL;
         MOVE "CANCEL   " TO VR_HIST_WKSP.TRANS_TYPE;
```

```
    PROCESSING
        CALL PROCEDURE  VR_WRITE_HIST_RECORD_PROC
        IN      VR_LOG_SERVER
        USING   VR_HIST_WKSP,
                VR_RESERVATIONS_WKSP;

END BLOCK;

    ACTION IS
        COMMIT TRANSACTION;
        MOVE "      " TO VR_CONTROL_WKSP.CTRL_KEY,
             "ACTWT" TO VR_SENDCTRL_WKSP.SENDCTRL_KEY;
```

The CANCEL_RES nested block step appears toward the end of the VR_RESERVE_TASK, and is processed if the terminal user wants to cancel a reservation. The nested block step starts a distributed transaction by specifying WITH TRANSACTION.

The first processing step calls the VR_CANCEL_RS_PROC procedure, which uses the reservation number in the VR_RESERVATIONS_WKSP to locate and delete the particular reservation record in the reservation database. If the procedure completes successfully, the action part of the step moves CANCEL to the TRANS_TYPE field in VR_HIST_WKSP.

The second processing step calls the VR_WRITE_HIST_RECORD_PROC procedure, which records the cancel transaction in the history database.

The COMMIT TRANSACTION clause in the action part of the nested block step instructs ACMS to call DECdtm to make permanent the effects of the two database operations. Because the distributed transaction starts on the nested block step, it must end in the action part of the same step.

7.4. Using Task Sequencing Actions in a Distributed Transaction

When writing some task definitions that use distributed transactions, you must be aware of restrictions that ACMS imposes on the use of certain sequencing action clauses. Specifically, use of distributed transactions affects the EXIT TASK, CANCEL TASK, GOTO TASK, and REPEAT TASK action clauses.

- Because a distributed transaction must end in the action part of the step on which it starts, you cannot specify EXIT TASK on the action part of a step within a distributed transaction. Instead, you can use the EXIT BLOCK action clause, which instructs ACMS to pass control to the action part of the block step that started the distributed transaction.
- If you specify CANCEL TASK in the action part of a step that starts a distributed transaction, ACMS provides a default of ROLLBACK TRANSACTION. You can override this default by specifying COMMIT TRANSACTION on the action part of the step that starts the transaction. If you specify CANCEL TASK on a step within a distributed transaction, ACMS always calls DECdtm to abort the distributed transaction.
- You cannot specify the GOTO TASK clause in the action part of a step within a distributed transaction; however, you can specify GOTO TASK in the action part of the step that starts a distributed transaction. If you specify GOTO TASK on a root block, that task cannot be called by a parent task to participate in a distributed transaction.
- You cannot specify the REPEAT TASK clause in the action part of a step within a distributed transaction; however, you can specify REPEAT TASK in the action part of the step that starts a

distributed transaction. If you specify REPEAT TASK on a root block, that task cannot be called by a parent task to participate in a distributed transaction. An alternative is to specify the WHILE DO block conditional clause at the start of the root block.

- If you specify the REPEAT STEP clause in the action part of a root block, that task cannot be called by a parent task to participate in a distributed transaction. As with REPEAT TASK, an alternative is to specify WHILE DO at the start of the root block.

7.5. Including a Called Task in a Distributed Transaction

Previous sections show how to define a distributed transaction within a single task. You can also define a distributed transaction that starts in a parent task, includes calls to other tasks, and then ends in the parent task. For a called task to be able to participate in a distributed transaction started by a parent task, the called task must conform to the following rules:

- The root block or root processing step must include the TRANSACTION phrase.
- The root block or root processing step cannot include a sequencing action clause other than EXIT TASK, CANCEL TASK, or RAISE EXCEPTION.
- The root block or root processing step cannot include the COMMIT TRANSACTION or ROLLBACK TRANSACTION action clause.
- The root block or root processing step cannot include an exception handler.
- The root block or root processing step cannot include the CANCEL ACTION phrase.

A task that conforms to these rules is said to be a **composable** task. A parent task cannot exclude a called task from participating in an existing distributed transaction.

In the VR_RESERVE_TASK, when customers make a reservation, they have the option of checking out the car immediately. Rather than require the terminal user to exit from the task, return to the AVERTZ menu, and select a different task to check out the car, you can have the VR_RESERVE_TASK call the VR_COMPLETE_CHECKOUT_TASK to handle the checkout. *Example 7.3, "Calling a Task to Participate in a Distributed Transaction"* shows the nested block in VR_RESERVE_TASK that includes a called task within a distributed transaction.

Example 7.3. Calling a Task to Participate in a Distributed Transaction

```
BLOCK WITH TRANSACTION

UPDATE_CUST_INFO:
 !+
 ! If the user wanted to checkout the car and has updated the
 ! driver license info then
 !-

     PROCESSING
         CALL PROCEDURE  VR_STORE_CU_PROC
         IN      VR_CU_UPDATE_SERVER
         USING   VR_CONTROL_WKSP,
                 VR_CUSTOMERS_WKSP,
                 VR_TRANS_WKSP;

     ACTION IS
```

```
         IF (ACMS$T_STATUS_TYPE = "B") THEN
                 GET MESSAGE INTO VR_CONTROL_WKSP.MESSAGEPANEL;
                 RAISE EXCEPTION VR_UPDATE_ERROR;
         END IF ;
!+
! If want to check car out now (=GTCAR) then call
! VR_COMPLETE_CHECKOUT_TASK to do that.
!-

     PROCESSING
         CALL TASK       VR_COMPLETE_CHECKOUT_TASK
         USING           VR_SENDCTRL_WKSP,
                         VR_CONTROL_WKSP,
                         VR_RESERVATIONS_WKSP,
                         VR_TRANS_WKSP,
                         VR_VEHICLES_WKSP;
END BLOCK;

     ACTION IS
                 MOVE "     " TO VR_CONTROL_WKSP.CTRL_KEY,
                      "ACTWT" TO VR_SENDCTRL_WKSP.SENDCTRL_KEY;
                 COMMIT TRANSACTION;
                 GOTO STEP DISP_STAT;
```

The nested block step in VR_RESERVE_TASK starts a distributed transaction with the TRANSACTION phrase. This step uses a distributed transaction because it performs processing work before it calls the VR_COMPLETE_CHECKOUT_TASK. The first processing step calls the VR_STORE_CU_PROC procedure to update the customer's record. If the called task fails, it is important that the effects of the VR_STORE_CU_PROC procedure be rolled back.

The second processing step calls VR_COMPLETE_CHECKOUT_TASK. Because the distributed transaction starts on the nested block step in the parent task, the action part of the same step ends the distributed transaction with the COMMIT TRANSACTION clause.

Example 7.4, "Complete Definition of the VR_COMPLETE_CHECKOUT_TASK" shows the complete definition of the VR_COMPLETE_CHECKOUT_TASK, called by VR_RESERVE_TASK.

Example 7.4. Complete Definition of the VR_COMPLETE_CHECKOUT_TASK

```
REPLACE TASK AVERTZ_CDD_TASK:VR_COMPLETE_CHECKOUT_TASK

USE WORKSPACES VR_CONTROL_WKSP,
                VR_VEHICLES_WKSP,
                VR_RENTAL_CLASSES_WKSP,
                VR_TRANS_WKSP,
                VR_SENDCTRL_WKSP,
                VR_RESERVATIONS_WKSP,
                VR_VE_ARRAY_WKSP,
                VR_HIST_WKSP;

TASK ARGUMENTS ARE VR_SENDCTRL_WKSP     WITH ACCESS READ,
                   VR_CONTROL_WKSP       WITH ACCESS MODIFY,
                   VR_RESERVATIONS_WKSP WITH ACCESS MODIFY,
                   VR_TRANS_WKSP         WITH ACCESS READ,
                   VR_VEHICLES_WKSP      WITH ACCESS READ;

BLOCK WITH TRANSACTION
```

```
            NO I/O
 !
PERFORM:
 !+
 ! Perform the checkout process or cancel the reservation depending
 ! on the user's choice
 !
      PROCESSING
         SELECT FIRST TRUE
                 (VR_CONTROL_WKSP.CTRL_KEY = "OK"):
                         CALL PROCEDURE    VR_COMPLETE_CHECKOUT_PROC
                         IN      VR_UPDATE_SERVER
                         USING   VR_RESERVATIONS_WKSP,
                                 VR_VEHICLES_WKSP;
                 (VR_CONTROL_WKSP.CTRL_KEY = "CANCL"):
                         CALL PROCEDURE  VR_CANCEL_RS_PROC
                         IN      VR_UPDATE_SERVER
                         USING   VR_RESERVATIONS_WKSP,
                                 VR_CONTROL_WKSP;
         END SELECT;

     ACTION IS
         SELECT FIRST TRUE OF
         (VR_CONTROL_WKSP.CTRL_KEY = "OK"):
            GET MESSAGE INTO VR_CONTROL_WKSP.MESSAGEPANEL;
            MOVE "CHECKOUT"  TO VR_HIST_WKSP.TRANS_TYPE,
               VR_VEHICLES_WKSP.VEHICLE_ID TO VR_HIST_WKSP.VEHICLE_ID;
         (VR_CONTROL_WKSP.CTRL_KEY = "CANCL"):
            GET MESSAGE INTO VR_CONTROL_WKSP.MESSAGEPANEL;
            MOVE "CANCEL  "  TO VR_HIST_WKSP.TRANS_TYPE,
               VR_VEHICLES_WKSP.VEHICLE_ID TO VR_HIST_WKSP.VEHICLE_ID;
         NOMATCH:
                 CANCEL TASK;
         END SELECT;

! Write to the history record to record the completion of the
! checkout or the cancellation of the reservation.

     PROCESSING
         CALL PROCEDURE  VR_WRITE_HIST_RECORD_PROC
         IN      VR_LOG_SERVER
         USING   VR_HIST_WKSP,
                 VR_RESERVATIONS_WKSP;


END BLOCK;

!
! This is a composable task called by the RESERVE and the CHECKOUT
! tasks. In the case of the RESERVE task the distributed transaction
! is started in the RESERVE task and therefore committed in the
! RESERVE task.  However, the CHECKOUT task does not start the
! distributed transaction but the COMPLETE_CHECKOUT task is composable
! so the commit is done as a default action by ACMS.
!
END DEFINITION;
```

For the VR_COMPLETE_CHECKOUT_TASK task to be composable, it must include the TRANSACTION phrase at the root block step. At the end of the block step, the task does not commit or roll back the distributed transaction. The transaction must end in the task in which it started, the VR_RESERVE_TASK.

Example 7.4, "Complete Definition of the VR_COMPLETE_CHECKOUT_TASK" shows one called task participating in a distributed transaction. You can include multiple called tasks in a distributed transaction. For example, the VR_COMPLETE_CHECKOUT_TASK can include a call to another task as long as that task is composable.

A parent task that does not start a distributed transaction can call a task that includes a distributed transaction. In this case, the called task ends the distributed transaction.

7.6. How Distributed Transactions Affect Server Context

By default, any server used by processing steps within a distributed transaction is reserved to that distributed transaction until the transaction ends. As a result, ACMS automatically retains server context between steps within a distributed transaction. You cannot specify the RELEASE SERVER CONTEXT action clause on a step that participates in a distributed transaction. When DECdtm ends a distributed transaction, by either committing it or rolling it back, ACMS automatically releases server context. You cannot specify the RETAIN SERVER CONTEXT or NO SERVER CONTEXT ACTION clause in the action part of a step that starts a distributed transaction.

Within a distributed transaction, if multiple processing steps call one or more procedures in the same server, ACMS allocates just one server process for all the processing steps. Therefore, the first procedure called within a distributed transaction must ready the database for all procedures in the transaction. See *VSI ACMS for OpenVMS Writing Server Procedures* [https://docs.vmssoftware.com/vsi-acms-writing-server-proc/] for more information on how to write procedures that ready the database.

A called task that participates in a distributed transaction started by the parent task does not share server context with the parent task. In other words, if the parent and called tasks include processing steps that call procedures in the same server, ACMS allocates a server process for the parent task and a second server process for the called task.

Note

A task can retain context in multiple servers only if each server participates in a distributed transaction. If a task attempts to start a distributed transaction while retaining context in a server, ACMS cancels the task. It is strongly recommended that you do not include exchange steps within a distributed transaction. Including exchange steps within a distributed transaction increases the system resources used by the application, and can adversely affect performance.

7.7. Excluding a Processing Step from a Distributed Transaction

Occasionally, within a block that starts a distributed transaction, you want to include a processing step that accesses a database independently of the distributed transaction. You need to exclude a processing step from a distributed transaction, if the application requires that the effects of the processing step survive even when the transaction is rolled back.

For example, an application requires a security log that records information about users who access or attempt to access sensitive information. In this case, the procedure that updates the security log is not part of the distributed transaction because if the transaction fails you do not want to roll back the security log update.

To exclude a processing step from a distributed transaction, use the NONPARTICIPATING SERVER phrase on the processing step. Because a task can retain context in multiple servers only if each of the servers participates in a distributed transaction, ACMS automatically releases server context at the end of a processing step that specifies NONPARTICIPATING SERVER. You cannot specify the RETAIN SERVER CONTEXT or NO SERVER CONTEXT ACTION clause in the action part of a processing step that specifies NONPARTICIPATING SERVER.

The AVERTZ sample application includes an agent written in C that starts a distributed transaction and calls a task that joins the distributed transaction and queues a task. The called task, VR_FAST_CHECKIN_TASK, uses the NONPARTICIPATING SERVER phrase to exclude two processing steps from the distributed transaction. *Example 7.5, "VR_FAST_CHECKIN_TASK with Nonparticipating Processing Steps"* shows the VR_FAST_CHECKIN_TASK definition.

Example 7.5. VR_FAST_CHECKIN_TASK with Nonparticipating Processing Steps

```
BLOCK WORK WITH DISTRIBUTED TRANSACTION
                NO I/O

!
! Retrieve the reservation record, using the reservation number/id
! entered by the customer and passed by the vr_agent agent.
!

         PROCESSING WITH NONPARTICIPATING SERVER
                 CALL PROCEDURE  VR_FIND_RES_PROC
                   IN    VR_READ_SERVER
                 USING   VR_FAST_CHECKIN_WKSP,
                         VR_RESERVATIONS_WKSP;

         ACTION IS
             IF (ACMS$T_STATUS_TYPE = "G")
             THEN
                MOVE VR_FAST_CHECKIN_WKSP.ACTUAL_RETURN_DATE
                 TO VR_VEHICLE_RENTAL_HISTORY_WKSP.ACTUAL_RETURN_DATE,
                    VR_FAST_CHECKIN_WKSP.RETURN_ODOMETER_READING
                 TO
                   VR_VEHICLE_RENTAL_HISTORY_WKSP.RETURN_ODOMETER_READING;
             ELSE
                 CANCEL TASK RETURNING  ACMS$L_STATUS;
             END IF;


!
! RETRIEVE THE VEHICLE AND VEHICLE_RENTAL_HISTORY RECORDS
!

         PROCESSING WITH NONPARTICIPATING SERVER
                 CALL PROCEDURE  VR_FIND_VE_VRH_PROC
                   IN    VR_READ_SERVER
                 USING   VR_RESERVATIONS_WKSP,
                         VR_VEHICLES_WKSP,
                         VR_VEHICLE_RENTAL_HISTORY_WKSP,
```

```
                        VR_RENTAL_CLASSES_WKSP,
                        VR_TRANS_WKSP;

         ACTION IS
             IF (ACMS$T_STATUS_TYPE = "B") THEN
                 CANCEL TASK RETURNING  ACMS$L_STATUS;
             END IF;
!
! QUEUE THE TASK TO BE RUN LATER
!

         PROCESSING
                 CALL PROCEDURE    VR_ENQ_FAST_CHECKIN
                   IN    VR_QUEUE_SERVER
                 USING   VR_FAST_CHECKIN_WKSP;

         ACTION IS
             IF  (ACMS$T_STATUS_TYPE = "G")
             THEN
                 MOVE  "FASTCHIN"            TO VR_HIST_WKSP.TRANS_TYPE,
                           VR_VEHICLES_WKSP.VEHICLE_ID
                                             TO VR_HIST_WKSP.VEHICLE_ID;
             ELSE
                 CANCEL TASK RETURNING  ACMS$L_STATUS;
             END IF;

!
! WRITE A RECORD OF A SUCCESSFUL CHECK IN TO THE HISTORY DATABASE
!
        PROCESSING
             CALL PROCEDURE    VR_WRITE_HIST_RECORD_PROC
             IN      VR_LOG_SERVER
             USING   VR_HIST_WKSP,
                     VR_RESERVATIONS_WKSP;

END BLOCK;
```

Because the VR_FAST_CHECKIN_TASK is joining a distributed transaction started by the agent, the root block step must use the DISTRIBUTED TRANSACTION phrase.

The first processing step in VR_FAST_CHECKIN_TASK uses the reservation ID, obtained from the customer by the agent, to retrieve the reservation record. The second processing step retrieves the car history record. Because the first two processing steps perform read-only operations, neither step needs to participate in the distributed transaction. Therefore, both steps use the NONPARTICIPATING SERVER phrase.

ACMS automatically releases context at the end of each of the first two processing steps, thereby freeing up the server processes for other tasks.

The third processing step calls the VR_ENQ_FAST_CHECKIN procedure to queue the VR_COMP_FAST_CHKIN_TASK.

The final processing step writes a record of the transaction into the database. If the procedure completes successfully, the task ends, and the agent calls DECdtm to commit the distributed transaction.

7.8. Handling Deadlocks and Transaction Failures

If your application contains tasks that use distributed transactions, deadlock problems are possible. One type of deadlock involves multiple tasks attempting to access the same server process. For example, suppose that two tasks each use two servers. Each server has one active server process. The first task accesses the first server, and the second task accesses the second server. If both tasks then attempt to access the other server, they become deadlocked waiting to use the server process being used by the other task.

If your application and databases are distributed across multiple systems that are not part of a single OpenVMS Cluster system, deadlocks can occur when multiple tasks attempt to access the same database records. The OpenVMS Lock Manager is able to detect deadlocks only within a single system or an OpenVMS Cluster system.

By specifying the TRANSACTION TIMEOUT subclause in the application definition, you can instruct ACMS to call the transaction services to abort a transaction if it has not completed within a certain number of seconds. See *Chapter 11, "Defining Applications"* for an example of an application definition that includes TRANSACTION TIMEOUT.

In addition to deadlocks, there are a number of reasons why distributed transactions might fail, such as:

- Communication with a remote node participating in the distributed transaction fails.
- A resource manager participating in the distributed transaction detects a data integrity constraint failure.
- A called task completes with a transaction error or is canceled.

ACMS lets you test for and recover from distributed transaction failures. For information on how to write task definitions that handle distributed transaction failures, see *Chapter 8, "Handling Task Execution Errors"*.

Chapter 8. Handling Task Execution Errors

Chapter 2, "Defining Tasks" shows how to handle some common task errors, such as when a requested database record cannot be found, by using conditional clauses to test workspace field values. However, this method cannot detect and handle certain events that prevent tasks from completing successfully. To handle all types of task execution errors, ACMS provides an exception handler action component of the task definition language. This chapter describes:

- The three classes of exceptions
- The RAISE EXCEPTION clause
- The EXCEPTION HANDLER ACTION clause
- How to recover from a transaction exception, a DECforms time-out exception, and a task-call-task exception
- How ACMS performs exception handling at run time

8.1. Why Use Exception Handling

Use of exception handling is optional. By default, if ACMS encounters an error that prevents it from executing a task, ACMS cancels the task. In some situations, you might want to handle the error and continue task execution. For example, if the task includes one or more distributed transactions, you might want to include exception handler action syntax to test for, and recover from, distributed transaction failures. If you do not specify an action to take when a distributed transaction fails, ACMS cancels the task. Instead, depending upon the reason the transaction failed, you might want to retry the distributed transaction or transfer control to another part of the task definition.

You might also want to use exception handling to test for, and recover from, DECforms timeouts. For example, if a terminal user does not fill in a DECforms field within the timeout period specified in the exchange step, DECforms returns an error. Rather than let ACMS cancel the task, you can specify an exception handler action to test for the timeout error and repeat the exchange step, reminding the terminal user to fill in the panel.

You can also use exception handling to enable a parent task to continue executing when a called task fails. In short, exception handling gives you flexibility in designing tasks that can detect a variety of execution errors and respond in a predictable manner.

8.2. What is an Exception

An **exception** is an event or an error condition that interrupts the normal flow of an executing task. Each exception has a unique OpenVMS longword exception code that indicates the failure condition. ACMS raises, or generates, an exception based on errors or events occurring in the following sources:

- System software such as ACMS, OpenVMS, Rdb, and DECforms
- A task definition action clause
- A user-written program running in a server or in an agent

- An operator command or request such as **ACMS/CANCEL TASK**

An **exception handler** is an optional part of an exchange, processing, or block step that lets you control task execution when an exception is raised in the work or action part of the step. Note that the work part of a block step includes the work, action, and exception handler parts of all exchange, processing, and nested block steps in that block step. After an exception has been raised, you can test for its presence with a conditional clause, and specify the action to take in the exception handler part of the step. The class of exception determines where in the task definition you can specify an exception handler, and whether you can handle the exception at all. The three classes of exceptions are:

- Step
- Transaction
- Nonrecoverable

8.2.1. Step Exceptions

A **step exception** is an error that you can handle in the exception handler part of the step on which the exception occurs or in an exception handler on an outer block step. *Table 8.1, "Step Exceptions"* shows the complete list of step exceptions.

Table 8.1. Step Exceptions

Source of Exception	Description of Exception
RAISE EXCEPTION action clause	You can specify an exception code with RAISE EXCEPTION; if you do not, ACMS provides a default exception code based on where in the task definition you specify RAISE EXCEPTION. *Section 8.3, "Using the RAISE EXCEPTION Clause"* describes how to use the RAISE EXCEPTION action clause.
ACMS$RAISE_STEP_EXCEPTION service	You can use this service to raise a step exception when the procedure detects an error that it cannot handle. You can specify actions to recover from the error in an exception handler part of the task definition. See *VSI ACMS for OpenVMS Writing Server Procedures* [https://docs.vmssoftware.com/vsi-acms-writing-server-proc/] for more information on using this service in step procedures.
VSI DECforms	A common reason for ACMS to raise a step exception is that the terminal user did not fill in a VSI DECforms field within the specified timeout period. ACMS also raises a step exception when VSI DECforms encounters data conversion problems while moving data from a task workspace field to a form.
TDMS	A common reason for ACMS to raise a step exception is that TDMS encounters data conversion problems while moving data from a task workspace field to a form.
User-written request procedure (URP)	ACMS raises a step exception if a user-written request procedure returns an error status.
Stream I/O operation	ACMS raises a step exception if an agent completes a stream I/O request by passing an error status to the ACMS$REPLY_TO_STREAM_IO service.

Source of Exception	Description of Exception
Called task	If a task, called by a parent task that does not start a distributed transaction, completes with any type of exception, ACMS raises a step exception in the parent task. If a called task that participates in a distributed transaction started by the parent task completes with a step exception, ACMS raises a step exception in the parent task. See *Section 8.5.2, "Recovering from a Task-Call-Task Exception"* for details on handling exceptions that occur in called tasks.

8.2.2. Transaction Exceptions

A *transaction exception* is an error that causes a distributed transaction to fail. *Table 8.2, "Transaction Exceptions"* shows the complete list of transaction exceptions.

Table 8.2. Transaction Exceptions

Source of Exception	Description of Exception
Transaction timeout error	In the application definition, you can use the TRANSACTION TIMEOUT phrase to specify a time limit within which the distributed transaction must complete. If the transaction does not end within the specified number of seconds, ACMS raises a transaction exception with the ACMS$_TRANSTIMEDOUT exception code. If the transaction started in the current task, ACMS calls the DECdtm services to abort the transaction. If the task was called to participate in a transaction started by a parent task, ACMS calls DECdtm to abort the transaction when ACMS resumes executing the parent task. If the task was called to participate in a transaction started by an agent, the agent must abort the transaction when the task ends and the ACMS$CALL or ACMS$WAIT_FOR_CALL_END service completes.
ACMS$RAISE_TRANS_EXCEPTION service	You can use this service to raise a transaction exception when the procedure detects an error that it cannot handle. You can specify actions to recover from the error in an exception handler part of the task definition. See *VSI ACMS for OpenVMS Writing Server Procedures* [https://docs.vmssoftware.com/vsi-acms-writing-server-proc/] for more information on using this service in step procedures.
Transaction is aborted	A transaction exception can be raised when a distributed transaction is aborted while ACMS is executing the task. For example, if a network link to a remote database server process that is participating in a distributed transaction fails, ACMS raises a transaction exception.
SYS$ABORT_TRANS[W] system service	If a user-written agent calls the SYS$ABORT_TRANS[W] service while ACMS executes a composed task, ACMS raises a transaction exception.

Source of Exception	Description of Exception
Called task	If a called task that participates in a distributed transaction started by the parent task completes with a transaction exception or a nonrecoverable exception, ACMS raises a transaction exception in the parent task.

Note

A step procedure that participates in a distributed transaction started by a task or an agent must not call the SYS$ABORT_TRANS[W] service, because the results are unpredictable.

8.2.3. Nonrecoverable Exceptions

A *nonrecoverable exception* is an error from which the task cannot recover. When a nonrecoverable exception is raised, ACMS cancels the task. *Table 8.3, "Nonrecoverable Exceptions"* shows the list of nonrecoverable exceptions.

Table 8.3. Nonrecoverable Exceptions

Source of Exception	Description of Exception
CANCEL TASK action clause	As with the RAISE EXCEPTION clause, the CANCEL TASK clause lets you raise an exception. However, while RAISE EXCEPTION lets you recover from the exception and continue executing the task, CANCEL TASK raises a nonrecoverable exception that always cancels the task.
ACMS$RAISE_NONREC_EXCEPTION system service	You can call ACMS$RAISE_NONREC_EXCEPTION from a step procedure to raise a nonrecoverable exception when the procedure detects an error from which it cannot recover. *VSI ACMS for OpenVMS Writing Server Procedures* [https://docs.vmssoftware.com/vsi-acms-writing-server-proc/] has more information on how to use the ACMS$RAISE_NONREC_EXCEPTION system service.
Server procedure	ACMS raises a nonrecoverable exception when a step procedure encounters a fatal error, such as an access violation.
Server process	ACMS raises a nonrecoverable exception if a server process dies while a task is executing a step procedure or a DCL command in the server process, or is retaining context in the server process.
Submitter-requested cancellation	If a terminal user presses **Ctrl/Y** to cancel a task, and the task definition does not include the NOT CANCELABLE phrase, ACMS raises a nonrecoverable exception.
Operator-requested cancellation	If a system operator invokes the **`ACMS/CANCEL TASK`** command to cancel a task, ACMS raises a nonrecoverable exception.

Source of Exception	Description of Exception
RMS file or database recovery unit	A file or database recovery unit might fail to start because a realm or relation is temporarily locked. A constraint check failure might prevent a recovery unit from committing successfully.
Unknown distributed transaction failure	If a distributed transaction fails for an unknown reason, ACMS raises a nonrecoverable exception.
Any other task cancellation	Any event or error condition that causes ACMS to unconditionally cancel the task causes ACMS to raise a nonrecoverable exception.

8.3. Using the RAISE EXCEPTION Clause

To explicitly raise a step exception in the task definition, use the RAISE EXCEPTION clause in the action part of the step where the error is detected. You do not have to specify an exception code with RAISE EXCEPTION; ACMS provides a default exception code of ACMS$_EXCPTN_TASKACTN when you raise an exception in the action part of a step. If you use the RAISE EXCEPTION clause in an exception handler, and you do not specify an exception code, ACMS raises a step exception using the exception code associated with the current exception. If you do specify an exception code, it must be a failure status. Otherwise, ACMS cancels the task. *Example 8.1, "RAISE EXCEPTION Clause in a Processing Step"* shows a processing step that uses the RAISE EXCEPTION clause.

Example 8.1. RAISE EXCEPTION Clause in a Processing Step

```
PROCESSING WORK
    CALL ENTER_ORDER IN DIST_CTR_DATABASE_UPDATE_SERVER
       USING ORDER_ENTRY_RECORD, RESTOCK_RECORD, STATUS_RECORD;
    ACTION IS
       IF (DATA_IS_VALID <> "Y")
       THEN
         RAISE EXCEPTION APPL_INVALID_DATA
       END IF;
```

In *Example 8.1, "RAISE EXCEPTION Clause in a Processing Step"*, the processing step calls a procedure to add a record to a database. The action part of the step tests the DATA_IS_VALID workspace field to see if the procedure returned a Y, indicating that the data to be stored in the database is valid. If the procedure returns a value other than Y, the processing step raises a step exception. APPL_INVALID_DATA is a user-defined exception code.

When an exception is raised, ACMS searches for an exception handler in the current step. If the current step does not contain an exception handler, ACMS searches the outer block or blocks until it finds an exception handler. If ACMS finds an exception handler, ACMS moves the exception code into the ACMS$L_STATUS field of the ACMS$PROCESSING_STATUS system workspace. You can then use a conditional clause in the exception handler action part of the step to test for the exception code and to direct the task flow. If ACMS does not find an exception handler, it cancels the task.

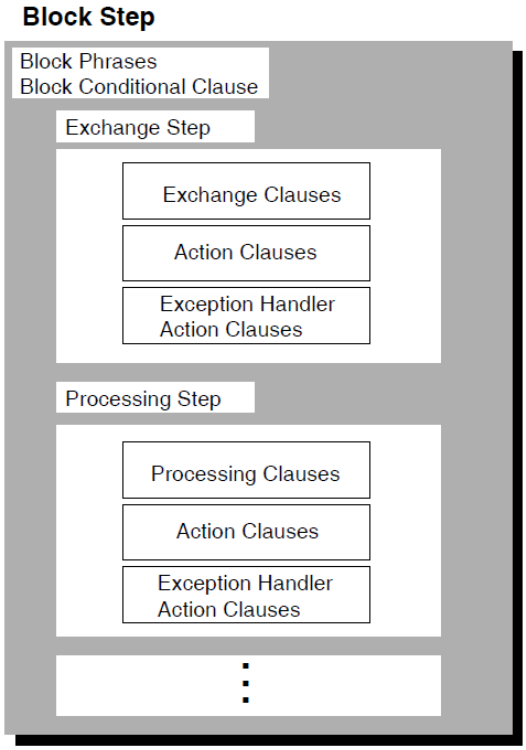
Section 8.6, "How ACMS Performs Exception Handling" contains a detailed description of how ACMS performs exception handling at run time.

8.4. Using Exception Handler Actions

Exception handler actions make up the third part of a step, following the work and action parts. The exception handler action part is optional. If you use it, it must appear after the work and action parts of

the step. The action part of a step is optional. Use the EXCEPTION HANDLER clause to introduce the exception handler component. *Figure 8.1, "Block Step Structure"* shows how exception handlers fit into the structure of a block step.

Figure 8.1. Block Step Structure



You use the same action clauses in the exception handler component as you use in the action part of a step with the exception of transaction actions and recovery actions. You cannot use a transaction action clause (COMMIT TRANSACTION or ROLLBACK TRANSACTION) or a recovery unit action clause in an exception handler. Because there is no logical default sequencing action for an exception handler, ACMS requires that you include a sequencing action clause. Regardless of the order in which the exception handler action clauses appear in the task definition, ACMS executes the clauses in the following order:

1. Workspace manipulation
2. Server process context action
3. Task sequencing

Example 8.2, "Performing Exception Handling in a Task" shows how to include exception handling in a task definition.

Example 8.2. Performing Exception Handling in a Task

```
BLOCK WORK WITH FORM I/O
     EXCHANGE
        RECEIVE FORM RECORD ORDER_ENTRY_RECORD IN ORDER_ENTRY_FORM
          RECEIVING ORDER_ENTRY_WKSP;
WRITE_ORDER:
     BLOCK WORK WITH DISTRIBUTED TRANSACTION
```

```
         PROCESSING
            CALL ENTER_ORDER IN DIST_CTR_DATABASE_UPDATE_SERVER
            USING ORDER_ENTRY_WKSP, RESTOCK_WKSP, STATUS_WKSP;
         ACTION IS
            IF (DATA_IS_VALID <> "Y")
            THEN
               RAISE EXCEPTION APPL_INVALID_DATA
            END IF;
         PROCESSING
            IF (ORDERED_AMOUNT > IN_STOCK_AMOUNT)
            THEN
               CALL QUEUE_REPLENISH_INVENTORY_TASK_PROC IN QUEUE_SERVER
               USING RESTOCK_WKSP;
            END IF;
     END BLOCK;
     ACTION IS
        COMMIT TRANSACTION;
        EXIT TASK;
     EXCEPTION HANDLER ACTION IS
        IF (ACMS$L_STATUS = APPL_INVALID_DATA)
        THEN
           GOTO STEP REENTER_DATA;
        END IF;
REENTER_DATA:
     EXCHANGE
        TRANSCEIVE FORM RECORD STATUS_RECORD, ORDER_ENTRY_RECORD
                   IN ORDER_ENTRY_FORM
                   SENDING STATUS_WKSP
                   RECEIVING ORDER_ENTRY_WKSP;
     ACTION IS
        IF (RETRY_TRANSACTION = "Y")
        THEN
           GOTO STEP WRITE_ORDER;
        ELSE
           EXIT TASK;
        END IF;
END BLOCK;
```

In *Example 8.2, "Performing Exception Handling in a Task"*, if the first processing step raises the APPL_INVALID_DATA exception, ACMS first looks for an exception handler in the same step. Because the processing step does not include an exception handler, ACMS searches the end of the nested block step where it finds an EXCEPTION HANDLER ACTION clause. If the nested block step does not include an exception handler, ACMS then looks at the end of the root block.

If an exception is raised other than the ones you test for in the exception handler part of a step, ACMS provides a default action of RAISE EXCEPTION. For example, in the above task definition, if an exception other than APPL_INVALID_DATA is raised, ACMS raises another exception, which you can handle with an exception handler on the root block.

8.5. Examples of Exception Handling

Because ACMS exception handling is an all-purpose task execution error handling tool, you can use it in task definitions to detect and recover from a variety of exceptions. The following sections show how to use exception handling to recover from a DECforms time-out exception, a task-call-task exception, and a transaction exception.

8.5.1. Recovering from a VSI DECforms Time-Out Exception

DECforms lets you specify a time limit within which the terminal user must fill in a DECforms field. You may want to provide a way to recover from a DECforms time-out error so that ACMS does not cancel the task. *Example 8.3, "Recovering from a VSI DECforms Time-Out Exception"* shows how to test for and recover from a DECforms time-out exception.

Example 8.3. Recovering from a VSI DECforms Time-Out Exception

```
DISPLAY_RESV_NO:
     EXCHANGE
!
! DISPLAY RESERVATION # AND PROMPT TO SEE IF CUSTOMER WANTS TO CHECK
! CAR OUT NOW.
!
         TRANSCEIVE RECORD       RES_INFO,
                                 RES_CHECKOUT_DETAILS
         SENDING                 VR_CONTROL_WKSP,
                                 VR_RESERVATIONS_WKSP
         RECEIVING               VR_CONTROL_WKSP,
                                 VR_RESERVATIONS_WKSP,
                                 VR_CUSTOMERS_WKSP,
                                 VR_TRANS_WKSP
         WITH TIMEOUT            30
              SEND CONTROL       VR_SENDCTRL_WKSP;

     ACTION IS
         CONTROL FIELD VR_CONTROL_WKSP.CTRL_KEY
         "QUIT" : EXIT TASK;
         "CANCL": MOVE "ACTWT" TO VR_SENDCTRL_WKSP.SENDCTRL_KEY,
                        0      TO VR_CONTROL_WKSP.RETRY_COUNT,
                       "Y"     TO VR_CONTROL_WKSP.INCREMENT_RETRY_COUNT;
                  GOTO STEP CANCEL_RES;
         "REPET": GOTO STEP DISPLAY_RESV_NO;
         "GTCAR": GOTO STEP FIND_CAR;
         "     ": MOVE "ACTWT" TO VR_SENDCTRL_WKSP.SENDCTRL_KEY;
                  GET MESSAGE NUMBER VR_RESVCOMP INTO
                    VR_CONTROL_WKSP.MESSAGEPANEL;
                  GOTO STEP DISP_STAT;
         END CONTROL;

! IF THE FORM TIMES OUT, REMIND THE USER ONCE TO MAKE A SELCTION -
! THEN CANCEL THE TASK.

     EXCEPTION HANDLER
       SELECT FIRST TRUE OF
       (ACMS$L_STATUS = FORMS$_TIMEOUT AND VR_CONTROL_WKSP.RETRY_COUNT =
 0):
                 MOVE 1         TO VR_CONTROL_WKSP.RETRY_COUNT,
                      "RETRY"   TO VR_CONTROL_WKSP.CTRL_KEY;
                 GET MESSAGE NUMBER VR_INACTIVE INTO
                                 VR_CONTROL_WKSP.MESSAGEPANEL;
                 GOTO STEP DISPLAY_RESV_NO;
       (ACMS$L_STATUS = FORMS$_TIMEOUT AND VR_CONTROL_WKSP.RETRY_COUNT >
 0):
```

```
                GET MESSAGE INTO VR_CONTROL_WKSP.MESSAGEPANEL;
                MOVE "ACTWT" TO  VR_SENDCTRL_WKSP.SENDCTRL_KEY,
                     "     " TO VR_CONTROL_WKSP.CTRL_KEY;
                GOTO STEP DISP_STAT;
        END SELECT;
```

In *Example 8.3, "Recovering from a VSI DECforms Time-Out Exception"*, the exchange step specifies that the terminal user must fill in each DECforms field within 30 seconds. The exception handler part of the exchange step tests the ACMS$L_STATUS field of the ACMS$PROCESSING_STATUS workspace for the presence of the FORMS$_TIMEOUT exception code. If ACMS raises the time-out exception, the exception handler action repeats the exchange step to give the terminal user a second chance to complete the DECforms field. If the time-out exception occurs a second time, ACMS passes control to the DISP_STAT exchange step, which displays an error message, then ends the task.

Note

If your application is distributed across a front-end node that performs terminal I/O and a back-end node that performs computation and database operations, ACMS Version 3.2 or higher must be installed on both nodes, if the task definition is to recover from exceptions raised by exchange steps.

8.5.2. Recovering from a Task-Call-Task Exception

If an exception occurs in a called task, you might want to continue executing the parent task rather than letting ACMS cancel it. You can specify an exception handler action in the processing step that called the task or on an outer block step in the parent task. Note that the CONTINUE ON BAD STATUS phrase is a declining feature that has been superseded by the exception handler mechanism. Do not use it in new task definitions. *Example 8.4, "Recovering from an Exception Raised in a Called Task"* shows part of a parent task that includes an exception handler action to recover from possible exceptions raised in the called task.

Example 8.4. Recovering from an Exception Raised in a Called Task

```
COMPLETE_PROC:

! CALL THE VR_COMPLETE_CHECKOUT_TASK TO COMPLETE THE CHECKOUT
! TRANSACTION.
!
!
     PROCESSING
         CALL TASK       VR_COMPLETE_CHECKOUT_TASK
         USING           VR_SENDCTRL_WKSP,
                         VR_CONTROL_WKSP,
                         VR_RESERVATIONS_WKSP,
                         VR_TRANS_WKSP,
                         VR_VEHICLES_WKSP;
!
! THE DISTRIBUTED TRANSACTION IS STARTED IN THE VR_COMPLETE_CHECKOUT_TASK
! BUT IS NOT EXPLICITLY COMMITTED SINCE IT IS A COMPOSABLE TASK. THE
! COMMIT WILL BE PERFORMED BY ACMS (DEFAULT ACTION).
!

     ACTION IS
         MOVE "     " TO VR_CONTROL_WKSP.CTRL_KEY,
              "ACTWT" TO VR_SENDCTRL_WKSP.SENDCTRL_KEY;
!
```

```
! RETRY IF A TIME OUT ERROR OCCURRED BEFORE CANCELING TASK.
!
     EXCEPTION HANDLER
          IF ( (ACMS$L_STATUS = ACMS$_TRANSTIMEDOUT AND
                  VR_CONTROL_WKSP.RETRY_COUNT = 0) )
          THEN
               MOVE 1       TO VR_CONTROL_WKSP.RETRY_COUNT;
               REPEAT STEP;
          ELSE
               GET MESSAGE INTO VR_CONTROL_WKSP.MESSAGEPANEL;
               MOVE "     " TO VR_CONTROL_WKSP.CTRL_KEY,
                    "ACTWT" TO VR_SENDCTRL_WKSP.SENDCTRL_KEY;
               GOTO STEP DISP_STAT;
          END IF;

!
! DISPLAY STATUS TO THE USER.
!
 DISP_STAT:

     EXCHANGE
          SEND RECORD             VR_CONTROL_WKSP
          SENDING                 VR_CONTROL_WKSP
          WITH SEND CONTROL       VR_SENDCTRL_WKSP;
```

In *Example 8.4, "Recovering from an Exception Raised in a Called Task"*, the COMPLETE_PROC processing step calls the VR_COMPLETE_CHECKOUT_TASK to compute the customer's bill and perform final checkout work. The VR_COMPLETE_CHECKOUT_TASK starts a distributed transaction. Because VR_COMPLETE_CHECKOUT_TASK is a composable task, it cannot explicitly commit or roll back the distributed transaction. Because a distributed transaction must end in the action part of the step where it starts, ACMS provides the default transaction action.

The exception handler part of the parent task uses an IF THEN ELSE conditional clause to test the contents of the ACMS$L_STATUS field in the ACMS$PROCESSING_STATUS workspace. If the called task raises the ACMS$_TRANSTIMEDOUT exception code, ACMS retries the called task once.

If a called task completes without raising an exception, ACMS returns the contents of any modify-access and write-access task argument workspaces to the parent task. ACMS then resumes executing the task from the action part of the processing step that called the task.

If a called task completes with an exception, ACMS does not return the contents of task argument workspaces to the parent task. If a task, called by a parent task that does not start a distributed transaction, completes with any type of exception, ACMS raises a step exception in the parent task. If a called task that participates in a distributed transaction started by the parent task completes with a step exception, ACMS raises a step exception in the parent task. If a called task that participates in a distributed transaction completes with a transaction or nonrecoverable exception, ACMS raises a transaction exception in the parent task.

8.5.3. Recovering from a Transaction Exception

If your task includes a distributed transaction, there are several exceptions you may want to test for and recover from by including an exception handler in the task definition. A communication link between nodes participating in the distributed transaction could be broken, or a resource manager that is part of the distributed transaction might detect an error and be unable to prepare its part of the distributed

transaction. *Example 8.5, "Recovering from a Transaction Exception"* shows how to test for and recover from a transaction exception.

Example 8.5. Recovering from a Transaction Exception

```
BLOCK WITH TRANSACTION

!+
! ONLY UPDATE CUSTOMER RECORD IF SHADOW RECORD INDICATES
! CHANGE HAS BEEN MADE ON PREVIOUS EXCHANGE.

     IF ( (VR_CUSTOMERS_SHADOW_WKSP.REC_STATUS = "1") OR
          (VR_TRANS_SHADOW_WKSP.REC_STATUS = "1")     OR
          (VR_SENDCTRL_WKSP.SENDCTRL_KEY = "TRAGN") ) THEN

         PROCESSING

             CALL PROCEDURE VR_STORE_CU_PROC
             IN      VR_CU_UPDATE_SERVER
             USING   VR_CONTROL_WKSP,
                     VR_CUSTOMERS_WKSP,
                     VR_TRANS_WKSP;

         ACTION IS
             IF (ACMS$T_STATUS_TYPE = "B") THEN
                 GET MESSAGE INTO VR_CONTROL_WKSP.MESSAGEPANEL;
                 MOVE "TRAGN" TO VR_SENDCTRL_WKSP.SENDCTRL_KEY;
                 EXIT BLOCK;
             ELSE
                 MOVE "     " TO VR_CONTROL_WKSP.CTRL_KEY,
                      "     " TO VR_SENDCTRL_WKSP.SENDCTRL_KEY;
             END IF;
     END IF;

 STORE_RESV:
!+
! CREATE RESERVATION NUMBER AND WRITE RESERVATION RECORD TO DB.
!-
     PROCESSING
         CALL PROCEDURE  VR_WRITE_RS_PROC
         IN      VR_UPDATE_SERVER
         USING   VR_CONTROL_WKSP,
                 VR_RESERVATIONS_WKSP,
                 VR_SITES_WKSP,
                 VR_RENTAL_CLASSES_WKSP,
                 VR_CUSTOMERS_WKSP;

     ACTION
         MOVE "RESERVE " TO VR_HIST_WKSP.TRANS_TYPE;

!+
! WRITE A RECORD TO THE HISTORY FILE TO LOG THE COMPLETED TRANSACTION
! - HISTORY RECORDS ARE RDB RECORDS (COULD BE ANY TYPE OF FILE SYSTEM
! OR DATABASE). THIS IS PART OF THE DISTRIBUTED TRANSACTION.
!-

     PROCESSING
```

```
         CALL PROCEDURE  VR_WRITE_HIST_RECORD_PROC
         IN      VR_LOG_SERVER
         USING   VR_HIST_WKSP,
                 VR_RESERVATIONS_WKSP;

!
! END OF DISTRIBUTED TRANSACTION
!
END BLOCK;

     ACTION  IS
         IF (ACMS$T_STATUS_TYPE = "G")

         THEN
                 COMMIT TRANSACTION;
                 MOVE "     " TO VR_SENDCTRL_WKSP.SENDCTRL_KEY,
                      "Y"     TO VR_CONTROL_WKSP.INCREMENT_RETRY_COUNT,
                       0      TO VR_CONTROL_WKSP.RETRY_COUNT;
         ELSE
                 ROLLBACK TRANSACTION;
                 GOTO STEP DISPLAY_CUST_INFO;
         END IF;
!
! EXCEPTION HANDLER FOR ABOVE BLOCK
!
! RETRY THE DISTRIBUTED TRANSACTION 5 TIMES (ONLY IF A
! ACMS$_TRANSTIMEDOUT ERROR OCCURRED) BEFORE CANCELING TASK.
! THE RETRY_COUNT IS INCREMENTED IN EITHER VR_STORE_CU_PROC
! OR VR_WRITE_RS_PROC.
!
     EXCEPTION HANDLER
          IF (ACMS$L_STATUS = ACMS$_TRANSTIMEDOUT AND
              VR_CONTROL_WKSP.RETRY_COUNT < 5)
          THEN
             REPEAT STEP;
          ELSE
             GET MESSAGE INTO VR_CONTROL_WKSP.MESSAGEPANEL;
             MOVE "ACTWT" TO  VR_SENDCTRL_WKSP.SENDCTRL_KEY,
                  "     " TO VR_CONTROL_WKSP.CTRL_KEY;
             GOTO STEP DISP_STAT;
          END IF;
```

In this example, a nested block starts a distributed transaction and includes three processing steps. The first processing step checks to see if the user has changed any customer information. If one of the shadow workspaces indicates that information has changed, the processing step calls the VR_STORE_CU_PROC procedure to update the database.

The STORE_RESV step adds a reservation record to the reservation database. The third processing step updates a transaction log in another database.

If the three processing steps complete successfully, the action part of the nested block step commits the distributed transaction. If an error occurs that causes the distributed transaction to fail before completing, ACMS raises a transaction exception. The exception handler part of the nested block checks for the ACMS$_TRANSTIMEDOUT transaction exception code to see if the distributed transaction did not complete within the time limit specified in the application definition with the TRANSACTION TIMEOUT phrase.

If the ACMS$L_STATUS field contains the ACMS$_TRANSTIMEDOUT exception code, ACMS repeats the block step up to five times. If ACMS$L_STATUS contains any other exception code, or if the distributed transaction has already timed out five times, ACMS passes control to the DISP_STAT exchange step, which displays an error message, and ends the task.

8.6. How ACMS Performs Exception Handling

To be able to design tasks that make efficient use of exception handling, you need to know how ACMS executes tasks that use exception handling. ACMS takes different actions depending upon the type of exception and where in the task definition the exception was raised; however, for step exceptions and transaction exceptions, ACMS performs the following steps:

1. Interrupts task execution. If the exception is raised while ACMS is executing an exchange step, ACMS cancels the exchange I/O. If the exception is raised while ACMS is executing a processing step, ACMS cancels a call to a server procedure or DCL command.

2. Searches for an exception handler. When ACMS finds an exception handler, ACMS stores the exception code in the ACMS$PROCESSING_STATUS system workspace, and evaluates the conditional clause, if present. If no exception handler has been specified, ACMS cancels the task.

3. Executes the exception handler action clauses. Regardless of the order in which the action clauses appear in the definition, ACMS processes them in the following order:

    a. Workspace manipulation

    b. Server context action

    c. Task sequencing

The following sections describe how ACMS executes each type of exception, and how ACMS cancels a task.

8.6.1. Executing a Step Exception Outside of a Distributed Transaction

If ACMS detects a step exception outside the bounds of a distributed transaction, ACMS first looks for an exception handler on the same step that raised the exception. If the current step does not include an exception handler, ACMS checks block steps out to the root block step for an exception handler. If ACMS does not find an exception handler, it cancels the task. If ACMS finds an exception handler, it stores the exception code in ACMS$PROCESSING_STATUS and performs the exception handler action clauses. If the exception handler does not specify how to handle the particular exception raised, ACMS searches out to the root block for another exception handler. If ACMS cannot find an exception handler that specifies how to handle the particular exception raised, ACMS raises a nonrecoverable exception and cancels the task. For example, if the exception handler specifies a recovery action only for the FORMS$_TIMEOUT exception code, and a different exception is raised, ACMS cancels the task.

8.6.2. Executing a Step Exception Within a Distributed Transaction

If ACMS detects a step exception within a distributed transaction, ACMS searches for an exception handler on the current step. If the current step does not include an exception handler, ACMS searches

any outer block steps within the distributed transaction. If ACMS finds an exception handler, ACMS performs the exception handler action clauses.

If ACMS does not find an exception handler, or if the exception is raised in a processing step that starts a distributed transaction, ACMS checks to see whether the distributed transaction was started by the current task, a parent task, or an agent. If the current task started the distributed transaction, ACMS aborts the distributed transaction and raises a transaction exception with the same exception code. If the current task is a called task that participates in a distributed transaction started by the parent task or an agent, ACMS cancels the current task and raises a transaction exception in the parent task, or returns the exception code to the agent.

8.6.3. Executing a Transaction Exception

As with step exceptions, when ACMS detects a transaction exception, it interrupts the task execution. If the task was not executing a COMMIT TRANSACTION clause when the transaction exception was raised, ACMS calls the server cancel procedure, if defined, for each server in which the task maintains context. If the task was executing COMMIT TRANSACTION, ACMS does not call any server cancel procedures. Before searching for an exception handler, ACMS releases context held in server processes.

ACMS then searches for an exception handler, starting on the step that started the distributed transaction and searching out to the root block.

8.6.4. Executing Nonrecoverable Exceptions

When ACMS detects a nonrecoverable exception, it interrupts the task execution and begins to cancel the task. ACMS performs the following steps to cancel a task:

1. Handles an active distributed transaction. If a distributed transaction is active, and the task is not a called task that participates in a distributed transaction started by the parent task or agent, ACMS aborts the distributed transaction. If the task is a called task that joins a distributed transaction, the parent task or agent that started the distributed transaction must abort it.

2. Calls the server cancel procedure for each server in which the task maintains context.

3. Writes a task cancellation record to the ACMS audit trail log.

4. Executes the task's CANCEL ACTION processing work, if defined.

5. If task is a called task, returns an exception to the parent task or agent.

8.7. How Exceptions Affect Server Cancel Procedures

When you define the characteristics of a server in the task group definition, you can specify a server cancel procedure. Server cancel procedures perform cleanup work when a task cancels. For example, a server cancel procedure might close a channel that was opened for terminal I/O, release resources used by a server procedure such as an OpenVMS lock, or roll back a database recovery unit. However, calling server cancel procedures after an exception has been raised can adversely affect the performance of your application, because ACMS must make an additional call to each server process.

If an exception causes the server process to be run down, calling a server cancel procedure increases the time it takes to shut down and restart the server process. If the server process does not have to be run

down, calling a server cancel procedure causes a delay in allocating the server process to another task instance. Therefore, wherever possible, design tasks and step procedures that do not call server cancel procedures when an exception has been raised. The following sections describe how ACMS uses the exception type to determine when to call server cancel procedures.

8.7.1. Step Exceptions and Server Cancel Procedures

If a step exception is raised, and the task definition does not include an exception handler to recover from the exception, ACMS raises a transaction exception or a nonrecoverable exception. If a distributed transaction was active when the step exception was raised, ACMS raises a transaction exception; otherwise, ACMS raises a nonrecoverable exception. ACMS then calls any server cancel procedures and cancels the task.

ACMS does not call server cancel procedures when a step exception is raised and the task definition includes an exception handler to recover from the exception.

8.7.2. Nonrecoverable Exceptions Raised by Action Clauses

If a nonrecoverable exception is raised while a task maintains context in one or more server processes, ACMS calls server cancel procedures. If the action part of the step in which the nonrecoverable exception is raised releases server context, ACMS does not call server cancel procedures. The action part of the processing step in the following example explicitly releases server context.

```
PROCESSING
    CALL UPDATE_ORDER_IN ORDER_SERVER USING ORDER_RECORD;
 ACTION IS
    IF (ACMS$L_STATUS <> 1)
    THEN
       RELEASE SERVER CONTEXT;
       CANCEL TASK RETURNING ACMS$L_STATUS;
    END IF;
```

Because ACMS executes server context actions before sequencing actions, ACMS releases server context before canceling the task. Therefore, ACMS does not call server cancel procedures.

The processing step in the previous example did not participate in a distributed transaction. Within a distributed transaction, ACMS requires that server context be retained between steps. Therefore, if a processing step within a distributed transaction cancels the task, ACMS calls server cancel procedures. *Example 8.6, "Canceling a Task without Calling Server Cancel Procedures"* shows how to cancel a task within a distributed transaction without calling server cancel procedures.

Example 8.6. Canceling a Task without Calling Server Cancel Procedures

```
BLOCK WORK WITH DISTRIBUTED TRANSACTION
    PROCESSING
       CALL UPDATE_ORDER IN ORDER_SERVER
            USING ORDER_RECORD;
    ACTION IS
       IF (ACMS$L_STATUS <> 1)
       THEN
          EXIT BLOCK;
       END IF;
    PROCESSING
```

```
       CALL WRITE_LOG_RECORD IN LOG_SERVER
            USING ORDER_RECORD;

END BLOCK;
ACTION IS
    IF (ACMS$L_STATUS = 1)
    THEN
       COMMIT TRANSACTION;
    ELSE
       ROLLBACK TRANSACTION;
       CANCEL TASK RETURNING ACMS$L_STATUS;
    END IF;
```

Example 8.6, "Canceling a Task without Calling Server Cancel Procedures" starts a distributed transaction on the block step. If the UPDATE_ORDER procedure returns a status value other than 1, ACMS passes control to the action part of the block step, rolls back the distributed transaction, and cancels the task. When ACMS processes the ROLLBACK TRANSACTION clause, the distributed transaction ends and ACMS releases server context. As a result, ACMS does not call server cancel procedures when it cancels the task.

8.7.3. Other Nonrecoverable Exceptions and Transaction Exceptions

When ACMS executes a COMMIT TRANSACTION clause, if a transaction exception is raised because the distributed transaction fails to prepare, ACMS does not call server cancel procedures.

In all other cases, if a transaction exception or a nonrecoverable exception is raised while the task maintains context in one or more server processes, ACMS calls server cancel procedures.

Chapter 9. Queuing ACMS Tasks

ACMS typically does interactive processing of the tasks in your application, but certain tasks have requirements that you can meet through the use of task queues. These requirements include:

- Data capture and deferred processing of data

  For example, an application has hundreds of time cards that must be processed in a very short time during a shift change. In this type of application, processing each data item immediately can have adverse effects on the performance of the system, so it is useful to capture the data and store it for future processing. This type of processing is also known as desynchronized processing, because the data capture and the data processing are not synchronized.

- High application availability

  In a distributed environment, if the back-end machine fails, the front-end machine or machines can continue processing by submitting tasks to queues.

- Transaction persistence

  If a system failure interrupts the processing of a queued task, the queuing facilities continue to dequeue and submit that task until it succeeds.

For these and other such requirements, you can use the ACMS queuing facility to capture and initiate the tasks in your ACMS application.

9.1. Understanding the ACMS Queuing Facility

In an ACMS application, exchange steps of a task gather data while processing steps of that task process the data.

When you use ACMS queuing, you may include in a processing step a call to an ACMS programming service to enqueue a deferred task on a **task queue** for later processing. This queued task includes task arguments, together with a task name, and an application name, to comprise a **queued task element** that is stored on the task queue. At a later time, ACMS executes a **queued task** (a task submitted by the ACMS queuing facility). ACMS processes the queued task and then deletes the queued task element from the task queue. In other words, a queued task element is stored in a task queue for later processing.

Queued tasks must have certain characteristics (discussed in *Section 9.6.1, "Characteristics of Queued Tasks That are Processed by the QTI"*) in order for them to be successfully executed by the ACMS queuing facility. For example, queued tasks cannot perform exchange steps to collect input, because all the input needed by the queued task must be passed to the task as task arguments. Aside from these characteristics, there is no difference between a queued task and any other ACMS task.

Note

ACMS task queues are distinct from OpenVMS batch queues. Many of the concepts are similar, but ACMS and OpenVMS queues are completely independent entities that are managed and used by independent interfaces.

You use the following components of the ACMS queuing facility to implement, manage, and execute your queuing applications:

- ACMS Queue Manager Utility (ACMSQUEMGR)

Provides commands that you use to create and manage task queues and queued task elements.

- ACMS Queued Task Services

  Provides the ACMS$QUEUE_TASK and ACMS$DEQUEUE_TASK services that you use to queue and dequeue task elements and their task arguments to and from task queues. You can call these services either from a step procedure or from a standalone, user-written program.

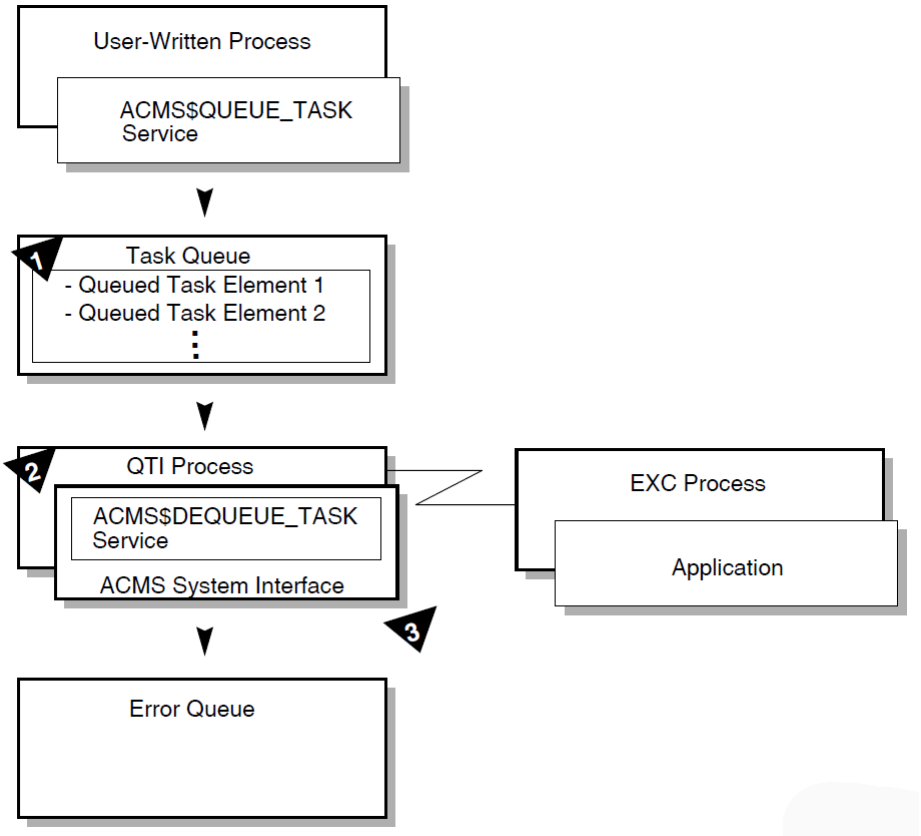
- ACMS Queued Task Initiator (QTI)

  Consists of a run-time component that automatically dequeues task elements from task queues and invokes each queued task in the appropriate application. You use the ACMS operator commands to start, stop, and show the QTI, and to start, stop, and show the queues the QTI processes.

  Conceptually, you can think of the QTI as a user of the ACMS$DEQUEUE_TASK service. The QTI calls the ACMS$DEQUEUE_TASK service to read the queued task elements and then uses the ACMS Systems Interface (SI) services to invoke queued tasks in ACMS applications. Hence, the QTI is an ACMS-supplied agent program.

Figure 9.1, "Queuing, Dequeuing, and Processing Tasks" shows how to use the ACMS$QUEUE_TASK service to queue tasks, and how to use the QTI to dequeue and initiate tasks in the queue.

Figure 9.1. Queuing, Dequeuing, and Processing Tasks



In *Figure 9.1, "Queuing, Dequeuing, and Processing Tasks"*:

1. A high-level language, user-written program calls the ACMS$QUEUE_TASK service to store a queued task element in a task queue. Any data needed by the queued task is passed to the ACMS$QUEUE_TASK service and is stored with the queued task element

2. The QTI process dequeues the queued task element and initiates the queued task in the specified application. The queued task element remains on the task queue as the queued task executes, but it is unavailable for dequeuing. When the queued task completes, the QTI process deletes the queued task element from the queue.

3. If the queued task fails, the QTI automatically retries the task until it determines the task cannot succeed. Then the QTI places the task on the error queue associated with the task queue. If no error queue exists, the QTI removes the task from the queue.

Queues can be shared within a cluster. Therefore, you can have user-written procedures, ACMS applications, and QTI processes that are spread across several nodes in the same cluster, and can queue and dequeue tasks to the same task queue.

The application can be on the same node or on the same cluster as the QTI, or anywhere on the network (using DECnet). The QTI locates the application using an application specification that uses the same semantics (logical names, search lists, and failover) that exist for invoking interactive tasks.

9.2. Using ACMS Queuing with Distributed Transactions

In writing applications that use queue services, you must be concerned with ensuring the integrity of task queues and databases in the event of abnormal system failures (such as a machine crash or a disk failure). For example, if the QTI calls a task and the task completes, but the application node crashes before the QTI is notified that the task completed, the QTI retries the task. Prior to ACMS Version 3.2, the QTI did not know whether the task successfully updated a database. Therefore, when the QTI retries the task, the database might be updated again.

If it is critical that each queued task element update the database exactly once, include queue operations in an atomic transaction. In an atomic transaction, each operation must complete successfully for the transaction to be successful. If one operation fails, the effects of the other operations are undone. An atomic transaction that includes operations involving multiple resources, such as a task queue and a database, is a distributed transaction.

The OpenVMS operating system provides a set of transaction services, DECdtm services, that ACMS uses to control distributed transactions. ACMS lets you use distributed transactions to coordinate the removal of queued task elements from a task queue with updates to a database or file. Likewise, you can coordinate the insertion of queued task elements to a task queue with database updates in a distributed transaction. If all operations in the distributed transaction complete successfully, ACMS calls the transaction services to commit, or make permanent, the effects of the operations. If any of the operations in the distributed transaction fail, ACMS calls the transaction services to roll back, or undo, the effects of the operations. In this way, ACMS ensures that updates to databases or files by the queued task are performed exactly once.

Note

If your application is distributed across OpenVMS Cluster systems, each system must be running ACMS Version 3.2 or higher; otherwise, tasks participating in distributed transactions might have unpredictable results.

To include a queued task in a distributed transaction, you must mark the task queue file for recovery-unit journaling by using the **`SET FILE/RU_JOURNAL`** DCL command. Any task queue file not marked for recovery-unit journaling cannot participate in a distributed transaction. In addition, for a queued task to be able to join an existing distributed transaction, the queued task must conform to the following rules:

- The root block or root processing step must include the TRANSACTION phrase.
- The root block or root processing step cannot include a sequencing action clause other than EXIT TASK, CANCEL TASK, or RAISE EXCEPTION.
- The root block or root processing step cannot include the COMMIT TRANSACTION or ROLLBACK TRANSACTION action clause.
- The root block or root processing step cannot include an exception handler.
- The root block or root processing step cannot include the CANCEL ACTION phrase.

A task that conforms to these rules is said to be a **composable** task.

When ACMS calls DECdtm to start a distributed transaction, DECdtm returns a unique transaction identifier (TID). The ACMS$QUEUE_TASK and ACMS$DEQUEUE_TASK services use the default TID.

At run time, the QTI process starts a distributed transaction for task queues marked for recovery-unit journaling, dequeues the queued task element from the task queue, and initiates the queued task in the specified application. If the task completes successfully, the QTI process deletes the queued task element from the task queue and commits the transaction.

If the queued task fails, the QTI process aborts the distributed transaction which rolls back any updates the queued task made to a database or file, and restores the task queue to the state it was in prior to the distributed transaction. The QTI process then starts another transaction and, depending on the reason for the task failure, performs one of the following actions:

- Retries the queued task element
- Moves the queued task element to an error queue
- Deletes the queued task element from the task queue if there is an error queue

See *Section 9.6.4, "How the QTI Handles Errors"* for details on how the QTI handles errors.

9.3. Steps in Using ACMS Queuing

The following is a list of the management and programming steps you must take to use queuing:

1. Create a task queue using the ACMSQUEMGR Utility. You must create a queue before the ACMS Queued Task services or the QTI run-time component can access that queue.
2. Create an error queue, if needed, using the ACMSQUEMGR Utility.
3. Enable RMS journaling using the DCL **`SET FILE/RU_JOURNAL`** command for task queue files, if the queued tasks are to participate in a distributed transaction.
4. Set ACMSGEN parameters for the QTI process.
5. Place queued task elements onto the task queue by using the ACMS$QUEUE_TASK service. The service can be called from a step procedure or from a standalone program. If the service is called from a step procedure, and you want to include the queue operations in a distributed transaction, include the TRANSACTION phrase in the task definition. If the service is part of a standalone program, use the $START_TRANS system service to include the queue operations in a distributed transaction.
6. Start the QTI process by using the **`ACMS/START QTI`** operator command.

7. Start one or more task queues and specify the error queue to be associated with the task queues by using the **ACMS/START QUEUE** operator command.

8. Process the error task queues, if any, in whatever manner is appropriate for your application. For example, if elements on the error queue could be corrected by a terminal user, then define a task to:

   - Dequeue an element from the error task queue using the ACMS$DEQUEUE_TASK service called from a step procedure
   - Display the data fields to the terminal user and accept corrections using an exchange step
   - Queue the corrected element back to the task queue using the ACMS$QUEUE_TASK service from a step procedure

9. Periodically perform routine maintenance on the task queue files.

The remainder of this chapter contains information about using many of these steps. *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/] describes how to use the ACMSQUEMGR Utility and ACMS operator commands affecting queues. *ACMS$QUEUE_TASK* and *ACMS$DEQUEUE_TASK* contain reference information on the ACMS$QUEUE_TASK and ACMS$DEQUEUE_TASK services.

Considerations for providing queue security are described in *Section 9.4, "Defining Queue Security"*.

9.4. Defining Queue Security

When you use the **CREATE QUEUE** command to create a queue, the ACMSQUEMGR Utility sets the owner of the queue file to [1,4] and sets the protection of the file to O:RWED,S:RWED,G,W. Therefore, to access the queue file, a user must have a system UIC or must have the SYSPRV privilege.

Access to a task is distinct from access to a task queue. Task security is defined by an access control list (ACL) in the application definition. For information on the user name under which queued tasks can run and, therefore, the user name used to determine access to a task, see *Section 9.5.1, "Queuing Tasks Using the* ACMS$QUEUE_TASK *Service"*.

The queue services used from within application programs to access queues have the following access to the queue file:

- The ACMS$QUEUE_TASK service that you use to queue tasks enables the SYSPRV privilege (only for the purpose of accessing the queue file) so it can always access the queue file. Therefore, any user can store a queued task element on any queue. The ACMS$QUEUE_TASK service always enables SYSPRV in order to gain write access to the queue file. SYSPRV is disabled after the ACMS$QUEUE_TASK call completes.
- The ACMS$DEQUEUE_TASK service that you use to dequeue tasks does not enable the SYSPRV privilege and cannot access the queue file automatically. Therefore, any process that uses the ACMS$DEQUEUE_TASK service must have the SYSPRV privilege or a system UIC; only privileged users can dequeue tasks.

The QTI process has the SYSPRV privilege and is, therefore, able to access the queue file to dequeue tasks.

You can use OpenVMS security interfaces to override the default queue security. For example, you can use the DCL **SET FILE** command to change the owner of the queue file or the protection on the queue file. You can also define an access control list for the queue file.

9.5. Using the ACMS Queue Services to Queue and Dequeue Tasks

There are two ACMS queue services:

- ACMS$QUEUE_TASK

  Stores the queued task element in a task queue. You call the ACMS$QUEUE_TASK service from a standalone program or a step procedure written in any OpenVMS-supported language.

- ACMS$DEQUEUE_TASK

  Dequeues, or optionally reads, a queued task element from a queue and returns information about the queued task element.

Any process, including a procedure server in an ACMS application, can call these services. You cannot call these services from a program run in a DCL server. A step procedure or standalone program can call these services from user mode at AST or non-AST level. These services are not AST reentrant. A service that is being called at non-AST level cannot be interrupted by an AST-level call to the service.

VSI ACMS for OpenVMS Writing Server Procedures [https://docs.vmssoftware.com/vsi-acms-writing-server-proc/] shows the calling sequence of the ACMS$QUEUE_TASK service and the ACMS$DEQUEUE_TASK service.

Section 9.7, "Processing Error Queues" and *Section 9.10, "Queuing Example"* contain examples of procedures that use these services.

9.5.1. Queuing Tasks Using the ACMS$QUEUE_TASK Service

Once you have created a task queue using the ACMS Queue Manager (ACMSQUEMGR) Utility, you can queue tasks using the ACMS$QUEUE_TASK service. You place tasks onto the queue from either a step procedure or a standalone program that calls the ACMS$QUEUE_TASK service. Before calling ACMS$QUEUE_TASK, the task or standalone program gathers the information necessary to run the task. Before queuing the task, you might want to perform some of the processing, as well. If you want the queued task to be part of a distributed transaction, you must declare the start of the distributed transaction, either in the task definition by using the TRANSACTION phrase or in the standalone program by using the $START_TRANS service.

The ACMS$QUEUE_TASK service passes several parameters including the queue name (which cannot contain trailing spaces), the task name, the name of the application in which the task is to run, as well as the data to be passed in the workspaces. You can optionally include parameters that assign a priority to the queued task element, place the queued task element in a hold state (make it unavailable for dequeuing), supply a user name under which you want the task to run, and request that the ACMS$QUEUE_TASK service return a unique element ID of the queued task element. The ACMS$QUEUE_TASK service queues the task element and its parameters onto the specified queue.

Associated with each queued task element is an **enqueuer user name**. Provided that the QTI runs under a user name that has the ACMS agent privilege in the User Definition Utility (UDU) user authorization file, the QTI submits tasks under the enqueuer user name. Access to the task and access to the resources used by the task are granted or denied based on the access rights of this user name. By default, the enqueuer user name is the user name of the process that called the ACMS$QUEUE_TASK service. A

user name other than the process user name can be passed to ACMS$QUEUE_TASK as the enqueuer user name if the process has the OpenVMS CMKRNL privilege.

9.5.2. Dequeuing Task Elements Using the ACMS$DEQUEUE_TASK Service

As with the ACMS$QUEUE_TASK service, any process, including a procedure server process, can call the ACMS$DEQUEUE_TASK service. To have the dequeue operation coordinated with other resource manager operations, such as database updates, you must have declared the start of a distributed transaction, either in the task definition by using the TRANSACTION phrase or in the procedure by using the $START_TRANS system service. The most common use of the ACMS$DEQUEUE_TASK service is for taking queued task elements off an error queue, although it is possible to use this service to remove task elements from a task queue and perform your own processing rather than using the QTI. See *Section 9.6, "Using the QTI to Dequeue Tasks"* for more information on using the QTI.

The ACMS$DEQUEUE_TASK service:

- Dequeues, or optionally reads, a queued task element from a queue and returns information about the queued task
- Does not dequeue queued task elements that are on hold (except by element ID)
- Deletes a queue element immediately upon removing it from the queue (unless the ACMS$DEQUEUE_TASK service is called with the READ_ONLY flag)

You can dequeue tasks by highest priority (first-in-first-out within priority), by element ID, or sequentially. See *ACMS$DEQUEUE_TASK* for the list of parameters that you can include on the ACMS $DEQUEUE_TASK service. See *Section 9.7, "Processing Error Queues"* for an example of a procedure that uses the ACMS$DEQUEUE_TASK service to process error queues.

9.6. Using the QTI to Dequeue Tasks

The ACMS QTI (Queued Task Initiator) is a run-time component that concurrently dequeues task elements from one or more queues and invokes the specified task in an ACMS application. The QTI is an ACMS-supplied Systems Interface (SI) agent program that automatically dequeues queued task elements that were queued by the ACMS$QUEUE_TASK service. You use the ACMS operator commands to start and stop the QTI, and to control the queues that are processed by the QTI.

You can start only one QTI process for each node on a cluster, but each QTI process can access multiple task queues anywhere on the same cluster. Multiple QTI processes can access the same task queues and error queues.

The QTI initiates processing of a queue by assigning an execution thread to the queue. An execution thread is a run-time entity which loops to do the processing described in the following list. The QTI is multithreaded, that is, it can execute many tasks (task threads) simultaneously. You can specify up to 255 task threads for a queue when you use the **ACMS/START QUEUE** command or the **ACMS/SET QUEUE/TASK_THREADS** command.

Specifying more execution threads for a queue increases the rate at which queued task elements are processed by performing the processing in parallel; the throughput for a queue increases with the number of execution threads. By assigning different numbers of execution threads to different queues, you can prioritize or weight several queues relative to one another. Normally, relatively small numbers (1 to 5) of task threads are sufficient.

Each execution thread of the QTI processes a queued task element in the following order:

1. If the task queue file is marked for recovery unit journaling, the QTI starts a distributed transaction.
2. Reads and locks the queued task element.
3. If the QTI has the agent privilege and has not already signed in the enqueuer user name, the QTI executes the systems interface ACMS$SIGN_IN service to sign in the submitter. If the QTI does not have the agent privilege, the submitter user name is the user name of the QTI process. You assign the agent privilege through the ACMS User Definition Utility (UDU).
4. Performs the systems interface ACMS$GET_PROCEDURE_INFO, ACMS$START_CALL, and ACMS$WAIT_FOR_CALL_END services to process the task in an ACMS application.
5. If the task completes successfully, the QTI deletes the queued task element from the queue and ends the distributed transaction. If the task fails, the QTI either places the element on hold so it can retry it later, places the task on an error queue (if one was specified), or deletes the element from the task queue (if it cannot be retried and an error queue was not specified).
6. When the queue is empty, the execution thread suspends processing until another queued task element is queued.

See *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/] for more information about the ACMS operator commands you use to manage the QTI.

9.6.1. Characteristics of Queued Tasks That are Processed by the QTI

When you write a task that will be invoked by the QTI, you must write the task so that it abides by the following characteristics:

- The task cannot perform any exchange I/O. You must explicitly use the block phrase WITH NO I/O in the task definition. Because the task has no I/O, the ACMS$TASK_SUBMITTER_DEVICE field of the ACMS$TASK_INFORMATION workspace contains spaces at task execution time.
- The task must be composable if it is to participate in a distributed transaction. *Section 9.2, "Using ACMS Queuing with Distributed Transactions"* describes how to make a task composable. The task queue file must be marked for recovery-unit journaling.
- You must define any input data needed by the task as task workspace arguments. Pass these workspaces to the task through the ACMS$QUEUE_TASK service. You must specify read access for the task workspace arguments being passed to the task.
- The task cannot assume any context in user or group workspaces as a result of previous tasks having executed. Tasks that are queued in a particular order might not be processed in the same order that they were submitted to the queue; therefore, a task invoked by the QTI should not assume context from tasks that are expected to run earlier. The following are some reasons why the order of the invocation of tasks might be different from the order that the queued tasks were queued:
  - Some queued task elements were placed on hold while others were not.
  - A task queue was suspended while another was not.
  - A queued task failed (for example, workspace pool exhausted) but before being retried by the QTI, other queued tasks ran successfully.

- Some queued task elements were higher or lower priority.

- The ACMS$SELECTION_STRING workspace contains the queued task element ID in binary format. A queued task element ID is generated for each queued task element. The element ID is unique for all time and space.

9.6.2. Setting ACMSGEN Parameters for the QTI Process

Table 9.1, "ACMSGEN Parameters Associated with QTI" shows the ACMSGEN parameters that are associated with the QTI process.

Table 9.1. ACMSGEN Parameters Associated with QTI

Parameter Name	Default	Range	Dynamic
QTI_POLLING_TIMER	5000	1 through –1 milliseconds	Yes
QTI_PRIORITY	4	0 through 31	No
QTI_SUB_TIMEOUT	7200	1 through –1 seconds	Yes
QTI_RETRY_TIMER	1800	1 through –1 seconds	Yes
QTI_USERNAME	SYSTEM	None	No

The next sections briefly explain these parameters. For more detail, see *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/].

9.6.2.1. Assigning a User Name to the QTI Process

You use the ACMSGEN parameter QTI_USERNAME to assign a user name to the QTI process. Be sure that the user name you assign has the SYSPRV and SYSLCK privileges. SYSPRV is needed for the QTI process in order to dequeue tasks from task queues. SYSLCK is needed for the QTI process to use the OpenVMS lock manager. See *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/] for information on setting up the user name of the QTI.

If the user name of the QTI has been defined with ACMS agent privilege in the User Definition Utility (UDU) user authorization file, then the QTI submits tasks under the enqueuer user name. See *Section 9.5.1, "Queuing Tasks Using the* ACMS$QUEUE_TASK *Service"* for more information about the enqueuer user name. If the user name of the QTI does not have the ACMS agent privilege, then QTI submits tasks under the QTI user name.

See *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/] for information about assigning the agent privilege.

9.6.2.2. Assigning a Priority to the QTI Process

You use the ACMSGEN parameter QTI_PRIORITY to assign an OpenVMS priority to the QTI process.

9.6.2.3. Controlling Submitter Sign-Ins

The QTI is an ACMS Systems Interface (SI) agent program. As with all agent programs, the QTI uses the ACMS$SIGN_IN service to sign in a submitter before invoking a task on behalf of that submitter. Normally, an agent program determines when to sign out a submitter based on some submitter action. For example, the submitter might type "LOGOUT". However, the submitters being handled by the

QTI are not active users who interact with the ACMS system (they are simply queued task elements). Therefore, the QTI leaves submitters signed in for a specified interval of time.

Because each submitter that is signed in to ACMS consumes memory resources, the QTI process uses a mechanism that enables it to bypass a submitter sign-in for each task. When the QTI process dequeues a queued task element, it checks the user name of that element. If it is the first time it has dequeued an element with that user name, it signs it in to ACMS but it does not sign out of ACMS when the task ends. Instead, it leaves the submitter signed in for a certain amount of time before signing it out. Consequently, if another queue element with the same user name is dequeued, the user name is already signed in.

You use the ACMSGEN parameter QTI_SUB_TIMEOUT to indicate how long a submitter can remain inactive before the QTI process signs that submitter out of ACMS. To determine how many submitters are signed in under a QTI process, use the **ACMS/SHOW QTI** command (see *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/] for more information on operator commands).

9.6.2.4. Setting the Retry Time for Failed Tasks

You use the ACMSGEN parameter QTI_RETRY_TIMER to determine how long the QTI process waits before it retries a task that it has already dequeued, but that did not complete successfully and that can be retried. For example, if the QTI process dequeues a task for an application that is not started, the QTI process waits for the number of seconds identified by the QTI_RETRY_TIMER parameter before retrying the task.

9.6.2.5. Setting the Polling Time for Task Queues

You use the ACMSGEN parameter QTI_POLLING_TIMER to specify the amount of time that the QTI process waits to poll populated queues that previously had an RMS lock outstanding.

9.6.3. Auditing Done by the QTI Process

The QTI audits all ACMS operator commands that start, stop, or set queues. In addition, the QTI audits all task invocations that fail. The audit trail record of failed task invocations shows the following:

- The error that caused the task invocation to fail
- What the QTI has done with the queued task element

Example 9.1, "Sample QTI Audit Entry" shows a sample QTI audit entry.

VSI ACMS for OpenVMS Managing Applications [https://docs.vmssoftware.com/vsi-acms-managing-applications/] provides more information on application auditing with the ACMS Audit Trail Report Utility.

Example 9.1. Sample QTI Audit Entry

```
*********************************************************
Type    : COMMAND    Time    : 24-NOV-1987 16:50:28.38
User    : OPERATOR
Text    : Successful start for queue PAYROLL_QUEUE
*********************************************************
Type    : ERROR      Time    : 24-NOV-1987 16:52:04.62
Queue   : PAYROLL_QUEUE
ErrQue  : PAYROLL_ERROR_QUEUE
Elem Id: MYNODE::28000114-00000003-87A9ECE0-0090A5F7
```

```
Appl    : PAYROLL
Task    : HIRE_EMPLOYEE
User    : JONES
Text    : Error processing queued task
-ACMSQUE-E-ERRGETPROC, Error returned from ACMS$GET_PROCEDURE_INFO
-ACMS-E-NOSUCH_PKG, There is no such package defined
-ACMSQUE-I-QTRETRY, Queued task will be retried later
*********************************************************
```

9.6.4. How the QTI Handles Errors

Once the QTI has submitted a task to an ACMS application, it must handle any errors that result when the task is processed. The QTI handles errors in one of four ways:

1. Retries the task. *Table 9.2, "Errors That Result in Queued Task Retry"* lists the errors that result in this action.

   Table 9.2. Errors That Result in Queued Task Retry

Error	QTI assumes that
ACMS$_APPL_NOT_STARTED	The application is not started.
ACMS$_CALL_CANCELLED	The task was canceled, perhaps due to a system or application shutdown.
ACMS$_INVPROCID	The application stopped unexpectedly.
ACMS$_MAX_TASKS	There is a temporary resource limitation.
ACMS$_NOSUCH_PKG	The application is not started.
ACMS$_QTI_RETRY	The task returns this value as the task completion status, thereby directing the QTI to retry this queued task later.
ACMS$_SRVDEAD	The server stopped unexpectedly.
ACMS$_TASK_DISABLED	The task will be enabled later.
ACMS$_WSPALLOCERR	There is a temporary resource limitation.
ACMS$_WSPLOCKED	There is a temporary resource limitation.

2. Does not retry the task, deletes the queued task element from the task queue and writes it to the error queue, if any. *Table 9.3, "Errors That Result in Writing Queued Task Elements to an Error Queue"* lists the errors that result in this action.

   Table 9.3. Errors That Result in Writing Queued Task Elements to an Error Queue

Error	Cause of Error
ACMS$_BADUSER	QTI was unable to sign in the user name.
ACMS$_CANTRETRY	The submitter services have tried to connect to server services twice, using different protocols, without success.
ACMS$_ERRREADARG	There is a problem with the workspaces being used.
ACMS$_INVQUEELM	Invalid queue element; assume the queue is corrupt.

Error	Cause of Error
ACMS$_NOSUCH_PROC	The task does not exist in the application.
ACMS$_NOTRANSADB	The task definition has been modified to participate in a distributed transaction, but the application database (ADB) has not been rebuilt.
ACMS$_NOTRANSNODE	The application node is not running ACMS Version 3.2 or later.
ACMS$_QTI_NORETRY	The task has returned this value as the task completion status, thereby directing the QTI not to retry this queued task.
ACMS$_SECURITY_CHECK_FAILED	The user name is not allowed access in the task ACL.
ACMS$_TASKARGWSPERR	The number or size of workspaces supplied to the task is incorrect.
ACMS$_TASKNOTCOMP	The task is not composable.
QUE$_INVIOMETH	The task being selected uses terminal, stream, or request (TDMS) I/O.

3. Retries the task once. If the same error occurs twice consecutively, then the QTI does not retry the task. Instead, it deletes it from the task queue and writes it to the error queue, if any. Any error not shown in *Table 9.2, "Errors That Result in Queued Task Retry"* or *Table 9.3, "Errors That Result in Writing Queued Task Elements to an Error Queue"* can cause this action.

4. Immediately retries the task up to five times. If one of the errors is still returned after five retries, the QTI handles the error according to the third way listed above. *Table 9.4, "Errors That Result in Immediate Retry of Queued Task "* lists the errors that result in this action.

Table 9.4. Errors That Result in Immediate Retry of Queued Task

Error	Cause of Error
ACMS$_QTI_RETRY_IMMEDIATELY	The task has returned this value as the completion status, thereby directing the QTI to retry the queued task immediately.
ACMS$_TRANSTIMEDOUT	The distributed transaction did not complete with the time limit specified in the application definition.

See *Section 9.7, "Processing Error Queues"* for information about how to process elements from an error queue.

If a task invocation does not complete successfully, the QTI writes an audit record to the ACMS audit trail log. The audit record indicates the reason for the task invocation failure as well as what the QTI did with the queued task element (to be retried later, moved to the error queue, or deleted from the task queue).

Note

There are three error messages, ACMS$_QTI_RETRY, ACMS$_QTI_RETRY_IMMEDIATELY, and ACMS$_QTI_NORETRY, that a queued task can return that explicitly direct whether or not the QTI retries the queued task.

9.7. Processing Error Queues

You process elements from an error queue in the same way that you can process elements from any ACMS task queue; you use the ACMS$DEQUEUE_TASK service.

Application programs can access queue files to queue and dequeue task elements using the ACMS queuing services. The queue or dequeue operation can be atomic with other operations by including them in a distributed transaction.

For example, a task definition includes the following operations within a distributed transaction:

- Use the ACMS$DEQUEUE_TASK service to dequeue queued task elements from an error queue.
- Correct the field that caused the task invocation to fail (which presumably is why the queued task element is on the error queue).
- Write a statistical record to an RMS file.
- Use the ACMS$QUEUE_TASK service to queue the queued task element back to a task queue. The QTI can then retry the queued task element.

By performing the operations in this example within a distributed transaction, ACMS ensures that all operations occur or none of the operations occur. For example, if a system failure occurs after dequeuing the task and writing the statistics record but before enqueuing the task, then ACMS rolls back the dequeue operation and the write operation to the state they were in before the operation began.

Example 9.2, "A Task That Dequeues from an Error Queue" shows a task that can be selected by a terminal user to dequeue a queued task element from an error queue, correct the element, and queue the corrected element to another task queue.

Example 9.2. A Task That Dequeues from an Error Queue

```
!
!   CORR_QTE_ERR.TDF This task calls a program to read queued task
!   element entries from the error queue PAY_ERR_QUE.
!   The entry will be displayed to a terminal operator.
!   The operator will either correct the entry and
!   resubmit the task to the PAY_QUE, or delete
!   the entry without further processing.

!
SET LOG CORR_QTE_ERR.TDF_LOG
REPLACE TASK CORR_QTE_ERR
DEFAULT SERVER IS QUE_EXAM_SERVER;
WORKSPACES ARE QTE_INFO, EMP_PAY_REC, QUE_MISC, CONT_PROC;

BLOCK WORK WITH REQUEST I/O
!
!  Start a distributed transaction, then call procedure to read the next
!  entry in the error queue PAY_ERR_QUE. QTE_INFO contains the queued
!  task element structure, and EMP_PAY_REC is the data record associated
!  with the task.
!
GET_ENTRY:

BLOCK WORK WITH DISTRIBUTED TRANSACTION
```

```
     PROCESSING
      CALL GET_QTE_ERRENT USING QTE_INFO,
           EMP_PAY_REC;
!
!  If the procedure call fails, roll back the transaction and cancel the
!  task.  If not, use GET ERROR MESSAGE to translate this QTE's error
!  status and display that text on the form.
!
     CONTROL FIELD ACMS$T_STATUS_TYPE
     "B" : ROLLBACK TRANSACTION;
           CANCEL TASK;
     "G" : GET ERROR MESSAGE NUMBER QTE_LAST_ERR_SYM INTO QTE_LAST_ERR;
     END CONTROL FIELD;

 DISPLAY_ENTRY:
     EXCHANGE
      REQUEST IS DISP_QTE_ERRENT USING QTE_INFO, QUE_MISC;
!
!  REQU will cause the entry to be queued back onto the PAY_QUE queue
!  for processing.  DELE will cause the removal of the entry from the
!  error queue to be committed.  QUE_MISC contains the PROGRAM_REQUEST_KEY.
!
     CONTROL FIELD IS PROGRAM_REQUEST_KEY
       "REQU"   : GOTO STEP REQUE_ENTRY;
       "DELE"   : EXIT BLOCK;
       NOMATCH  : ROLLBACK TRANSACTION;
                  CANCEL TASK;
     END CONTROL FIELD;

REQUE_ENTRY:
     PROCESSING
       CALL REQUE_QTE_ERRENT USING QTE_INFO;

     CONTROL FIELD ACMS$T_STATUS_TYPE
     "B"      : ROLLBACK TRANSACTION;
                CANCEL TASK;
     "G"      : EXIT BLOCK;
     NOMATCH  : ROLLBACK TRANSACTION;
                CANCEL TASK;
     END CONTROL FIELD;

END BLOCK;

ACTION IS
   COMMIT TRANSACTION;

CHECK_CONTINUE:

     EXCHANGE
      READ CONT_PROC WITH PROMPT
      "Enter Y to continue, N to EXIT TASK==> ";

     ACTION
      CONTROL FIELD IS CONT_PROC.CONT_PROC_CHECK
      "Y"      : REPEAT TASK;
      "N"      : EXIT TASK;
      NOMATCH  : CANCEL TASK;
      END CONTROL FIELD;
```

```
END BLOCK WORK;
END DEFINITION;
```

By starting a distributed transaction on the nested block step, the CORR_QTE_ERR task ensures that the GET_ENTRY, DISPLAY_ENTRY, and REQUE_ENTRY steps complete successfully or are rolled back.

Be sure that you enable journaling for both the queue file from which you are dequeuing elements and the queue file to which you are enqueuing elements. Use the DCL command **SET FILE/RU_JOURNALING** to enable journaling on those files.

Example 9.3, "A Dequeue Procedure" shows the GET_QTE_ERRENT procedure called in the first processing step of the task shown in *Example 9.2, "A Task That Dequeues from an Error Queue"*. *Example 9.4, "An Enqueue Procedure"* shows the REQUE_QTE_ERRENT procedure that is called in the second processing step of that task.

Example 9.3. A Dequeue Procedure

```
IDENTIFICATION DIVISION.
*****************************************************************
*                                                               *
PROGRAM-ID. GET_QTE_ERRENT.
*                                                               *
*                                                               *
*               Version:        01                              *
*               Edit date:      06-APR-1988                     *
*                                                               *
*****************************************************************

*****************************************************************
*         P R O G R A M    D E S C R I P T I O N                *
*                                                               *
* GET_QTE_ERRENT is called from task CORR_QTE_ERR.  This        *
*  ACMS procedure calls ACMS$DEQUEUE_TASK to dequeue a          *
*  queued task element (QTE) from the error queue               *
*  PAY_ERR_QUEUE.                                               *
*                                                               *
* INPUT:  QTE_INFO, a record containing the queue task          *
*         element fields, queue name and queued task            *
*         element information.                                  *
*  EMP_PAY_REC, the employee record needed for                  *
*   the task.                                                   *
*                                                               *
* OUTPUT: The queued task element information returned          *
*   in QTE_INFO and the data returned to EMP_PAY_REC            *
*         from ACMS$DEQUEUE_TASK call.                          *
*                                                               *
*                                                               *
*****************************************************************
                                                 *
*                                                               *
*                                                               *
*                   C O P Y R I G H T                           *
*                                                               *
*                                                               *
*                                                               *
*                                                               *
```

```
* © Copyright 2006 Hewlett-Packard Development Company, L.P. *
*                                                             *
*   VSI assumes no responsibility for the use or               *
*   reliability of its software on equipment that is not      *
*   supplied by VSI.                                           *
*                                                             *

*                                                             *
***************************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.
***************************************************************
DATA DIVISION.

***************************************************************
*                                                             *
*          C O M M O N   D E C L A R A T I O N S              *
*                                                             *
***************************************************************
WORKING-STORAGE SECTION.
*
* Set up fields to receive element id from ACMS$DEQUEUE_TASK.
*
01 RET-QTE-ID.
   03  PIP   PIC S9(9) COMP.
   03  SEQ-NO   PIC S9(9) COMP.
   03  ADT1   PIC S9(9) COMP.
   03  ADT2   PIC S9(9) COMP.
   03  NODE-NAME-LEN  PIC X.
   03  NODE-NAME  PIC X(15) VALUE ''.

*
01 STATUS-RESULT   PIC S9(9) COMP.

* Set up a descriptor list with number of workspaces this task
*  requires and pointers to each individual workspace descriptor.
*  The USAGE IS POINTER allows us to define the address at runtime.
*
01 WS_DESC_LIST.
   05 DESC_LIST_CNT             PIC S9(9) COMP VALUE 1.
   05 QTE_WS_DESC_PTR           USAGE IS POINTER.
*
* Set up a descriptor for each workspace being passed with the
*  workspace size and a pointer to the address of the record,
*  also to be defined at runtime.
*
01 QTE_WS_DESC.
   05 EMP_PAY_REC_SIZE          PIC S9(9) COMP VALUE IS 13.
   05 EMP_PAY_REC_PTR           USAGE IS POINTER.
***************************************************************

LINKAGE SECTION.
COPY "QTE_INFO" FROM DICTIONARY.
COPY "EMP_PAY_REC" FROM DICTIONARY.
***************************************************************
PROCEDURE DIVISION USING QTE_INFO,
```

```
    EMP_PAY_REC
            GIVING STATUS-RESULT.
*************************************************************
MAIN-PROCEDURE.
*
* Set up addresses for the pointers to reference.
*
 SET QTE_WS_DESC_PTR TO REFERENCE OF QTE_WS_DESC.
 SET EMP_PAY_REC_PTR TO REFERENCE OF EMP_PAY_REC.
*
* Enable the flag to say we do not want to wait if the queue is
*  empty.
*
 MOVE 2 TO FLAG.

CALL-DEQUE.

        CALL "ACMS$DEQUEUE_TASK" USING BY DESCRIPTOR PAY_ERR_QUE,
                                                     OMITTED,
                                       BY REFERENCE  FLAG,
                                       BY DESCRIPTOR RET_TASK,
                                       BY DESCRIPTOR RET_APPL,
                                       BY REFERENCE WS_DESC_LIST,
                                       BY REFERENCE RET_WKSP_CNT,
                                       BY REFERENCE QTE_PRIO,
                                       BY DESCRIPTOR QTE_SUBMITTER,
                                       BY REFERENCE RET-QTE-ID,
                                       BY REFERENCE QTE_ERR_CNT,
            BY REFERENCE QTE_LAST_ERR_SYM,
            BY REFERENCE QTE_ERR_ADT
     GIVING STATUS-RESULT.

*
* Move information into EMP_PAY_REC returned by the call to the QTE_INFO
* workspace, as it will be used by the request to display both record
* data and QTE information.
*

 MOVE BADGE OF EMP_PAY_REC TO BADGE OF QTE_INFO.
 MOVE PAYCODE OF EMP_PAY_REC TO PAYCODE OF QTE_INFO.
 MOVE HOURS OF EMP_PAY_REC TO HOURS OF QTE_INFO.
 MOVE WAGE OF EMP_PAY_REC TO WAGE OF QTE_INFO.
 MOVE VACATION OF EMP_PAY_REC TO VACATION OF QTE_INFO.
 MOVE SPACES TO QTE_SUBMITTER OF QTE_INFO.
*
* Move PID into displayable field to be returned to the request.
*
SPACE-MOVE.

 MOVE NODE-NAME TO QTE_NODE.
 MOVE PID TO QTE_PID.

EXIT-GET-QTE-ERRENT.

 EXIT PROGRAM.
```

Example 9.4. An Enqueue Procedure

```
IDENTIFICATION DIVISION.
*****************************************************************************
*                                                                           *
PROGRAM-ID. REQUE_QTE_ERRENT.
*                                                                           *
*               Version:        01                                          *
*               Edit date:      06-APR-1988                                 *
*                                                                           *
*****************************************************************************
*****************************************************************************
*         P R O G R A M    D E S C R I P T I O N                            *
*                                                                           *
* REQUE_QTE_ERRENT is called from task CORR_QTE_ERR.  This ACMS procedure   *
* calls ACMS$QUEUE_TASK to requeue a queued task element (QTE) entry from   *
* the PAY_ERR_QUE.  The QTE is requeued to the PAY_QUE queue.               *
*                                                                           *
* INPUT:  QTE_INFO, a record containing the task element information        *
*         to be queued.                                                     *
*   EMP_PAY_REC, a record contains the user data needed by the task.        *
*                                                                           *
*****************************************************************************
*                                                                           *
*                                                                           *
*                      C O P Y R I G H T                                    *
*                                                                           *
*                                                                           *
*                                                                           *
* © Copyright 2006 Hewlett-Packard Development Company, L.P.                *

*                                                                           *
*                                                                           *
*   VSI assumes no responsibility for the use or reliability of its
 *
*   software on equipment that is not supplied by VSI.
 *
*****************************************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.
*****************************************************************************
DATA DIVISION.
*****************************************************************************
*                                                                           *
*          C O M M O N   D E C L A R A T I O N S                            *
*                                                                           *
*****************************************************************************
WORKING-STORAGE SECTION.
*
01 STATUS-RESULT   PIC 9(9) COMP VALUE IS 0.
*
* Copy in record this task needs to run.
*
COPY "EMP_PAY_REC" FROM DICTIONARY.
*
```

```
* Set up descriptor list with number of workspaces we are passing
*  and pointers to each individual workspace descriptor.
*
01 WS_DESC_LIST.
   05 DESC_LIST_CNT  PIC S9(9) COMP VALUE 1.
   05 QTE_WS_DESC_PTR  POINTER VALUE IS REFERENCE QTE_WS_DESC.

01 QTE_WS_DESC.
   05 EMP_PAY_REC_SIZE  PIC S9(9) COMP VALUE IS 13.
   05 EMP_PAY_REC_PTR  POINTER VALUE IS REFERENCE EMP_PAY_REC.

****************************************************************
LINKAGE SECTION.
COPY "QTE_INFO" FROM DICTIONARY.
****************************************************************
PROCEDURE DIVISION USING QTE_INFO
            GIVING STATUS-RESULT.
****************************************************************
MAIN-PROCEDURE.
*
* Move data received from the call to ACMS$DEQUEUE_TASK, and
*  passed to the operator, back to EMP_PAY-REC.
*
 MOVE BADGE IN QTE_INFO TO BADGE IN EMP_PAY_REC.
 MOVE PAYCODE IN QTE_INFO TO PAYCODE IN EMP_PAY_REC.
 MOVE WAGE IN QTE_INFO TO WAGE IN EMP_PAY_REC.
 MOVE VACATION IN QTE_INFO TO VACATION IN EMP_PAY_REC.

CALL-QUEUE.

 CALL "ACMS$QUEUE_TASK" USING BY DESCRIPTOR PAY_QUE,
         BY DESCRIPTOR RET_TASK,
         BY DESCRIPTOR RET_APPL,
         BY REFERENCE WS_DESC_LIST,
         BY REFERENCE FLAG,
         BY REFERENCE QTE_PRIO
    GIVING STATUS-RESULT.


EXIT-REQUE-QTE-ERRENT.

 EXIT PROGRAM.
```

9.8. Debugging Queued Tasks

There are several sources of information or methods you can use when debugging queued tasks:

- Look at the audit trail log. The QTI process puts entries in the audit trail log for certain kinds of errors. See *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/] for more information. Because the QTI is an agent program and uses the Systems Interface to submit tasks to an ACMS application, error messages in the audit trail log are likely to be Systems Interface messages. Therefore, you might want to learn enough about the Systems Interface to understand these messages. See *VSI ACMS for OpenVMS Systems Interface Programming* [https://docs.vmssoftware.com/vsi-acms-sys-interface-prog/] for more information about the ACMS Systems Interface.
- Check the Software Event Log. In some cases, there is an entry there.

- Use the following techniques to isolate problems:
  - Queue the queued task element, then use the ACMSQUEMGR **SHOW ELEMENT** command to examine the element in the queue.
  - Use the ACMS$DEQUEUE_TASK service to dequeue the task. By dequeuing the task yourself, you can look at the workspace contents of the queued task element.
  - Run the task interactively. For example, write a test task that calls the task that would have been invoked by the QTI. The test task can fill in the necessary arguments to be passed to the called task.
- Check the accessibility of each entity that is involved in the queue and dequeue operation. Specifically, check the ACMSQDF.DAT file, the queue file, and the access control list on the task. Check the privileges of the QTI process and the enqueuing process to make sure that these two processes have the necessary privileges and access rights to the entities just listed.

9.9. Online Backup of Task Queue Files

ACMS lets you back up task queue files without having to stop the QTI or programs that call the ACMS$QUEUE_TASK and ACMS$DEQUEUE_TASK services. To enable a program that calls a queuing service to continue processing while you back up a task queue file, have the program check for the return status of ACMS$_QUEENQSUS, ACMS$_QUEDEQSUS, or both. The ACMS$QUEUE_TASK service returns the ACMS$_QUEENQSUS status if the enqueue operations for that task queue file are suspended. The ACMS$DEQUEUE_TASK service returns the status ACMS$_QUEDEQSUS if the dequeue operations for that task queue file are suspended.

If a queuing service returns one of these statuses, the program can set a timer and then retry the call. For example, if the ACMS$QUEUE_TASK service returns the ACMS$_QUEENQSUS status, the program could wait for 10 seconds and then retry the call. The program would continue to retry the call every 10 seconds until the ACMS$QUEUE_TASK service completed successfully.

See *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/] for information on how to back up a task queue file.

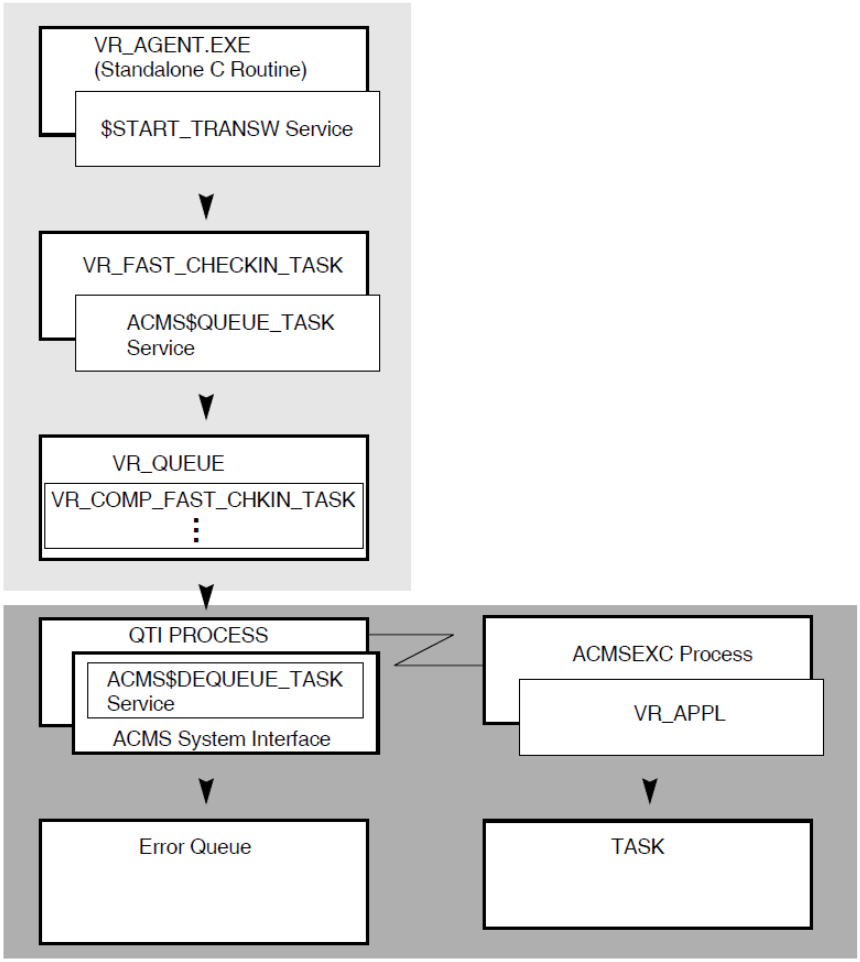
9.10. Queuing Example

This section shows an example of a car rental reservation application that uses the ACMS queuing facility. The purpose of the application is to process the customer's account when the customer returns the rental car. Because there might be peak periods when many customers return their cars at the same time, the application accepts the necessary reservation information from a customer, then queues that information to be processed at a later, less busy time. The example consists of:

- A C routine that gets reservation information from the terminal user, starts a distributed transaction, calls a task, performs error handling, and commits or rolls back the distributed transaction
- A task that validates the reservation information entered by the terminal user and calls a procedure to enqueue a task
- A COBOL procedure used in the processing step of the task to enqueue a task
- A queued task that is called by the QTI process

Figure 9.2, "A Queuing Example" shows how these items work together.

Figure 9.2. A Queuing Example



The following is the flow of events leading up to and including the example:

1. The application manager does the following:
   - Defines queue security
   - Uses the UDU to define agent privileges for the QTI
   - Uses the ACMSQUEMGR **CREATE** command to create the queue VR_QUEUE and the error queue
   - Enables RMS journaling for the queue file and the error queue file
   - Sets ACMSGEN parameters for the QTI
   - Starts the QTI process
   - Starts the VR_QUEUE queue with the error queue
2. The VR_AGENT agent gets information from the terminal user, starts a distributed transaction, and calls the VR_FAST_CHECKIN_TASK
3. The VR_FAST_CHECKIN_TASK validates the information entered by the terminal user, and queues the task VR_COMP_FAST_CHKIN_TASK to the queue VR_QUEUE

4. The QTI process dequeues VR_COMP_FAST_CHKIN_TASK and submits the task to the application

5. VR_COMP_FAST_CHKIN_TASK runs

Example 9.5, "C Agent that Starts a Distributed Transaction" shows the main part and the process_this_transaction subroutine of the C agent that uses the $START_TRANSW system service to start a distributed transaction, and calls the VR_FAST_CHECKIN_TASK. See *VSI ACMS for OpenVMS Systems Interface Programming* [https://docs.vmssoftware.com/vsi-acms-sys-interface-prog/] for a detailed explanation of the complete agent code.

Example 9.5. C Agent that Starts a Distributed Transaction

```
/************************************************************/
/*                                                          */
/*               Version:        01                         */
/*               Edit dates:     06-MAR-90                  */
/*               Authors:        VSI                         */
/*                                                          */
/************************************************************/
/************************************************************/
/*         F U N C T I O N A L   D E S C R I P T I O N      */
/*                                                          */
/*                                                          */
/*    VR_AGENT is an ACMS agent program that acts like an ATM */
/*    where you type in your reservation number and odometer  */
/*    reading, drop the keys in a slot, and walk away.  The   */
/*    system bills you later for the amount you owe.  The     */
/*    agent uses QIOs to get the data, starts a distributed   */
/*    transaction, then calls a task to do the work. The task */
/*    consists of a nonparticipating step that validates the  */
/*    reservation number, a step that queues a task to do the */
/*    actual checkin work, and a step that writes a history   */
/*    record.  If the task succeeds, the agent commits the    */
/*    transaction.  If the task  fails, the agent aborts the  */
/*    the transaction and notifies the user of the problem.   */
/*    The agent is also responsible for handling errors, such */
/*    as transaction timeouts.                                */
/*                                                          */
/************************************************************/
/************************************************************/
/*                                                          */
/*                   C O P Y R I G H T                      */
/*                                                          */
/* © Copyright 2006 Hewlett-Packard Development Company, L.P. */
/*                                                          */
/*     Confidential computer software. Valid license          */
/*     from VSI required for possession, use or copying.       */
/*     Consistent with FAR 12.211 and 12.212, Commercial      */
/*     Computer Software, Computer Software Documentation,    */
/*     and Technical Data for Commercial Items are licensed   */
/*     to the U.S. Government under vendor's standard         */
/*     commercial license.                                    */
/*                                                          */
/*     The information contained herein is subject to         */
/*     change without notice. The only warranties for VSI      */
/*     products and services are set forth in the express     */
```

```
/*     warranty statements accompanying such products and     */
/*     services. Nothing herein should be construed as        */
/*     constituting an additional warranty. VSI shall not be    */
/*     liable for technical or editorial errors or omissions  */
/*     contained herein.                                      */
/*                                                            */

/************************************************************/


                              /* . */
                              /* . */
                              /* . */
                              /* . */
main ()

/************************************************************************/
/*                                                                      */
/*  Get Procedure information to see if the application is running.   */
/*                                                                      */
/*  While the application is up and running prompt user for           */
/*  reservation ID and odometer reading.                                */
/*                                                                      */
/*  If the user enters the data, process the fast checkin transaction.*/
/*                                                                      */
/*  If the user aborts, notify the user that the transaction was not  */
/*  processed.                                                          */
/*                                                                      */
/************************************************************************/
{
    for (;;)
    {
        status = initialization ();

        check_status(status);


        status = ACMS$GET_PROCEDURE_INFO(&submitter_id,
                                         &task_name_desc,
                                         &appl_name_desc,
                                         &task_info_list);

        while (status & STS$M_SUCCESS)
        {
            status = get_data ();

            if (status & STS$M_SUCCESS)
                status = process_this_transaction();
            else if (status == RMS$_EOF)
                    status = report_user_abort();

            check_status(status);

            status = ACMS$GET_PROCEDURE_INFO(&submitter_id,
                                             &task_name_desc,
                                             &appl_name_desc,
                                             &task_info_list);
        }
```

```
        if (status == ACMS$_NOSUCH_PKG)
            status = application_not_running();

        check_status(status);

        status = termination ();

        check_status(status);

    }

}
                            */ . */
                            */ . */
                            */ . */
                            */ . */
process_this_transaction()

/*****************************************************************/
/*                                                               */
/*    Start Transaction. Call the task. Commit if successful.    */
/*    Abort if failure. Retry if timed out. Notify user whether  */
/*    transaction succeeded or failed.                           */
/*                                                               */
/*****************************************************************/

{
    short retry, trans_completed;
    retry = 0;
    trans_completed = FALSE;

    while ((trans_completed == FALSE) && (retry < MAX_RETRY))
    {
        status = SYS$START_TRANSW (0,0,&iosb,0,0,tid);

        if (status & STS$M_SUCCESS)
            status = iosb.status;

        check_status(status);

        status = call_return_task();

        if (status & STS$M_SUCCESS)
        {
            status = SYS$END_TRANSW (0,0,&iosb,0,0,tid);

            if (status & STS$M_SUCCESS)
                status = iosb.status;

            check_status(status);

            trans_completed = TRUE;
        }
        else
        {
            if ((status == ACMS$_TRANSTIMEDOUT) ||
                (status == ACMS$_SRVDEAD) ||
```

```
                (status == RDB$_DEADLOCK) ||
                (status == RDMS$_DEADLOCK) ||
                (status == RDB$_LOCK_CONFLICT) ||
                (status == RDMS$_LCKCNFLCT) ||
                (status == RDMS$_TIMEOUT))
                ++retry;
            else
                retry = MAX_RETRY;

            status = SYS$ABORT_TRANSW (0,0,&iosb,0,0,tid);

            if (status & STS$M_SUCCESS)
                status = iosb.status;

            check_status(status);
        }
    }
    if (trans_completed == FALSE)
        status = notify_failure();
    else
        status = notify_success();

    return status;
}
```

If the distributed transaction fails because of an error that can be handled, such as server deadlock or transaction timeout, the agent retries the distributed transaction up to five times. *Example 9.6, "VR_FAST_CHECKIN_TASK Definition"* shows the task definition called by the C agent.

Example 9.6. VR_FAST_CHECKIN_TASK Definition

```
REPLACE TASK AVERTZ_CDD_TASK:VR_FAST_CHECKIN_TASK

USE WORKSPACES  VR_FAST_CHECKIN_WKSP,
                VR_RESERVATIONS_WKSP,
                VR_RENTAL_CLASSES_WKSP,
                VR_VEHICLES_WKSP,
                VR_TRANS_WKSP,
                VR_VEHICLE_RENTAL_HISTORY_WKSP,
                VR_HIST_WKSP;

TASK ARGUMENTS ARE VR_FAST_CHECKIN_WKSP WITH ACCESS READ;

! TASK MUST BE COMPOSABLE TO BE CALLED AS PART OF A DISTRIBUTED
! TRANSACTION

BLOCK WORK WITH DISTRIBUTED TRANSACTION
                NO I/O

!
! Retrieve the reservation record, using the reservation number/ID
! entered by the customer and passed by the vr_agent agent.
!

        PROCESSING WITH NONPARTICIPATING SERVER
                CALL PROCEDURE  VR_FIND_RES_PROC
                  IN    VR_READ_SERVER
                USING   VR_FAST_CHECKIN_WKSP,
```

```
                        VR_RESERVATIONS_WKSP;

        ACTION IS
            IF (ACMS$T_STATUS_TYPE = "G")
            THEN
               MOVE VR_FAST_CHECKIN_WKSP.ACTUAL_RETURN_DATE
                TO VR_VEHICLE_RENTAL_HISTORY_WKSP.ACTUAL_RETURN_DATE,
                   VR_FAST_CHECKIN_WKSP.RETURN_ODOMETER_READING
                TO
                  VR_VEHICLE_RENTAL_HISTORY_WKSP.RETURN_ODOMETER_READING;
            ELSE
                CANCEL TASK RETURNING  ACMS$L_STATUS;
            END IF;


!
! RETRIEVE THE VEHICLE AND VEHICLE_RENTAL_HISTORY RECORDS
!

        PROCESSING WITH NONPARTICIPATING SERVER
                CALL PROCEDURE  VR_FIND_VE_VRH_PROC
                  IN    VR_READ_SERVER
                USING   VR_RESERVATIONS_WKSP,
                        VR_VEHICLES_WKSP,
                        VR_VEHICLE_RENTAL_HISTORY_WKSP,
                        VR_RENTAL_CLASSES_WKSP,
                        VR_TRANS_WKSP;

        ACTION IS
            IF (ACMS$T_STATUS_TYPE = "B") THEN
                CANCEL TASK RETURNING  ACMS$L_STATUS;
            END IF;
!
! QUEUE THE TASK TO BE RUN LATER
!

        PROCESSING
                CALL PROCEDURE    VR_ENQ_FAST_CHECKIN
                  IN    VR_QUEUE_SERVER
                USING   VR_FAST_CHECKIN_WKSP;

        ACTION IS
            IF  (ACMS$T_STATUS_TYPE = "G")
            THEN
                MOVE  "FASTCHIN"            TO VR_HIST_WKSP.TRANS_TYPE,
                          VR_VEHICLES_WKSP.VEHICLE_ID
                                            TO VR_HIST_WKSP.VEHICLE_ID;
            ELSE
                CANCEL TASK RETURNING  ACMS$L_STATUS;
            END IF;
!
! WRITE A RECORD OF A SUCCESSFUL CHECK IN TO THE HISTORY DATABASE
!
        PROCESSING
            CALL PROCEDURE    VR_WRITE_HIST_RECORD_PROC
            IN      VR_LOG_SERVER
            USING   VR_HIST_WKSP,
                    VR_RESERVATIONS_WKSP;
```

```
END BLOCK;
END DEFINITION;
```

Because the VR_FAST_CHECKIN_TASK is joining a distributed transaction started by the agent, the root block step must use the TRANSACTION phrase.

The first processing step in VR_FAST_CHECKIN_TASK uses the reservation ID, obtained from the customer by the agent, to retrieve the reservation record. The second processing step retrieves the car history record. Because the first two processing steps perform read-only operations, neither step participates in the distributed transaction. Both steps use the NONPARTICIPATING SERVER phrase.

The third processing step calls the VR_ENQ_FAST_CHECKIN procedure to queue the VR_COMP_FAST_CHKIN_TASK.

The final processing step writes a record of the transaction into the database. If the procedure completes successfully, the task ends, and the agent calls the transaction services to commit the distributed transaction. *Example 9.7, "Enqueuing Routine"* shows the COBOL procedure that the VR_FAST_CHECKIN_TASK calls to enqueue a task.

Example 9.7. Enqueuing Routine

```
IDENTIFICATION DIVISION.
*****************************************************************
PROGRAM-ID. VR-ENQ-FAST-CHECKIN.
*                                                               *
*               Version:        01                              *
*               Edit:           00                              *
*               Edit dates:     16-OCT-1990                     *
*               Authors:        VSI                              *
*               Called From:    AGENT                           *
*                                                               *
*****************************************************************
*****************************************************************
*         F U N C T I O N A L   D E S C R I P T I O N           *
*                                                               *
* This routine is called by the VR_FAST_CHECKIN_TASK.           *
* The VR_FAST_CHECKIN_WKSP is initialized by the VR_AGENT.      *
* The VR_COMP_FAST_CHKIN_TASK uses the information in the       *
* workspace to complete the checkin database functions.         *
*                                                               *
*****************************************************************
*****************************************************************
*                                                               *
*                   C O P Y R I G H T                           *
*                                                               *
* © Copyright 2006 Hewlett-Packard Development Company, L.P.    *
*                                                               *
*     Confidential computer software. Valid license             *
*     from VSI required for possession, use or copying.          *
*     Consistent with FAR 12.211 and 12.212, Commercial         *
*     Computer Software, Computer Software Documentation,       *
*     and Technical Data for Commercial Items are licensed      *
*     to the U.S. Government under vendor's standard            *
*     commercial license.                                       *
*                                                               *
*     The information contained herein is subject to            *
*     change without notice. The only warranties for VSI         *
```

```
*     products and services are set forth in the express     *
*     warranty statements accompanying such products and     *
*     services. Nothing herein should be construed as        *
*     constituting an additional warranty. VSI shall not be    *
*     liable for technical or editorial errors or omissions  *
*     contained herein.                                      *
*                                                            *
**************************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
**************************************************************
DATA DIVISION.
**************************************************************
WORKING-STORAGE SECTION.
*
* Return status to pass status to ACMS
*
01 RET-STAT     PIC S9(9) COMP.
*
* Task name
*
01 TASK         PIC X(24) VALUE IS "VR_COMP_FAST_CHKIN_TASK".
*
* Application Name
*
01 APPL         PIC X(7) VALUE IS "VR_APPL".
*
*
* Queue Name
*
* Queue name cannot contain trailing spaces.
*
01 QNAM         PIC X(8) VALUE IS "VR_QUEUE".
*
* Flag - 0 = NOHOLD - the queued task is free to run
*        1 = HOLD   - the queued task is stored but may not run
*
01 FLAG         PIC 9(9)        COMP VALUE IS 0.
*
* A list of workspace descriptors is sent to the queue service.
* For this task, only 1 workspace is required (VR_FAST_CHECKIN_WKSP).
* 1 is the number of workspaces in the list, QTE_WS_DESC is the
* descriptor being sent, so the address of the address of
* QTE_WS_DESC is stored.
*
01 WS_DESC_LIST.
   05 DES_LIST_CNT      PIC S9(9) COMP VALUE 1.
   05 QTE_WS_DESC_PTR POINTER VALUE IS REFERENCE QTE_WS_DESC.
*
* Each workspace must be described by a desriptor. 16 is the number
* of bytes in the workspace VR_FAST_CHECKIN_WKSP so its address is
* stored in the descriptor.
*
01 QTE_WS_DESC.
   05 FAST_CHECKIN_WS_SIZE PIC S9(9) COMP VALUE IS 16.
   05 FAST_CHECKIN_PTR     USAGE IS POINTER.

**************************************************************
```

```
LINKAGE SECTION.
*
COPY "VR_FAST_CHECKIN_WKSP" FROM DICTIONARY.
*
*****************************************************************
PROCEDURE DIVISION USING VR_FAST_CHECKIN_WKSP
                   GIVING RET-STAT.
*****************************************************************
MAIN-SECTION.

        SET RET-STAT TO SUCCESS.

        SET FAST_CHECKIN_PTR TO REFERENCE OF VR_FAST_CHECKIN_WKSP.

        CALL "ACMS$QUEUE_TASK"  USING BY DESCRIPTOR QNAM,
                                      BY DESCRIPTOR TASK,
                                      BY DESCRIPTOR APPL,
                                      BY REFERENCE  WS_DESC_LIST,
                                      BY REFERENCE  FLAG
        GIVING RET-STAT.

EXIT-PROGRAM.
        EXIT PROGRAM.
```

Example 9.8, "VR_COMP_FAST_CHKIN_TASK Definition" shows the definition of the VR_COMP_FAST_CHKIN_TASK.

Example 9.8. VR_COMP_FAST_CHKIN_TASK Definition

```
!
! This task is queued by the VR_FAST_CHECKIN_TASK which in turn
! is called by an ACMS agent.  The agent obtains information from
! the user and sends it to the VR_FAST_CHECKIN_TASK.  The
! VR_FAST_CHECKIN_TASK passes the information to the task.
!
REPLACE TASK  AVERTZ_CDD_TASK:VR_COMP_FAST_CHKIN_TASK

USE WORKSPACES  VR_FAST_CHECKIN_WKSP,
                VR_RESERVATIONS_WKSP,
                VR_RENTAL_CLASSES_WKSP,
                VR_VEHICLES_WKSP,
                VR_TRANS_WKSP,
                VR_VEHICLE_RENTAL_HISTORY_WKSP,
                VR_HIST_WKSP,
                VR_CONTROL_WKSP;

TASK ARGUMENTS ARE VR_FAST_CHECKIN_WKSP WITH ACCESS READ;

! QTI STARTS A DISTRIBUTED TRANSACTION BEFORE BEGINNING THIS TASK.
! THIS TASK JOINS THE DISTRIBUTED TRANSACTION. SHOULD THE TASK FAIL
! FOR ANY REASON, THE QTI ABORTS THE DISTRIBUTED TRANSACTION AND STARTS
! A NEW DISTRIBUTED TRANSACTION SO THAT IT CAN TRANSFER THE FAILED
! QUEUED TASK RECORD ONTO AN ERROR QUEUE, IF ONE EXISTS.

BLOCK WITH TRANSACTION
                NO I/O
!
! FIND THE RESERVATION RECORD.
```

```
!
        PROCESSING WITH NONPARTICIPATING SERVER
                CALL PROCEDURE  VR_FIND_RES_PROC
                  IN    VR_READ_SERVER
                USING   VR_FAST_CHECKIN_WKSP,
                        VR_RESERVATIONS_WKSP;

        ACTION IS
            IF (ACMS$T_STATUS_TYPE = "G")
            THEN
             MOVE VR_FAST_CHECKIN_WKSP.ACTUAL_RETURN_DATE
                   TO VR_VEHICLE_RENTAL_HISTORY_WKSP.ACTUAL_RETURN_DATE,
                  VR_FAST_CHECKIN_WKSP.RETURN_ODOMETER_READING
                   TO
               VR_VEHICLE_RENTAL_HISTORY_WKSP.RETURN_ODOMETER_READING;

            ELSE
                CANCEL TASK RETURNING ACMS$L_STATUS;
            END IF;
!
! RETRIEVE THE VEHICLE AND VEHICLE_RENTAL_HISTORY RECORDS.
!
        PROCESSING WITH NONPARTICIPATING SERVER
                CALL PROCEDURE  VR_FIND_VE_VRH_PROC
                  IN    VR_READ_SERVER
                USING   VR_RESERVATIONS_WKSP,
                        VR_VEHICLES_WKSP,
                        VR_VEHICLE_RENTAL_HISTORY_WKSP,
                        VR_RENTAL_CLASSES_WKSP,
                        VR_TRANS_WKSP;

        ACTION IS
            IF (ACMS$T_STATUS_TYPE = "B")
            THEN
                CANCEL TASK RETURNING ACMS$L_STATUS;
            END IF;
!
! COMPUTE THE BILL.
!
        PROCESSING WITH NONPARTICIPATING SERVER
                CALL PROCEDURE  VR_COMPUTE_BILL_PROC
                  IN    VR_READ_SERVER
                USING   VR_RESERVATIONS_WKSP,
                        VR_RENTAL_CLASSES_WKSP;

        ACTION IS
            IF (ACMS$T_STATUS_TYPE = "B")
            THEN
                CANCEL TASK RETURNING ACMS$L_STATUS;
            END IF;

!
! COMPLETE THE CHECKIN PROCESS.
!

        PROCESSING
                CALL PROCEDURE  VR_COMPLETE_CHECKIN_PROC
                  IN    VR_UPDATE_SERVER
```

```
                USING   VR_RESERVATIONS_WKSP,
                        VR_VEHICLES_WKSP,
                        VR_VEHICLE_RENTAL_HISTORY_WKSP,
                        VR_CONTROL_WKSP;

        ACTION IS
                MOVE    "FSTCHKIN"
                                    TO  VR_HIST_WKSP.TRANS_TYPE,
                         VR_VEHICLES_WKSP.VEHICLE_ID
                                    TO  VR_HIST_WKSP.VEHICLE_ID,
                        VR_RESERVATIONS_WKSP.RENTAL_TOTAL_AMT
                                    TO  VR_HIST_WKSP.RENTAL_TOTAL_AMT;


        PROCESSING
                CALL PROCEDURE  VR_WRITE_HIST_RECORD_PROC
                  IN    VR_LOG_SERVER
                USING   VR_HIST_WKSP,
                        VR_RESERVATIONS_WKSP;

END BLOCK;

END DEFINITION;
```

The task queue VR_QUEUE is marked for RMS recovery unit journaling. Therefore, the QTI starts a distributed transaction before submitting the VR_COMP_FAST_CHKIN_TASK for processing. To join the distributed transaction, the task specifies TRANSACTION on the root block step.

The first two processing steps use the reservation ID, obtained by the agent, to retrieve reservation and car records. The third processing step computes the bill. Because these three processing steps are not writing to the database, they use the NONPARTICIPATING SERVER phrase to exclude themselves from the distributed transaction.

9.11. Procedure Parameter Notation for Programming Services

ACMS$DEQUEUE_TASK and *ACMS$QUEUE_TASK* contain reference material for the ACMS$DEQUEUE_TASK and ACMS$QUEUE_TASK programming services. The format descriptions for the services use OpenVMS procedure parameter notation. Each parameter can have four characteristics, represented by two groups of symbols following the parameter. The characteristics definable for each parameter are:

```
<name>.<access type><data type>.<pass mech><parameter form>
```

The characteristics are always listed in this order. A period (.) separates access and data types from passing mechanism and parameter form. For example:

```
comp_status.wq.r
```

Table 9.5, "Procedure Parameter Notation" defines the symbols used for procedure parameter notation.

Table 9.5. Procedure Parameter Notation

Notation	Symbol	Meaning
Access Type	m	Modify access

Notation	Symbol	Meaning
	r	Read access only
	s	Call without stack unwinding
	w	Write and read access
Data Type	adt	Absolute date and time
	bu	Byte logical (unsigned)
	l	Longword integer (signed)
	lc	Longword return status
	lu	Longword logical (unsigned)
	q	Quadword integer (signed)
	qu	Quadword integer (unsigned)
	r	Record
	t	Character-coded text string
	w	Word integer (signed)
	x	Data type by descriptor
	z	Unspecified
	zem	Procedure entry mask
Passing Mechanism	d	By descriptor
	r	By reference
	v	By immediate value
Parameter Form	none	Scalar (also called atomic data type)
	x	Class type by descriptor

For a complete explanation of all the OpenVMS data structures, data types, access mechanisms and passing mechanisms, see *Guide to Creating OpenVMS Modular Procedures*.

ACMS$DEQUEUE_TASK

ACMS$DEQUEUE_TASK — Removes or reads a queued task element from the queued task file and returns information about the task. This service does not dequeue or read queued task elements that are on hold unless you include the element ID in the parameter list. ACMS$DEQUEUE_TASK deletes a queue element immediately upon removing it from the queue (unless READ ONLY is specified).

Format

```
ACMS$DEQUEUE_TASK
(queue_name.rt.dx,
[element_id.rr.r],
[flags.rlu.r],
[ret_task.wt.dx],
[ret_application.wt.dx],
[ret_workspace_list.wz.r],
[ret_workspace_count.wl.r],
[ret_element_priority.wl.r],
[ret_username.wt.dx],
[ret_element_id.wr.r],
```

```
[ret_error_count.wlu.r],
[ret_last_error.wlu.r],
[ret_last_error_adt.wadt.r])
```

Parameters

queue_name

The name of the queue from which to dequeue the task. The name is subject to logical name translation in the calling process. The logical translation is performed only the first time that the queue is accessed by the calling process. Any queue names resulting from logical translation can have a maximum length of 39 characters. Logical search lists are not supported.

The queue name parameter is case-sensitive, so this parameter must be uppercase in order to match the queue name created with ACMSQUEMGR. The ACMSQUEMGR uppercases all queue names unless you explicitly enclose the queue name in quotes when using the **CREATE QUEUE** command.

The queue name cannot contain any trailing spaces. For example, if the queue name is "UPDATE_QUEUE" then the descriptor must have a length of 12.

Always create the queue before using this service. Create a task queue by using the ACMS Queue Management Utility (ACMSQUEMGR).

Dequeue tasks in one of three access modes:

- By highest element priority (first-in-first-out, FIFO, within priority). This is the default mode for this service.
- By element ID. By supplying the element_ID parameter, you can directly access a queued task element. When you enqueue a task, the ACMS$QUEUE_TASK service returns the element ID.
- Sequentially, by specifying the ACMS$M_QUE_GETNEXT flag.

element_id

The queued task element ID in binary format. The parameter allows direct access to a queued task element. The ACMS$QUEUE_TASK service returns the element ID (also in binary format). When you specify the element ID, the queued task element is dequeued regardless of whether or not it is on hold.

The binary queued task element ID is 32 bytes long and has the following format:

Field Name	Length
Process ID (PID)	Longword
Sequence number	Longword
Enqueue absolute date/time	Quadword
Node name length	Byte
Node name	15 bytes

When the ACMSQUEMGR utility displays or accepts the queued task element ID, it uses the following display format:

```
node_name::pid-seq_no-adt1-adt2
```

In this format:

- `node_name` is the node name.
- `PID` is the process ID formatted as 8 hex characters.
- `seq_no` is the sequence number formatted as 8 hex characters.
- `adt1` is the low order longword of the enqueue absolute date/time formatted as 8 hex characters.
- `adt2` is the high-order longword of the enqueue absolute date/time formatted as 8 hex characters.

flags

Flags specifying options for the ACMS$DEQUEUE_TASK operation. The flags argument is a longword bit mask that is the logical OR of each bit set, where each bit corresponds to an option.

- ACMS$M_QUE_READONLY

  When this bit is set, the ACMS$DEQUEUE_TASK service reads a queued task element but does not delete the queue element from the task queue file. The service performs the read operation regardless of whether or not dequeues have been suspended for the queue.

  If the ACMS$M_QUE_READONLY bit is set and you use the default access mode (FIFO within priority), the ACMS$DEQUEUE_TASK service returns the same queued task element each time you call the service (assuming a static queue file).

  By default this bit is clear.

- ACMS$M_QUE_NOWAIT

  When this bit is set, the ACMS$DEQUEUE_TASK service does not wait for a queued task element before returning. If there are no elements and this bit is set, then the error ACMS$_QUEEMPTY is returned.

  When this bit is not set, the ACMS$DEQUEUE_TASK service waits for a queue element if no elements are currently queued.

  By default this bit is clear.

- ACMS$M_QUE_GETNEXT

  When this bit is set, the ACMS$DEQUEUE_TASK service operates in sequential mode.

  Before you set this bit, you can establish a position in the queue file by using a prior ACMS$DEQUEUE_TASK service call in either of the other two access modes. If you have not established a position in the queue file by using a prior ACMS$DEQUEUE_TASK service call, then by default you are positioned to the top of the queue file.

  If you set the ACMS$M_QUE_GETNEXT bit after initially calling the ACMS$DEQUEUE_TASK service in the default access mode, position the internal file pointer to the highest priority element. Then the queue file will be read sequentially on successive calls to the ACMS$DEQUEUE_TASK service.

Because of the queue file organization, this sequential access returns queued task elements in the same order that they would be returned if you were using the default access mode without setting the ACMS$M_QUE_READONLY bit (assuming a static queue file). The ordering of the queue file is by highest priority, and FIFO within priority.

By default, this bit is clear.

ret_task

The task name for this queued task element.

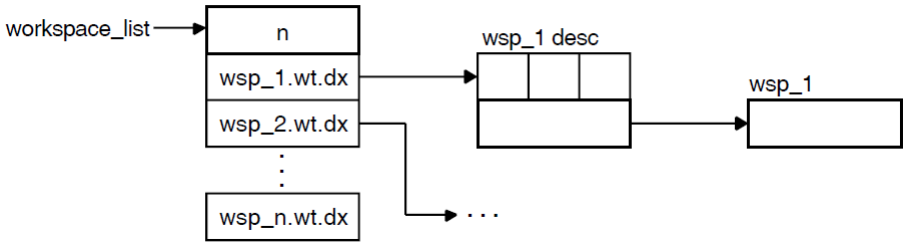
ret_application

The application specification for this queued task element.

ret_workspace_list

The list of workspaces for this queued task element. This is a varying length argument list of workspaces passed by descriptor, as shown in *Figure 9.3, "List of Workspaces Passed by ACMS $DEQUEUE_TASK"*.

Figure 9.3. List of Workspaces Passed by ACMS$DEQUEUE_TASK



If the queued task element has more workspaces than are passed in ret_workspace_list, then the warning error message ACMS$_TOOFEWWSPS is returned and the element is deleted. In all cases, the actual number of workspaces contained in the queued task element is returned in the ret_workspace_count parameter.

ret_workspace_count

The total number of workspaces for this queued task element.

ret_element_priority

The priority of this queued task element [0 through 255].

ret_username

The user name of the process that enqueued the queued task element, or, if the enqueuing process had CMKRNL privilege, this could be the user name specified with the user name parameter on the ACMS$QUEUE_TASK call.

ret_element_id

The queued task element ID for this queued task element.

ret_error_count

The number of times (if any) that the QTI component attempted a task invocation which failed for this queued task element.

ret_last_error

The error message code (if any) returned to the QTI due to a task invocation error. If the QTI retries a task invocation and the retry also fails, then this parameter contains the most recent failure status. If the queued task element has never failed on a task invocation, then the error code is ACMS$_NORMAL.

ret_last_error_adt

The absolute date and time of the most recent error message returned to QTI due to a task invocation error. If the queued task element has never failed on a task invocation, then the absolute date and time is zero.

Return Status

The error messages that can be returned by the ACMS$DEQUEUE_TASK service include:

Status	Severity Level	Description
ACMS$_NORMAL	Success	Normal successful completion
ACMS$_TOOFEWWSPS	Warning	The workspace list parameter had fewer workspaces than the queued task
ACMS$_ERROPNAQF	Error	Error opening the ACMS Queue Definition File
ACMS$_ERROPNPAR	Error	Error opening the ACMS parameter file
ACMS$_ERROPNQUE	Error	Error opening queue
ACMS$_ERRQUEINIT	Error	Error initializing the queue services
ACMS$_INSUFPRIV	Error	Insufficient privilege for attempted operation
ACMS$_INVNUMWSP	Error	Invalid number of workspaces
ACMS$_INVQUENAM	Error	Invalid queue name
ACMS$_LNE	Error	Logical name translation count exceeded
ACMS$_QUEDEQSUS	Error	Dequeue operations are suspended
ACMS$_QUEEMPTY	Error	Queue is empty
ACMS$_QUENOTFND	Error	Queue does not exist
SS$_ACCVIO	Fatal	Access violation; an argument that you passed was not accessible

Errors from the RMS services SYS$OPEN, SYS$CONNECT, SYS$GET, and the RTL service LIB$SCOPY.dx.dx can also be returned.

Note

You can include the ACMS$DEQUEUE_TASK call within a distributed transaction. When ACMS calls the DECdtm to start the distributed transaction, DECdtm returns a unique transaction identifier (TID). ACMS$DEQUEUE_TASK uses this default TID.

ACMS$QUEUE_TASK

ACMS$QUEUE_TASK — Stores the queued task element in an on-disk queued task file.

Format

ACMS$QUEUE_TASK

```
(queue_name.rt.dx,
 task.rt.dx,
 application.rt.dx,
 [workspace_list.rz.r],
 [flags.rlu.r],
 [element_priority.rl.r],
 [username.rt.dx],
 [element_id.wr.r])
```

Parameters

queue_name

The name of the queue in which you want to store this task. The name is subject to logical name translation in the calling process. The logical is performed only the first time that the queue is accessed by the calling process. The queue_name and any queue names resulting from logical translation can have a maximum length of 39 characters. Using logical names for queue names allows the actual queue name to change without recoding programs. Logical search lists are not supported.

The queue name parameter is case-sensitive, so this parameter must be uppercase in order to match the queue name created with ACMSQUEMGR. The ACMSQUEMGR uppercases all queue names unless you explicitly enclose the queue-name in quotes when using the **CREATE QUEUE** command.

The queue name cannot contain any trailing spaces. For example, if the queue name is "UPDATE_QUEUE" then the descriptor must have a length of 12.

Always create the queue before using this service. Create a task queue by using the ACMS Queue Management Utility (ACMSQUEMGR).

task

The name of the task queued. The task name must be in capital letters.

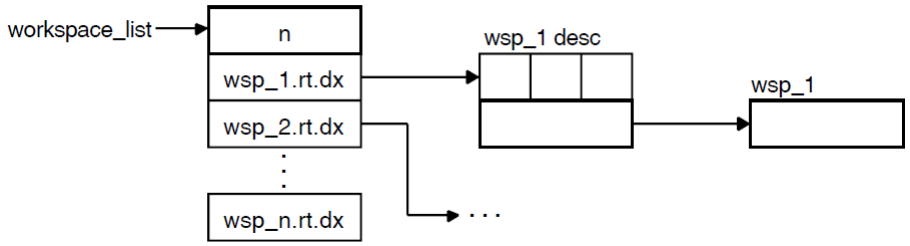
application

The name of the application in which the task is to be run. The application name must be in capital letters. This parameter names an application specification. The semantics associated with application specifications (for example, logical translation, search lists, failover) occur as the task is invoked by the QTI. See *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/] for more information on application specifications.

workspace_list

A list of workspaces to pass to the task. This is a variable length argument list of workspaces passed by descriptor, as shown in *Figure 9.4, "List of Workspaces Passed by ACMS$QUEUE_TASK Service"*.

Figure 9.4. List of Workspaces Passed by ACMS$QUEUE_TASK Service



flags

Flags specifying options for the ACMS$QUEUE_TASK operation. The flags argument is a longword bit mask that is the logical OR of each bit set, where each bit corresponds to an option.

ACMS$M_QUE_HOLD

Setting the bit puts the queued task element in the hold state. Queued task elements that are in the hold state are not available for dequeuing by either the QTI or the ACMS$DEQUEUE_TASK service. Once a task is queued, you can use the ACMSQUEMGR utility to modify its state.

By default, this bit is clear.

element_priority

The relative priority of the queue element: higher priority elements are dequeued before lower priority elements. Valid values are 0 through 255.

The default value is 10.

username

The user name under which the task runs. To specify this parameter, you must have the OpenVMS CMKRNL privilege. If you specify the parameter without this privilege, the ACMS$QUEUE_TASK service fails and returns ACMS$_INSUFPRIV.

If you do not use this parameter, the user name of the enqueuing process is stored as the user name under which the task runs.

The QTI uses the enqueuer user name as the submitter user name for the task selection (provided the process user name of the QTI has the ACMS agent privilege).

element_id

A returned queued task element ID (in binary format) for this queued task element. See the ACMS$DEQUEUE_TASK service for a description of the binary and display formats of the element ID.

You can use the element ID from the ACMSQUEMGR Utility and in the ACMS$DEQUEUE_TASK service to directly access this queued task element for element operations.

Return Status

The error messages returned by the ACMS$QUEUE_TASK service include:

Status	Severity Level	Description
ACMS$_NORMAL	Success	Normal successful completion
ACMS$_ERROPNAQF	Error	Error opening the ACMS Queue Definition File
ACMS$_ERROPNPAR	Error	Error opening the ACMS parameter file
ACMS$_ERROPNQUE	Error	Error opening queue
ACMS$_ERRQUEINIT	Error	Error initializing the queue services
ACMS$_INSUFPRIV	Error	Insufficient privilege for attempted operation
ACMS$_INVAPPLE	Error	The application specification or its descriptor was invalid
ACMS$_INVNUMWSP	Error	Invalid number of workspaces
ACMS$_INVPRIO	Error	Invalid priority for element
ACMS$_INVQUENAM	Error	Invalid queue name
ACMS$_INVSIZWSP	Error	Total size of workspaces exceeds the size defined in the ACMS Queue Definition File
ACMS$_INVTASK	Error	The task name or its descriptor was invalid
ACMS$_INVUSER	Error	The user name was invalid
ACMS$_LNE	Error	Logical name translation count exceeded
ACMS$_QUEENQSUS	Error	Enqueue operations are suspended
ACMS$_QUENOTFND	Error	Queue does not exist
SS$_ACCVIO	Fatal	Access violation; an argument that you passed was not accessible

Errors from the RMS services SYS$OPEN, SYS$CONNECT, and SYS$PUT can also be returned.

Note

You can include the ACMS$QUEUE_TASK call within a distributed transaction. When ACMS calls the DECdtm to start the distributed transaction, DECdtm returns a unique transaction identifier (TID). ACMS$QUEUE_TASK uses this default TID.

Chapter 10. Defining Task Groups

Once you create one or more task definitions, you can take the two final steps to prepare the definitions for debugging:

- Define a task group that sets up characteristics for the task or tasks.
- Use the **BUILD** command to produce object modules for the procedure servers in the task group and to produce information ACMS uses to run the tasks in that group.

This chapter explains these steps.

10.1. Defining a Task Group

Often you have a set of tasks that have common characteristics or require the same resources. You group these tasks in task groups. **Task groups** let you set up resources needed by a number of tasks, such as the servers those tasks use.

Just as the task part of a task definition sets up characteristics for the steps in the task, a task group definition sets up characteristics for the tasks in that group.

There are several characteristics you can define for a task group:

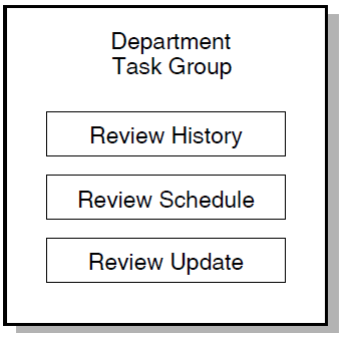
- Task or tasks belonging to the group
- Server or servers that do work for those tasks
- Request libraries and message files used by tasks in the group
- Workspaces used by tasks in the task group
- Name of the task group database file

Of these characteristics, you must always define the first two: the tasks that belong to the group and the server or servers that do work for those tasks. The third characteristic is required only if you use requests or the GET ERROR MESSAGE clause in any of the tasks in the task group. Declaring workspaces used by tasks in the task is optional; the task database file name is optional.

10.2. Identifying Which Tasks Belong to the Task Group

In a task group definition you must identify the tasks belonging to that group. For example, in *Figure 10.1, "A Task Group and Its Tasks"*, the Department task group includes three tasks.

Figure 10.1. A Task Group and Its Tasks



You name the tasks in the definition for the Department task group as follows:

```
TASKS ARE
  REVIEW_HISTORY  :  TASK IS REVIEW_HISTORY_TASK;
  REVIEW_SCHEDULE :  TASK IS REVIEW_SCHEDULE_TASK;
  REVIEW_UPDATE   :  TASK IS REVIEW_UPDATE_TASK;
END TASKS
```

When you name a task in a task group definition:

- Assign a name, such as REVIEW_HISTORY, to the task. The name, an alphanumeric string of 1 to 31 characters, can contain dollar signs ($) and underscores but no embedded blanks. The name must begin with a letter, dollar sign ($), or underscore (_).
- Use the CDD path name to indicate the location of the task definition in the CDD.

Use a colon (:) to separate the task name and the path name. Unless you set your CDD default to the correct directory when you build the definition, you must use the full CDD path name of each task you define. If you have set your CDD default, you can use just the given name of each task, as in the previous example, or a partial CDD path name.

You can also include single-step task definitions directly in a task group definition rather than creating a separate definition for this task and referencing it by a CDD path name in the task group definition:

```
TASKS ARE
  EDITOR  : DELAY;
            PROCESSING IS
              DCL COMMAND IS "$EDIT/EDT 'P1'" IN PRIVATE_UTILITY_SERVER;
```

See *Chapter 13, "Defining Existing Applications as ACMS Tasks"* for an explanation of single-step tasks.

10.3. Identifying Which Servers Are Required in the Group

When you write a definition for a multiple-step task, you name the server or servers required to handle the processing work for that task. Many tasks can use the same server. DCL servers handle DCL commands or procedures, DATATRIEVE commands or procedures, and OpenVMS images. Procedure servers handle calls to procedures or subroutines.

There are two characteristics you must define for each server you name in a task group:

- Server type

- Procedures handled by that server, if it is a procedure server

The server type can be either DCL PROCESS or PROCEDURE SERVER. The procedures you name for a procedure server are the procedures named in the processing steps of tasks using that server.

The Department task group uses a single procedure server.

```
SERVER IS
  DEPARTMENT_SERVER:
    PROCEDURE SERVER IMAGE IS "ACMS$EXAMPLES:DEPRMSCOB.EXE";



END SERVER;
```

When you use the PROCEDURE SERVER clause to identify the server type, you must also name the file specification of the image for the server you are describing. The procedure server image is the runnable collection of all the procedures handled by a server. It is the result of linking:

- Object module for the server created when building the task group
- Object modules created by compiling the step procedures handled by the server
- Object modules created by compiling the cancel, initialization, and termination procedures for the server
- Object modules for the messages used by the server

For example, the file DEPRMSCOB.EXE is the result of linking:

1. Object module for DEPARTMENT_SERVER, created when building the Department task group
2. Object modules for the procedures used in the Review History, Review Schedule, and Review Update tasks
3. Object modules for the cancel, initialization, and termination procedures for DEPARTMENT_SERVER
4. Object module for the DEPARTMSG messages

If you define the server type to be PROCEDURE SERVER, you must name the step procedures handled by a server. For example:

```
SERVER IS
  DEPARTMENT_SERVER:
    PROCEDURE SERVER IMAGE IS "ACMS$EXAMPLES:DEPRMSCOB.EXE";
    PROCEDURES ARE
      REVIEW_HISTORY_GET, REVIEW_SCHEDULE_GET,
      REVIEW_UPDATE_GET, REVIEW_UPDATE_PUT;



END SERVER;
```

These are the procedure names the Review History, Review Schedule, and Review Update task definitions use in their processing steps. The names are the entry points for these procedures in the procedure server image. In the PROCEDURES clause, you can also name initialization, termination, and cancel procedures; however, only the names of step procedures are required.

There are additional characteristics you can describe for a procedure server:

- Procedures that run when the server processes start and stop
- Cancel procedure used by the server
- File name of the server object module produced when you build the task group

Tasks using the same procedure server usually require one or more of the same files. For example, the Review History, Review Schedule, and Review Update tasks all use the Personnel and History files. Rather than opening and closing files for each task, you can name, for a server, procedures that open and close files when that server starts and stops. These procedures are called **initialization** and **termination** procedures. You name each procedure as part of the server definition:

```
INITIALIZATION PROCEDURE IS DEPART_STARTUP;
TERMINATION PROCEDURE IS DEPART_SHUTDOWN;
```

You identify an initialization or termination procedure by its entry point in the procedure server image. DEPART_STARTUP is the entry point of the initialization procedure in the image for the PERSONNEL procedure server. If you are using a procedure written in COBOL, it begins with an IDENTIFICATION DIVISION, which defines the entry point as the PROGRAM-ID:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DEPART_STARTUP.
```

If you use BASIC, the entry point is the function name of the initialization or termination procedure. You do not enclose the procedure name within quotation marks.

A server definition can also name a cancel procedure. This procedure usually releases record locks and performs other cleanup work for the server. If a task is canceled either during a processing step or while ACMS is retaining process context in a task, ACMS runs the cancel procedure named for the server being used by the task when the cancel occurs.

```
CANCEL PROCEDURE IS DEPART_CANCEL;
```

As with initialization and termination procedures, the name of the cancel procedure is its entry point in the procedure server image. Do not enclose the procedure name within quotation marks.

When you build a task group, ACMS creates an object file, called the **procedure server transfer module**, for each procedure server named in the definition for that group. You can use the DEFAULT OBJECT FILE clause to define, for a server, the file specification you want ACMS to use for the object file it creates. For example:

```
DEFAULT OBJECT FILE IS "ACMS$EXAMPLES:DEPRMSCOB.OBJ";
```

You can also use logical names to name the location of the default object file. Enclose the file specification within quotation marks. If you do not name an object file, ACMS derives the name of the file from the the full given name you assign to the server in the task group definition, including dollar signs and underscores.

Here is the SERVER IS clause for the Department task group:

```
SERVER IS
  DEPARTMENT_SERVER:
    PROCEDURE SERVER IMAGE IS "ACMS$EXAMPLES:DEPRMSCOB.EXE";
    PROCEDURES ARE
       REVIEW_HISTORY_GET, REVIEW_SCHEDULE_GET,
       REVIEW_UPDATE_GET, REVIEW_UPDATE_PUT;
```

```
    INITIALIZATION PROCEDURE IS DEPART_STARTUP;
    TERMINATION PROCEDURE IS DEPART_SHUTDOWN;
    CANCEL PROCEDURE IS DEPART_CANCEL;
    DEFAULT OBJECT FILE IS "ACMS$EXAMPLES:DEPRMSCOB.OBJ";
END SERVER;
```

10.3.1. Assigning Server Attributes

In addition to naming the procedures, procedure server image, and default object file, you can use the server definition to specify attributes that affect the modularity and, possibly, the performance of your application. This section describes two such attributes, the RUNDOWN ON CANCEL IF INTERRUPTED and ALWAYS EXECUTE TERMINATION PROCEDURE server subclauses.

By default, ACMS processes a server's termination procedure when the server process is run down, unless the server process is being run down because the task was canceled. There might be times when you want to override this default by specifying ALWAYS EXECUTE TERMINATION PROCEDURE. For example, a server that uses global sections might need to clean up information in the global sections when the server process is run down. The ALWAYS EXECUTE TERMINATION PROCEDURE subclause instructs ACMS to process the server's termination procedure whenever the server process is run down.

By default, ACMS runs down a server process when the task is canceled while the task is keeping context in that server. When the server exits, ACMS releases server context. Preventing unnecessary process deletions and creations can improve the performance of your application. You can use the RUNDOWN ON CANCEL IF INTERRUPTED server subclause to instruct ACMS to run down the server process only if ACMS interrupts the execution of a step procedure due to an exception. ACMS does not run down the server process if, for example, the task is simply retaining context when you specify RUNDOWN ON CANCEL IF INTERRUPTED.

Example 10.1, "Definition of VR_READ_SERVER" shows a server definition that specifies ALWAYS EXECUTE TERMINATION PROCEDURE and RUNDOWN ON CANCEL IF INTERRUPTED.

Example 10.1. Definition of VR_READ_SERVER

```
SERVERS ARE
    VR_READ_SERVER:
      PROCEDURE SERVER IMAGE IS "AVERTZ_DEFAULT:VR_READ_SERVER.EXE";
      INITIALIZATION PROCEDURE IS VR_READ_INIT;
      TERMINATION PROCEDURE IS VR_TERM;
      ALWAYS EXECUTE TERMINATION PROCEDURE ON RUNDOWN;
      RUNDOWN ON CANCEL IF INTERRUPTED;
      PROCEDURES ARE
                        VR_COMPUTE_BILL_PROC,
                        VR_FIND_CU_PROC,
                        VR_FIND_SI_PROC,
                        VR_FIND_VE_VRH_PROC,
                        VR_FIND_RES_PROC,
                        VR_RES_DETAILS_PROC;
      DEFAULT OBJECT FILE IS "AVERTZ_DEFAULT:VR_READ_SERVER.OBJ";
```

Once you have defined the tasks in a task group and the servers used by those tasks, you can consider other characteristics of the task group, such as the request libraries, message files, and workspaces used by the tasks in the group.

10.4. Naming Request Libraries

When you define tasks, you can name TDMS requests in the exchange steps. For ACMS to process a request, it must know in which request library to find it. This library stores the definitions for all the requests that you use for tasks in the group. For example, all the requests you use in the exchange steps of tasks in the Department task group are in the same request library.

For each task group definition, you name all the request libraries used by tasks in the task group:

```
REQUEST LIBRARY IS "ACMS$EXAMPLES_DEPRMSRLB";
```

In this example, ACMS$EXAMPLES_DEPRMSRLB is the name of the request library. This name is a logical name pointing to the location of the request library file. The default file type is .RLB. If you do not name a directory, ACMS uses the directory named by the DEFAULT DIRECTORY clause in the application definition. You must enclose the file name of the request library within quotation marks if there are characters in that name that are invalid for an identifier (such as periods, colons, or semicolons), or if the name is longer than 31 characters.

If you name just one request library, ACMS uses that request library for all the tasks in the group. If different tasks in the group use different libraries, or if different steps in a single task use different libraries, you must name all these request libraries in the task group definition. In this case, you must also assign a name to each request library you name. For example:

```
REQUEST LIBRARY IS "ACMS$EXAMPLES_DEPRMSRLB" WITH NAME DEPARTREQ;
```

The name you assign to a request library is an identifier containing 1 to 31 characters. When writing a task definition, you use this name in the DEFAULT REQUEST LIBRARY clause or with the IN keyword of the REQUEST clause to identify the request library used by the task or exchange step. For example:

```
EXCHANGE
   REQUEST IS REVIEW_HISTORY_INPUT_REQUEST
     IN DEPARTREQ USING REVIEW_HISTORY_WORKSPACE;
```

If you do not name the request library with the IN keyword of the REQUEST clause of an exchange step, and you do not use the DEFAULT REQUEST LIBRARY clause in the definition of the task containing that step, ACMS uses the first request library named in the task group containing the task.

When defining a task group whose tasks call only user request procedures (URPs) , you can use the REQUEST LIBRARY IS clause to name the shareable image containing URP procedures. The shareable image must have a .EXE extension. You may specify a complete file specification or a logical name in the REQUEST LIBRARY IS clause. If you use a logical name (the recommended method), be sure to define the logical name to translate to a full file specification with a .EXE extension. If the logical name does not contain the .EXE extension, the EXC uses the .RLB default when it tries to open all the files declared in the REQUEST LIBRARY IS clauses. This action causes the **ACMS/START APPLICATION** to fail, unless ACMS finds a TDMS (.RLB) file of the same name.

For information on building request libraries, see *VAX TDMS Request and Programming Manual*.

10.5. Identifying Which Message Files Are Used in the Group

Just as you declare the request libraries containing requests used by tasks in a task group, you must name the message files used by the tasks in a group. You use the MESSAGE FILES clause as follows:

```
MESSAGE FILE IS "ACMS$EXAMPLES:DEPARTMSG.EXE";
```

When you use the MESSAGE FILES clause, you use the name of the image (.EXE) version of the message file. In this example, the message file used by the Department task group is the file ADMINMSG.EXE in the directory with the logical name ACMS$EXAMPLES. If you do not name a directory, ACMS uses the directory named by the DEFAULT DIRECTORY clause of the application definition.

For information on creating message files, see *VSI ACMS for OpenVMS Writing Server Procedures* [https://docs.vmssoftware.com/vsi-acms-writing-server-proc/].

10.6. Naming Workspaces in a Task Group Definition

Often workspaces are used by more than one task in a task group. You can name these workspaces in the task group definition. You use the USE WORKSPACES clause in a task definition that uses workspaces named in the task group. For example, suppose that you declare DEPT_WORKSPACE in the definition for the Department task group:

```
WORKSPACE IS DEPT_WORKSPACE WITH TYPE USER;
```

The Review History task uses the USE WORKSPACE clause to refer to the DEPT_WORKSPACE named in the task group definition:

```
USE WORKSPACE DEPT_WORKSPACE;
```

When you use the USE WORKSPACE clause, you can change the access restrictions that are defined for the workspace in the task group definition. You cannot change the workspace type.

10.7. Naming the Task Database for a Task Group

When you build a task group, ACMS creates a task database for that task group. ACMS uses the database at run time to get information about processing tasks selected by a user. You can use the DEFAULT TASK GROUP FILE clause to assign the file specification you want ACMS to use when creating the task database. For example:

```
DEFAULT TASK GROUP FILE IS "ACMS$EXAMPLES:DEPRMSCOB.TDB";
```

If you do not use a file specification with the **BUILD** command, ACMS uses the file specification, if any, you named in the DEFAULT TASK GROUP FILE clause. If you do not use the DEFAULT TASK GROUP FILE clause, ACMS derives the file specification from the full given name of the task group, including dollar signs and underscores.

Example 10.2, "Definition of Department Task Group" shows the complete definition for the Department task group.

Example 10.2. Definition of Department Task Group

```
REPLACE GROUP DEPART_TASK_GROUP/LIST=DEPGRP.LIS

  REQUEST LIBRARY IS "ACMS$EXAMPLES_DEPRMSRLB";
```

```
  MESSAGE FILE IS "ACMS$EXAMPLES:DEPARTMSG.EXE";
  DEFAULT TASK GROUP FILE IS "ACMS$EXAMPLES:DEPRMSCOB.TDB";
  TASKS ARE
    REVIEW_HISTORY : TASK IS REVIEW_HISTORY_TASK;
    REVIEW_SCHEDULE: TASK IS REVIEW_SCHEDULE_TASK;
    REVIEW_UPDATE  : TASK IS REVIEW_UPDATE_TASK;
  END TASKS;
  SERVER IS
    DEPARTMENT_SERVER:
      PROCEDURE SERVER IMAGE IS "ACMS$EXAMPLES:DEPRMSCOB.EXE";
      PROCEDURES ARE
        REVIEW_HISTORY_GET, REVIEW_SCHEDULE_GET,
        REVIEW_UPDATE_GET, REVIEW_UPDATE_PUT;
      INITIALIZATION PROCEDURE IS DEPART_STARTUP;
      TERMINATION PROCEDURE IS DEPART_SHUTDOWN;
      CANCEL PROCEDURE IS DEPART_CANCEL;
      DEFAULT OBJECT FILE IS "ACMS$EXAMPLES:DEPRMSCOB.OBJ";
   END SERVER;
END DEFINITION;
```

This definition does not include all the clauses you can use to define a task group. An additional characteristic you can define for a task group is a list of the workspaces used by tasks in that group. For a complete list and explanation of all the task group clauses, see the *VSI ACMS for OpenVMS ADU Reference Manual* [https://docs.vmssoftware.com/vsi-acms-adu-ref-manual/].

Once you have written a task group definition, you can store it in the CDD. If you include the **CREATE** or **REPLACE** command in the definition source file, you submit the file to the ADU as a command file:

```
ADU> @DEPARTGRP.COM
```

If you did not include the **REPLACE** command in the source definition file, you use the **REPLACE** command in response to the ADU> prompt. For example:

```
ADU> REPLACE GROUP DEPART_TASK_GROUP DEPART.GDF
```

This command stores the definition in the DEPART.GDF file in the CDD, using the name DEPART_TASK_GROUP. Common errors you might receive when you use the **CREATE** command are:

- The definition already exists. You can rename the group you have defined. Or you can use the **REPLACE** command to replace the existing definition with the new one.
- The CDD directory you named is invalid or access is not allowed to the directory. You may have used the wrong path name for the task group definition. Or you may need to change the access control list for the CDD directory.
- There are errors in the syntax, such as omitted semicolons (;). You may also have omitted required clauses or required keywords.

The **CREATE** command does not check whether the task group definition is valid or correct, such as whether the tasks named in the definition exist or whether file specifications are correct; it checks only for correct syntax. ACMS checks file specifications only at run time. However, references to CDD definitions are checked when you build the task group.

Once you have defined a task group and stored the definition in the CDD, you can go on to build the task group.

10.8. Changing Characteristics of Task Argument Workspaces

If you change the number of TASK ARGUMENT workspaces defined for a task or if you change the ACCESS method of a TASK ARGUMENT workspace, you must rebuild the application database (.ADB) after you rebuild the task group database (.TDB). If you forget to rebuild the .ADB and attempt to select a task that has been changed in this manner, ACMS cancels the task and writes the following message to the audit trail log:

```
************************************************************
Type   : TASK      Time   : 16-FEB-1988 15:24:48.45
Appl   : CTAPPL
Task   : PWT
User   : USER1
ID     : SAILNG::00010014-00000001-8A8853C0-0090E729
Sub    : SAILNG::00010014-00000000-8A8853C0-0090E729
Text   : Task start failed during initialization
Application must be rebuilt due to TASK ARGUMENT workspace changes
************************************************************
Type   : TASK      Time   : 16-FEB-1988 15:24:48.45
Appl   : CTAPPL
Task   : PWT
User   : USER1
ID     : SAILNG::00010014-00000001-8A8853C0-0090E729
Sub    : SAILNG::00010014-00000000-8A8853C0-0090E729
Text   : Task end
Task completion status: Unexpected error during task initialization. See
audit/error logs for details
************************************************************
```

Chapter 11. Defining Applications

Once the tasks and task groups of an application are implemented, you create application and menu definitions to control the application and to present tasks to users. *Chapter 11, "Defining Applications"* begins by walking you through the creation of a simple, complete application. The rest of the chapter explains how to define control characteristics for tasks, servers, and the application as a whole. *Chapter 12, "Defining Menus"* explains how to create menu definitions to access the application.

11.1. Defining a Simple Application

Suppose that your personnel department needs a way to monitor performance reviews for all company employees. The department also needs to use the DATATRIEVE procedure DUE, which displays reviews that are due, and the DCL command **EDIT**.

First, you develop the four tasks and the two task groups you need. The Department task group contains two tasks described in *Chapter 2, "Defining Tasks"*:

- Review History displays the performance history of one employee.
- Review Schedule displays the performance review schedules of an entire department.

The Administration task group contains two single-step tasks:

- DATATRIEVE procedure, DUE
- DCL command, **EDIT**

Chapter 13, "Defining Existing Applications as ACMS Tasks" explains how to define the Administration task group. See *Example 13.1, "A Task Group Definition"* for a listing of the task group definition.

The following sections explain how to create an application definition that contains and controls the tasks in these two task groups.

11.2. Describing the Application Environment

In the application definition, you describe the application environment by defining control characteristics for the application. Application definitions must:

- Name the task groups that contain the tasks of the application
- Assign a user name to the Application Execution Controller (EXC), the ACMS system process that controls the application

Access to tasks is always controlled from the application definition. You use application clauses to describe these and other application characteristics, as described in the following sections.

11.2.1. Naming Task Groups

You begin writing an application definition by naming the task groups of the application. You use the TASK GROUPS clause to name the task groups. The clause begins with the keywords TASK GROUPS ARE and ends with END TASK GROUPS. Between these keywords, you include:

- Name for each of the task groups in the application

- Name of the task group database file created by building each task group definition

The TASK GROUPS clause for the Personnel application is:

```
TASK GROUPS ARE
    DEPARTMENT_COBOL_TASK_GROUP :
        TASK GROUP FILE IS "ACMS$EXAMPLES:DEPRMSCOB.TDB";
    ADMINISTRATION_COBOL_TASK_GROUP :
        TASK GROUP FILE IS "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUPS;
```

Task groups in the application are DEPARTMENT_COBOL_TASK_GROUP and ADMINISTRATION_COBOL_TASK_GROUP. These names are not CDD path names for the task group; they are names that you assign in the application to identify the task groups. DEPRMSCOB.TDB and ADMRMSCOB.TDB are the task group database files that are created when you build the two task group definitions by using the ADU **BUILD** command.

You can name all the task groups in an application in a single TASK GROUPS clause, or you can group them in multiple TASK GROUPS clauses. For example, you could put the two task groups DEPARTMENT_COBOL_TASK_GROUP and ADMINISTRATION_COBOL_TASK_GROUP in separate TASK GROUPS clauses:

```
TASK GROUP IS
  DEPARTMENT_COBOL_TASK_GROUP : TASK GROUP FILE IS
      "ACMS$EXAMPLES:DEPRMSCOB.TDB";
END TASK GROUP;

TASK GROUP IS
  ADMINISTRATION_COBOL_TASK_GROUP : TASK GROUP FILE IS
      "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUP;
```

11.2.2. Naming a User Name for the Application Execution Controller

ACMS uses a process called the Application Execution Controller (EXC) to manage the servers that handle processing work for tasks. The EXC process requires special quotas and privileges to perform this work. Therefore, the application definition must include an OpenVMS user name for the EXC process that has the necessary quotas and privileges. Include the APPLICATION USERNAME clause in your source file.

```
APPLICATION USERNAME IS ACMSAMPLE;
TASK GROUPS ARE
    DEPARTMENT_COBOL_TASK_GROUP :
      TASK GROUP FILE IS "ACMS$EXAMPLES:DEPRMSCOB.TDB";
    ADMINISTRATION_COBOL_TASK_GROUP :
      TASK GROUP FILE IS "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUPS;
```

In this example, the user name is ACMSAMPLE. End the APPLICATION USERNAME clause with a semicolon (;).

11.2.3. Assigning Characteristics to Tasks and Servers

For applications more complex than the Personnel application, you can assign task and server control characteristics that affect how ACMS handles tasks and servers at run time. You assign these

characteristics by using subclauses within the DEFAULTS and ATTRIBUTES clauses of the application definition. There are two characteristics, however, that are very important for even the simplest application:

- Who can select a task
- The user names under which you want servers to run

One important purpose of an application definition is to control who can run which tasks in that application. You use the ACCESS subclause to control access to tasks.

For example, suppose that you want to let everyone in the Personnel department run all the tasks in the Personnel application and you want to prevent anyone outside the department from running any of the tasks in the application. If all the users in the Personnel department have user identification codes (UICs) with a group number of 300, such as [300,1], [300,2], and so on, you can set up this access control list (ACL) for the tasks in the application:

```
TASK DEFAULT IS
  ACCESS CONTROL LIST
    IDENTIFIER [300,*] ACCESS EXECUTE;
END TASK DEFAULT;
```

Any user with a UIC group number of 300 can run the tasks in the application. ACMS does not allow any other users to run the tasks. This ACCESS subclause overrides the default ACMS ACL, which allows all users to run all tasks.

For characteristics assigned with the TASK DEFAULTS clause to take effect, the TASK DEFAULTS clause must be placed before the TASK GROUPS clause and the TASK ATTRIBUTES clause in the application definition.

In addition to defining control characteristics such as access control to tasks, you can define control characteristics for servers.

It is important to assign user names to servers in an application because when a server runs, it takes on the privileges, priority, and quotas associated with its user name. The default value for server user names is the user name of the application. Because the application requires more privileges and quotas, and a higher priority than servers, it is a good idea to assign servers a user name different from the application user name.

For example, you can assign a server user name to all the servers in the Personnel application with the USERNAME subclause in the SERVER DEFAULTS clause:

```
SERVER DEFAULT IS
  USERNAME IS PERSONSVR;
END SERVER DEFAULT;
```

This subclause assigns the user name PERSONSVR to all the servers in the Personnel application. For characteristics assigned with the SERVER DEFAULTS clause to take effect, the SERVER DEFAULTS clause must be placed before the TASK GROUPS clause and the SERVER ATTRIBUTES clause in the application definition.

Example 11.1, "Personnel Application Definition" shows the application definition for the Personnel application.

Example 11.1. Personnel Application Definition

```
APPLICATION USERNAME IS ACMSAMPLE;
```

```
TASK DEFAULT IS
  ACCESS CONTROL LIST
    IDENTIFIER [300,*] ACCESS EXECUTE;
END TASK DEFAULT;

SERVER DEFAULT IS
  USERNAME IS PERSONSVR;
END SERVER DEFAULT;

TASK GROUPS ARE
  DEPARTMENT_COBOL_TASK_GROUP : TASK GROUP FILE IS
      "ACMS$EXAMPLES:DEPRMSCOB.TDB";
  ADMINISTRATION_COBOL_TASK_GROUP : TASK GROUP FILE IS
      "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUPS;
END DEFINITION;
```

You can give any number of users or groups of users access to tasks by using the ACCESS subclause. You can also use the same subclause to prevent certain users or groups of users from running tasks with the same subclause.

11.3. Controlling Tasks

You control tasks in an application by assigning task control characteristics in an application definition. These characteristics determine:

- Who can run a task
- Whether or not information about a task is written to the ACMS audit trail log
- What happens at the end of a task
- Whether or not the task is available for execution

When ADU begins processing an application definition, it assigns default values to all characteristics of tasks. You can reset these default values by assigning different characteristics to the tasks of an application by using the TASK ATTRIBUTES or TASK DEFAULTS clauses. Within these clauses, you use subclauses to describe specific task control characteristics.

The examples in the following sections use the TASK ATTRIBUTES clause to explain how to describe control characteristics for tasks in an application. *Section 11.3.4, "TASK ATTRIBUTES and TASK DEFAULTS Clauses"* describes the TASK ATTRIBUTES and TASK DEFAULTS clauses in more detail.

11.3.1. Controlling Access to Tasks

One of the important purposes of the application definition is to control who can run which tasks in an application. You use the ACCESS subclause to control access to tasks.

The Personnel application used in the examples in this book includes the DATR task. You can define an ACL to control which users have access to the DATR task. For example, you may want to let everyone in the Personnel department run this task, but to prevent anyone outside the department from running it. Because all the users in the Personnel department have UICs with a group number of 300, the ACL for the DATR task is [300,*] ACCESS EXECUTE. This ACL says that all users with a group number of 300 can run the task:

```
TASK ATTRIBUTE IS
```

```
  DATR : TASK DATR IN ADMINSTRATION_COBOL_TASK_GROUP;
             ACCESS CONTROL LIST
               IDENTIFIER [300,*] ACCESS EXECUTE;
END TASK ATTRIBUTE;
```

The default value for access control is to allow all users to run all tasks. This ACCESS subclause overrides this default ACL.

In some cases you may want to provide access to the same task by different groups of users. You can include access definitions for more than one group in the same ACCESS subclause. For example:

```
TASK ATTRIBUTE IS
  DATR : TASK DATR IN ADMINISTRATION_COBOL_TASK_GROUP;
             ACCESS CONTROL LIST
               IDENTIFIER [100,*] ACCESS EXECUTE,
               IDENTIFIER [300,*] ACCESS EXECUTE;
END TASK ATTRIBUTE;
```

This ACL allows all users with UIC group numbers of 100 or 300 to run the DATR task. If you include more than one access definition in an ACCESS subclause, separate them with commas. End the ACCESS subclause with a semicolon (;).

You can also use the NONE keyword to prevent a user or group of users from running a particular task. For example:

```
TASK ATTRIBUTE IS
  DATR : TASK DATR IN ADMINISTRATION_COBOL_TASK_GROUP;
             ACCESS CONTROL LIST
               IDENTIFIER [200,*] ACCESS NONE,
               IDENTIFIER [*,*] ACCESS EXECUTE;
END TASK ATTRIBUTE;
```

Because ACCESS NONE overrides the default value that allows all users to run all tasks, you must follow it with IDENTIFIER [*,*] ACCESS EXECUTE, so that users who are not in group 200 can run the DATR task.

Note

The most general access definition must be last in the list.

When ACMS looks at an ACL to find out whether a user can run a task, it starts at the top of the list and reads only until it finds an entry matching that user. If the definition for [*,*] is first in the ACL, then ACMS stops before finding the entry for [200,*], allowing users with group number 200 to run the DATR task.

In most cases, you group all access definitions for a single task in a single ACCESS subclause, but you can include more than one ACCESS subclause for each task, as well as more than one access definition in each ACCESS subclause. If you include more than one ACCESS subclause in one TASK ATTRIBUTES or TASK DEFAULTS clause, ACMS uses the lists as though they were one list; the values set in the TASK ATTRIBUTES clause override the values set in the TASK DEFAULTS clause.

11.3.2. Auditing Task Events

ACMS provides an auditing facility to record task events such as unexpected canceling of tasks. The Audit Trail Report Utility writes reports on task events from the audit trail log file.

You use the AUDIT subclause to control whether or not events such as task selections and completions are recorded in the audit trail log:

```
TASK ATTRIBUTE IS
 DATR : TASK DATR IN ADMINISTRATION_COBOL_TASK_GROUP;
        AUDIT;
END TASK ATTRIBUTE;
```

In this example, the audit trail log records task events for the DATR task whenever that task is run. The default value for the AUDIT subclause is NOAUDIT. For a list of the events written to the audit trail log, see *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/]. Even if you do not specify the AUDIT subclause, ACMS records all failure statuses.

11.3.3. Controlling What Happens When a Task Ends

From the application definition you can control what happens when an ACMS task ends. ACMS provides three options:

- Display the menu immediately
- Display the menu after a 3-second delay
- Display the menu after the user presses **Return**

The default value for this attribute is to display a menu immediately when a task ends. To delay the display of a menu for three seconds after a task ends, use the DELAY subclause:

```
TASK ATTRIBUTE IS
  DATR : TASK DATR IN ADMINISTRATION_COBOL_TASK_GROUP;
         DELAY;
END TASK ATTRIBUTE;
```

When a user finishes using the DATR task, ACMS waits for three seconds before displaying the next menu.

To delay the display of a menu until the user presses **Return** after a task ends, use the WAIT subclause. For example:

```
TASK ATTRIBUTE IS
  DATR  : TASK DATR IN ADMINISTRATION_COBOL_TASK_GROUP;
          WAIT;
END TASK ATTRIBUTE;
```

Now when a user finishes using the DATR task, ACMS waits until the user presses **Return** before displaying the menu.

You can also define the wait and DELAY subclauses in the task group definition. A WAIT or DELAY characteristic that you assign in a TASK ATTRIBUTES clause in an application definition overrides a WAIT or DELAY assignment in the task group definition.

Although all the examples so far have defined task control attributes within a TASK ATTRIBUTES clause, you can use these subclauses in a TASK DEFAULTS clause. The next section discusses the TASK ATTRIBUTES and TASK DEFAULTS clauses, the differences between the two, and the circumstances in which to use each.

11.3.4. TASK ATTRIBUTES and TASK DEFAULTS Clauses

When ADU begins processing an application definition, it assigns default values to all characteristics of tasks. Characteristics assigned with the TASK DEFAULTS or TASK ATTRIBUTES clauses reset the values of ACMS-supplied defaults. A characteristic assigned with the TASK DEFAULTS clause can become the value of the characteristic, or you can override it with a value supplied in the task group definition or a value supplied in a TASK ATTRIBUTES clause.

ACMS uses the following order of default to find values for task control characteristics:

1. TASK ATTRIBUTES clause in the application definition

   If you specifically define an attribute for a task in a TASK ATTRIBUTES clause, ADU uses that value for the attribute for that task.

2. TASKS clause in the task definition or the TASK subclause in the task group definition

   If you do not define a task attribute for a task in a TASK ATTRIBUTES clause, and if the attribute is one that you can assign in a task group definition, ADU looks in the task group database that defines implementation attributes for that task to see whether the attribute is defined there. The control attributes that you can define in a task group definition are DELAY, WAIT, CANCELABLE, LOCAL, and GLOBAL.

3. TASK DEFAULTS clause in the application definition

   ADU looks at the TASK DEFAULTS clauses in the application definition for any attribute that you do not define in the TASK ATTRIBUTES clause of the application or task group definition. The TASK DEFAULTS clause changes the ACMS-supplied default values for task attributes. An application definition can include more than one TASK DEFAULTS clause. The position of the TASK DEFAULTS clauses, TASK ATTRIBUTES clauses, and TASK GROUPS clauses in the application definition determines which task defaults apply to which tasks.

4. ACMS-supplied defaults

   ADU uses the default value it supplies only if you do not assign a value for the attribute in the TASK ATTRIBUTES or TASK DEFAULTS clause of the application definition, or in the task group database.

11.3.4.1. Using the TASK ATTRIBUTES Clause

You can include more than one task in a TASK ATTRIBUTES clause, and you can include more than one TASK ATTRIBUTES clause in an application definition.

In a TASK ATTRIBUTES clause, you must always name the task or tasks to which you want a subclause to apply. The task name must be unique in the application and must conform to the rules for ACMS identifiers. For example:

```
TASK ATTRIBUTE IS
  DATR : TASK DATR IN ADMINISTRATION_COBOL_TASK_GROUP;
         AUDIT;
END TASK ATTRIBUTE;
```

This TASK ATTRIBUTES clause assigns the name DATR to the DATR task in the ADMINSTRATION_COBOL_TASK_GROUP task group. A colon (:) separates the name from the

TASK and AUDIT subclauses. The name you assign to the left of the colon, DATR, *must* be unique within the application definition. However, the actual task name to the right of the colon needs to be unique within the task group only, *not* within the application. End each subclause with a semicolon (;).

The TASK keyword points to a task in a task group. In this example, the TASK keyword points to the DATR task in the Administration task group. The task group name must be the same as the name you used in the TASK GROUPS clause in the application definition.

11.3.4.2. Using the TASK DEFAULTS Clause

You can use TASK DEFAULTS clauses with TASK ATTRIBUTES clauses to simplify your application definition.

The TASK DEFAULTS clause changes the ACMS-supplied defaults for task control characteristics. These new defaults apply until the end of the definition, or until they are changed again with another TASK DEFAULTS clause. You can override the TASK DEFAULTS by assigning a value in a task group definition or a TASK ATTRIBUTES clause.

Several tasks can have one or more control attributes in common that are different from the ACMS-supplied defaults. In this case, one way to simplify your application definition is to use a TASK DEFAULTS clause.

The TASK DEFAULTS clause allows you to define an attribute that several tasks have in common in a single subclause. If you use the TASK ATTRIBUTES clause, you must name each task and the identical attribute for each task. If you use the TASK DEFAULTS clause, you can give users with the group UIC 100 access to both the DATR and EDIT tasks in a single subclause:

```
TASK DEFAULT IS
  ACCESS CONTROL LIST IDENTIFIER [100,*] ACCESS EXECUTE;
END TASK DEFAULT;

TASK ATTRIBUTES ARE
  DATR : ADMINISTRATION_COBOL_TASK_GROUP;
  EDIT : ADMINISTRATION_COBOL_TASK_GROUP;
END TASK ATTRIBUTES;
```

When you build an application database, ADU takes the ACL for the DATR and EDIT tasks from the TASK DEFAULTS clause. ACMS uses the defaults it supplies for all other task control attributes for those tasks.

The TASK DEFAULTS clause must precede the TASK GROUPS or TASK ATTRIBUTES clause to which you want it to apply.

Example 11.2, "Application Definition Using TASK DEFAULTS" shows an application definition that uses a TASK DEFAULTS clause to define control attributes for all the tasks in the application. The application includes only one task group.

Example 11.2. Application Definition Using TASK DEFAULTS

```
REPLACE APPLICATION PERSONNEL_APPLICATION
USERNAME IS PERSONNEL;
TASK DEFAULTS ARE
  ACCESS CONTROL LIST IDENTIFIER [200,*] ACCESS EXECUTE;
  AUDIT;
END TASK DEFAULTS;
```

```
TASK GROUP IS
 ADMINISTRATION_COBOL_TASK_GROUP : TASK GROUP FILE IS
   "ACMS$EXAMPLES:ADMRMSCOB.TDB";
 END TASK GROUP;
END DEFINITION;
```

If an application includes only one task group, and if all the tasks in the application use the same control characteristics, the application definition can be as simple as this, even if the application includes many tasks.

11.3.4.3. Defaulting Task and Task Group Names

Depending on the position of TASK ATTRIBUTES clauses in the application definition, you may not need to explicitly name the task or task group to which you assign a control characteristic. ACMS provides some defaulting of task and task group names in the application definition.

The following TASK ATTRIBUTE clause includes both the task and the task group name of the task to which the AUDIT subclause applies:

```
TASK ATTRIBUTE IS
  DATR : TASK DATR IN ADMINISTRATION_COBOL_TASK_GROUP;
         AUDIT;
END TASK ATTRIBUTE;
```

In the preceding TASK ATTRIBUTE clause, there are two phrases: the task name and the task group name. In some cases, you can omit one of these phrases. If the task has the same name in the application as it has in the task group, you do not have to use the task name phrase. For example:

```
TASK DEFAULT IS
 ACCESS CONTROL LIST IDENTIFIER [100,*] ACCESS EXECUTE;
END TASK DEFAULT;

TASK ATTRIBUTES ARE
  DATR : IN ADMINISTRATION_COBOL_TASK_GROUP;
  EDIT : IN ADMINISTRATION_COBOL_TASK_GROUP;
END TASK ATTRIBUTES;
```

If the task group is the same as the last one named in the immediately preceding TASK GROUPS clause, you do not have to use the task group phrase. For example:

```
TASK GROUP IS
  ADMINISTRATION_COBOL_TASK_GROUP : TASK GROUP FILE IS
      "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUP;

TASK ATTRIBUTE IS
  DATR :  AUDIT;
END TASK ATTRIBUTE;
```

If you do not specify a task group name in a TASK ATTRIBUTES clause, ACMS defaults the task group name from the last task group name in the immediately preceding TASK GROUPS clause. If you name the task group in the TASK ATTRIBUTES clause, then the TASK ATTRIBUTES clause does not have to follow the TASK GROUPS clause to which you want it to apply.

Task attribute and default values affect tasks in a task group depending on the position of the clauses in relation to each other in an application definition. The next section discusses the positioning of the TASK ATTRIBUTES and TASK DEFAULTS clauses in an application definition.

11.3.4.4. Positioning TASK ATTRIBUTES and TASK DEFAULTS Clauses

The way you place TASK ATTRIBUTES and TASK DEFAULTS clauses in an application definition affects how ACMS assigns control characteristics to the tasks in the application.

For example, *Example 11.3, "Application Definition Using Multiple TASK DEFAULTS"* shows an application definition that uses multiple TASK DEFAULTS clauses to define different task control characteristics for the tasks in two task groups.

Example 11.3. Application Definition Using Multiple TASK DEFAULTS

```
REPLACE APPLICATION PERSONNEL_APPLICATION
USERNAME IS PERSONNEL;
TASK DEFAULTS ARE
  ACCESS CONTROL LIST
    IDENTIFIER [100,*] ACCESS EXECUTE,
    IDENTIFIER [200,*] ACCESS EXECUTE;
  AUDIT;
END TASK DEFAULTS;

TASK GROUP IS
  DEPARTMENT_COBOL_TASK_GROUP : TASK GROUP FILE IS
    "ACMS$EXAMPLES:DEPRMSCOB.TDB";
END TASK GROUP;

TASK DEFAULTS ARE
  ACCESS CONTROL LIST IDENTIFIER [200,*] ACCESS EXECUTE;
END TASK DEFAULTS;

TASK GROUP IS
  ADMINISTRATION_COBOL_TASK_GROUP : TASK GROUP FILE IS
    "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUP;
END DEFINITION;
```

This first TASK DEFAULTS clause defines a default ACL. ADU assigns this ACL to all the tasks in the Department group. The second TASK DEFAULTS clause changes that default ACL. ADU assigns the second ACL to all the tasks in the Administration group. So, the only users who can run the tasks in the Administration task group are those who have a group UIC of 200.

The application definition also assigns AUDIT to all the tasks in the application. AUDIT applies to both task groups because the first TASK DEFAULTS clause used the AUDIT subclause and the second TASK DEFAULTS clause did not use the NOAUDIT subclause.

Defaults set in a TASK DEFAULTS clause remain in effect unless changed by a later TASK DEFAULTS clause. Any control attributes not named in a TASK DEFAULTS clause retain their ACMS-supplied defaults. You can also override a default value by assigning a control attribute in the task group definition of a task or by using the TASK ATTRIBUTES clause.

Example 11.4, "Application Using TASK ATTRIBUTES and TASK DEFAULTS" shows an application definition that includes one task group and uses TASK ATTRIBUTES and TASK DEFAULTS clauses to define the control attributes of the tasks in the application.

Example 11.4. Application Using TASK ATTRIBUTES and TASK DEFAULTS

```
REPLACE APPLICATION PERSONNEL_APPLICATION
```

```
USERNAME IS PERSONNEL;
TASK DEFAULTS ARE
  ACCESS CONTROL LIST
    IDENTIFIER [100,*] ACCESS EXECUTE,
    IDENTIFIER [200,*] ACCESS EXECUTE;
  AUDIT;
END TASK DEFAULTS;

TASK GROUP IS
  ADMINISTRATION_COBOL_TASK_GROUP : TASK GROUP FILE IS
    "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUP;

TASK DEFAULTS ARE
  ACCESS CONTROL LIST IDENTIFIER [200,*] ACCESS EXECUTE;
END TASK DEFAULTS;

TASK ATTRIBUTES ARE

  DATR : IN ADMINISTRATION_COBOL_TASK_GROUP;
  EDIT : IN ADMINISTRATION_COBOL_TASK_GROUP;

END TASK ATTRIBUTES;
END DEFINITION;
```

In *Example 11.4, "Application Using TASK ATTRIBUTES and TASK DEFAULTS"*, all the tasks in the application take the ACMS default values for all attributes except ACCESS and AUDIT. The definition assigns ACLs for the DATR and EDIT tasks; only users with a group UIC of 100 or 200 can run these tasks. For the other tasks in the application, the definition assigns an ACL that allows all users with group UIC of 100 or 200 to run the tasks. The application also assigns AUDIT to all tasks with the AUDIT clause in the first TASK DEFAULTS clause.

When you write an application definition, use the order of TASK DEFAULTS, TASK GROUP, and TASK ATTRIBUTES clauses that lets you take maximum advantage of defaulting. Your goal is to make the application definition as simple and easy to understand as possible so that the control characteristics for your application are clear to anyone who works with the application definition.

11.3.5. Enabling and Disabling Tasks in the Application Definition

Section 11.6, "Modifying an Active Application" describes how to enable or disable tasks on a temporary basis with the **ACMS/MODIFY APPLICATION** command. By using the DISABLED/ENABLED attribute in the application definition, you can specify on a permanent basis whether or not the task is available for selection by task submitters.

Example 11.5, "Enabling and Disabling Tasks in the Application Definition" shows this attribute in a definition.

Example 11.5. Enabling and Disabling Tasks in the Application Definition

```
REPLACE APPLICATION TEST_APPL
USERNAME IS TPSS;

TASK DEFAULTS ARE
 DISABLE;
```

```
END TASK DEFAULTS;

TASK GROUPS ARE
 CDUTEST_GROUP1:
  TASK GROUP FILE IS "CDU$:CDUTEST_GROUP1.TDB";
END TASK GROUPS;

SERVER ATTRIBUTES ARE
 RDBLSRV1:
  MINIMUM SERVER PROCESSES IS 1;
  SERVER RDBLSRV1 IN CDUTEST_GROUP1;
  AUDIT;
END SERVER ATTRIBUTES;

TASK ATTRIBUTES ARE
 RDB_SINGLE_TASK;
  ENABLE;
  TASK RDB_SINGLE_TASK IN CDUTEST_GROUP1;
END TASK ATTRIBUTES;

END DEFINITION;
```

In this example, all tasks are initially disabled by default in the TASK DEFAULTS ARE clause. The RDB_SINGLE_TASK is subsequently enabled in the TASK ATTRIBUTES ARE clause. The other tasks in the task group CDU_TEST_GROUP1 remain disabled.

11.3.6. Controlling Transaction Timeouts in the Application Definition

If your application contains tasks that use distributed transactions, you might need to concern yourself with possible deadlock problems. One type of deadlock involves multiple tasks attempting to access the same server process. For example, suppose two tasks each use two servers. Each server has one active server process. The first task accesses the first server, and the second task accesses the second server. If both tasks then attempt to access the other server, they will become deadlocked waiting to use the server process being used by the other task.

If your application and databases are distributed across multiple systems that are not part of a single OpenVMS Cluster system, deadlocks can occur when multiple tasks attempt to access the same database record. The OpenVMS Lock Manager is able to detect deadlocks only within a single system or an OpenVMS Cluster system.

By specifying the TRANSACTION TIMEOUT subclause in the application definition, you can instruct ACMS to abort a transaction if it has not completed within a certain number of seconds. *Example 11.6, "Using TRANSACTION TIMEOUT in the Application Definition"* shows this attribute in a definition.

Example 11.6. Using TRANSACTION TIMEOUT in the Application Definition

```
REPLACE APPLICATION AVERTZ_CDD_APPL:VR_APPL
USERNAME IS AVERTZ_EXC;

AUDIT;
TASK DEFAULTS ARE
 AUDIT;
 TRANSACTION TIMEOUT IS 1200;
END TASK DEFAULTS;
```

```
SERVER CONTROL ATTRIBUTES ARE
 VR_READ_SERVER:
  SERVER VR_READ_SERVER IN VR_TASK_GROUP;
  AUDIT;
  MAXIMUM SERVER PROCESSES IS 3;
  MINIMUM SERVER PROCESSES IS 1;

 VR_UPDATE_SERVER:
  SERVER VR_UPDATE_SERVER IN VR_TASK_GROUP;
  AUDIT;
  MAXIMUM SERVER PROCESSES IS 1;
  MINIMUM SERVER PROCESSES IS 1;
END SERVER ATTRIBUTES;

TASK GROUP IS
 VR_TASK_GROUP:
  TASK GROUP FILE IS "AVERTZ_DEFAULT:VR_TASK_GROUP.TDB";
END TASK GROUP;

END DEFINITION;
```

In this example, the TRANSACTION TIMEOUT subclause specifies that by default all transactions should be aborted if they do not complete within 20 minutes. Be sure to set the transaction time limit to a value higher than the amount of time it takes to complete your longest transaction when the system is at peak load. When a transaction aborts, ACMS cancels the task. For information on how to define a task definition so that the task can recover from a transaction timeout error and retry the distributed transaction, see *Chapter 8, "Handling Task Execution Errors"*.

11.4. Controlling Servers

You control servers in an application by assigning server control attributes in the application definition. These attributes determine the processing characteristics of a server, including:

- Its OpenVMS user name
- Whether the user name, UIC, and default directory of a server remain the same when the server processes tasks for different terminal users
- Maximum and minimum number of server processes the server can have
- Logical names and name tables for a server process
- Whether or not server events are audited
- Default directory for the server
- How frequently new server processes are created and deleted
- Protected workspaces
- Server process dumps

ACMS supplies default values for server control characteristics. If one or more servers in your application need control characteristics different from the defaults, you use the SERVER ATTRIBUTES

and SERVER DEFAULTS clauses to override the ACMS-supplied defaults. Within these clauses, you use subclauses to describe specific server control characteristics.

11.4.1. Assigning a Server User Name

Every server has an OpenVMS user name that you assign in the application definition with the USERNAME subclause and that determines:

- How the work done by a server process is logged by the OpenVMS ACCOUNTING facility
- Privileges, priority, and quotas assigned to a server process
- Default directory assigned to a server process when the process is created

ACMS assigns the user name of the application to the server by default. However, if you use the USERNAME OF TERMINAL USER in the task group definition, ADU always assigns the server that name. Because an application requires more privileges, higher quotas, and larger working sets than a server, you should instead assign a server a user name that has only the necessary privileges, quotas, and priority for the work that server has to do. The user name must be a valid OpenVMS user name consisting of 1 to 12 alphanumeric characters, including underscores.

In many cases you want all servers handling tasks in a group to run under the same user name. For example, many of the tasks in the ACMS sample application use a server named ADMINISTRATION_SERVER. The following application definition uses the SERVER DEFAULTS clause to assign the user name PERSONNEL to this server:

```
SERVER DEFAULTS ARE
  USERNAME IS PERSONNEL;
END SERVER DEFAULTS;

TASK GROUP IS
  ADMINISTRATION_COBOL_TASK_GROUP : TASK GROUP FILE IS
    "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUP;
```

This example uses the SERVER DEFAULTS clause and a USERNAME subclause to change the ADU default for SERVER USERNAME from USERNAME OF APPLICATION to PERSONNEL. All servers used by ADMINISTRATION_COBOL_TASK_GROUP run under the same user name. For all other attributes of servers, ADU uses the defaults in effect, which are ACMS supplied, except for the user name, which was reset in the SERVER DEFAULTS clause.

If you want a server to have the same priorities, privileges, and quotas as the Application Execution Controller, define the server to run under the same user name as the Application Execution Controller. Use the USERNAME OF APPLICATION subclause to define this type of server user name.

```
SERVER DEFAULTS ARE
  USERNAME IS USERNAME OF APPLICATION;
END SERVER DEFAULTS;
```

If the application user name is PERSONNEL, this USERNAME subclause has the same effect as defining PERSONNEL as the user name. USERNAME OF APPLICATION is the default value for the user name.

Sometimes a task may require a server process with all the characteristics assigned to the terminal user who selected the task. For example, if a user needs to run a task under the user's own user name, the task

requires a server process with all the characteristics assigned to that user. There are two ways to define this type of server for a task:

- Define a specific dynamic user name for servers. Dynamic user names are discussed in *Section 11.4.2, "Assigning a Dynamic or Fixed Server User Name"*.
- Define a server process with all the characteristics assigned to the terminal user who selected the task, using the USERNAME OF TERMINAL USER subclause. For example:

```
SERVER DEFAULTS ARE
  USERNAME IS USERNAME OF TERMINAL USER;
END SERVER DEFAULTS;

SERVER ATTRIBUTES ARE
  PRIVATE_UTILITY_SERVER: IN PERSONNEL_TASK_GROUP;
END SERVER ATTRIBUTES;
```

  In this case, each time a task from the Personnel task group needs a server, ACMS starts a server process with all the characteristics of the terminal user's user name. When the task finishes using the server, ACMS stops the server process. The PRIVATE_UTILITY_SERVER is a DCL server.

  The USERNAME OF TERMINAL USER subclause is intended for use with DCL servers only.

  If you use the USERNAME OF TERMINAL USER subclause for a server:

  - The value of MINIMUM SERVER PROCESSES for that server must be 0
  - The server process is deleted when the task finishes using the server

Before a server can be started, the server user name must be authorized with the OpenVMS Authorize Utility. See *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/] for information on the Authorize Utility.

11.4.2. Assigning a Dynamic or Fixed Server User Name

When a task is selected by the terminal user, you may want a server to change its OpenVMS user name to the user name of the user who selected the task if:

- The task requires access to the terminal user's files
- The task uses the terminal user's UIC to access files that are not in the default directory of the server in which the task is running

A server can change its OpenVMS user name if you assign a dynamic user name to the server with the DYNAMIC USERNAME subclause for DCL servers. Procedure servers must always have a fixed user name.

Assigning a dynamic user name to a DCL server allows each server process allocated by the server to change its user name to match the user name of the terminal user. For example:

```
SERVER DEFAULTS ARE
  DYNAMIC USERNAME;
END SERVER DEFAULTS;

SERVER ATTRIBUTES ARE
```

```
  PRIVATE_UTILITY_SERVER: IN PERSONNEL_TASK_GROUP;
                          USERNAME IS PERSONNEL;
END SERVER ATTRIBUTES;
```

Here you use USERNAME IS to assign an initial user name of PERSONNEL to all servers in the Personnel task group. The PRIVATE_UTILITY_SERVER is a DCL server. Each time a server is started, it has the user name, privileges, quotas, priority, and other characteristics of PERSONNEL. Because the user name is defined as dynamic, each time a server process is allocated to a task, ACMS changes the user name of the server process to match that of the user who selected the task. These changes include:

- User name set to that of the terminal user
- Default directory set to that of the terminal user
- UIC set to that of the terminal user
- SYS$LOGIN set to the user's default device and directory
- SYS$SCRATCH set to the user's default device and directory
- SYS$DISK set to the user's default device

ACMS does not change the quotas, privileges, or priority of the server process. These remain the same as they were when ACMS started the server process. If you need the server process to have the quotas, privileges, or priority of the terminal user's process, use the USERNAME OF TERMINAL USER subclause. Because a USERNAME OF TERMINAL USER subclause forces ACMS to start and stop a server process each time a task needs that server, it adds to the processing cost. A dynamic user name does not stop the server process when a task is finished using the process. Instead, the process becomes available for another task instance. Not having to start and stop a server process each time a task needs a server reduces the processing cost.

By default, server user names are fixed. If the server user name is fixed, the UIC, user name, default directory and other characteristics of the server remain the same as they were when the server process was started.

For more information about how different types of user names affect the performance of your ACMS system, see *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/].

11.4.3. Assigning Server Default Directories

ACMS assigns every server process a default device and directory. The values come from the device and directory assigned to the user name of each server.

Assigning the default directory for a server is necessary when you need a different default directory from the one assigned to the server's user name in the SYSUAF file. For example, you need to define server default directories when two servers need most of the same characteristics, but the servers access different databases in different directories. You also need to assign a default directory if files needed by the server, like data files or the procedure server image for a procedure server, are identified only by file name rather than by complete file specification.

Note

Do not use both a DYNAMIC USERNAME clause *and* a DEFAULT DIRECTORY clause for the same server. If you do, when a DCL server process is created for that server, the DEFAULT DIRECTORY

clause sets the default directory. However, each time a task uses that DCL server process, the DYNAMIC USERNAME clause changes the default directory to that of the terminal user submitting the task.

You can name a default device and directory with the DEFAULT DIRECTORY subclause. For example:

```
SERVER DEFAULTS ARE
  USERNAME IS PERSONNEL;
  DEFAULT DIRECTORY IS ACMS$EXAMPLES;
END SERVER DEFAULTS;

TASK GROUP IS
  ADMINISTRATION_COBOL_TASK_GROUP : TASK GROUP FILE IS
    "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUP;
```

This definition assigns the device and directory pointed to by the logical name ACMS$EXAMPLES as the default directory for the servers in the Administration task group. You can name a directory specification and device name rather than a logical name, but using logical names lets you change directory locations without redefining and rebuilding the application.

By default, ACMS assigns USERNAME DEFAULT DIRECTORY for the DEFAULT DIRECTORY clause. When the default is in effect, ACMS assigns the server process the default directory that is in the SYSUAF entry for the server user name.

11.4.4. Assigning Server Logical Names

Sometimes the image running in a server process can use logical names to refer to the data files it uses, or the definition for a procedure server can use a logical name to identify a procedure server image. In both these cases, you need to make the translation of the logical names available to the server process. There are several ways to do this:

- Define the logical names as system logicals.
- Define the logical names as group logicals. Make sure that the logical names are available to the group UIC for the server process.
- Use the LOGICAL NAMES subclause to define process logical names for the server process.

  There are two reasons for using the LOGICAL NAMES subclause:

  - If you are using servers with dynamic user names, and if users with different group UIC numbers can run tasks, group logicals available to one task instance may not be available to other task instances.
  - By using the LOGICAL NAMES subclause to define process logical names, you ensure that logical name assignments cannot conflict with logical names needed by other processes.
- Use the NAME TABLE subclause to define a list of tables that define logical names.

In the LOGICAL NAMES subclause, list each logical name and its equivalent name. For example:

```
SERVER DEFAULTS ARE
  USERNAME IS PERSONNEL;
  LOGICAL NAMES
    ACMS$EXAMPLES  = "DBA7:[PERSONNEL]",
```

```
    PERS_FILE  = "ACMS$EXAMPLES:PERSFILE.DAT";
END SERVER DEFAULTS;

TASK GROUP IS
  ADMINISTRATION_COBOL_TASK_GROUP : TASK GROUP FILE IS
    "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUP;
```

In this example, the LOGICAL NAMES subclause defines the logical name ACMS$EXAMPLES as the directory DBA7:[PERSONNEL]. It also defines the logical name PERS_FILE as the file name PERSFILE.DAT, in the ACMS$EXAMPLES directory. If a logical name or equivalent name does not conform to the syntax rules for ACMS identifiers, enclose it in quotation marks.

The following logical names are always assigned by ACMS:

- SYS$DISK
- SYS$LOGIN
- SYS$SCRATCH

ACMS automatically assigns the following logical names for the DCL servers you define. The logicals are assigned to either the terminal or the null device, depending on whether or not the processing step of the task uses the terminal. The logical names are assigned in supervisor mode. These logical names are:

- SYS$INPUT
- SYS$OUTPUT
- SYS$ERROR
- SYS$COMMAND
- TT

For procedure servers, if the processing step uses a terminal, ACMS assigns the following logical names:

- SYS$INPUT
- SYS$OUTPUT
- SYS$ERROR
- SYS$COMMAND
- TT

11.4.5. Creating Logical Name Tables for Application Servers

Using name tables can give you the following advantages over using logical names:

- Modify logicals used by an application or server without rebuilding application definitions
- Share common logicals among servers and applications without duplicating definitions
- Reduce server process initialization time by reducing the number of server process logicals

You can specify a list of logical name tables in the application definition for reference by applications and servers. For example:

```
APPLICATION NAME TABLE IS LNM$PROCESS,
                          LNM$JOB,
                          OUR_APPL_LOGICALS,
                          LNM$GROUP,
                          LNM$SYSTEM;
```

If you specify the APPLICATION NAME TABLE IS clause and you want the Application Execution Controller to use any of the default tables, you must name these default tables in the clause.

ACMS uses the list of logical name tables specified in the application definition to define the logical LNM$FILE_DEV in the process directory table for the appropriate server or application process. The system uses **process directory tables** when translating logical names for the application or server process. For example, LNM$FILE_DEV must translate to a search list of one or more logical name table names which specify the tables and the search order of the tables for translating file specifications.

If you do not specify APPLICATION NAME TABLE IS, ACMS uses the default definition of LNM$FILE_DEV in the system logical name directory table, which normally holds the process, job, group, or system logical name tables.

For more information on logical name tables, see the OpenVMS documentation set.

11.4.6. Controlling the Number of Server Processes

One of the important factors in improving the performance of your ACMS application is the number of active server processes allowed for that application. There must be enough server processes available for users to do their work and as few idle processes as possible. To control the number of server processes available for each server, you use the MAXIMUM SERVER PROCESSES and MINIMUM SERVER PROCESSES subclauses. For example:

```
SERVER DEFAULTS ARE
  MAXIMUM SERVER PROCESSES IS 5;
  MINIMUM SERVER PROCESSES IS 1;
END SERVER DEFAULTS;

TASK GROUP IS
  ADMINISTRATION_COBOL_TASK_GROUP : TASK GROUP FILE IS
    "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUP;
```

For each server in the Administration task group, ACMS starts a single server process when it starts the application. As more server processes are necessary, ACMS starts them. When the number of processes for the server reaches five, ACMS does not start any more processes. Any further tasks that needs a server process must wait until a process is free. As tasks finish using the server processes, ACMS gradually decreases the number of processes for each server, until the number reaches the minimum of one.

If you define a server with USERNAME OF TERMINAL USER, MINIMUM SERVER PROCESSES must be zero for that server because when the application starts, there is no terminal user for the server. Zero is the default value for MINIMUM SERVER PROCESSES.

The best way to judge how many server processes are necessary for your application is to use the **ACMS/SHOW APPLICATION** command to determine how your application is using server processes.

You can also check with terminal users about system response time. See *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/] for information on the **ACMS/SHOW APPLICATION** command and on how to interpret the information it supplies to determine appropriate values for the MAXIMUM and MINIMUM SERVER PROCESSES subclauses. See *Section 11.5.6, "Controlling the Number of Server Processes"* for information on the application-level MAXIMUM SERVER PROCESSES clause and on how ACMS determines a value for MAXIMUM SERVER PROCESSES.

11.4.7. Creating and Deleting Server Processes

Business environments can differ widely in the pattern of the work day. For example, a stock broker's office is busiest when the New York Stock Exchange is in session. A dairy farm depends on the cows' schedule. Your work environment may be mildly busy in the morning, slack over the lunch hour, and very busy in the afternoon. The demand for ACMS server processes may vary, too. It could be stable through the day, with only minor variations, or it could suddenly and sharply increase and just as suddenly decrease.

ACMS creates and deletes new server processes as necessary, within the parameters of maximum and minimum server processes. The queue of tasks waiting for server processes is polled at 5-second intervals, as are inactive server processes. Before beginning to create new server processes, ACMS delays 10 seconds (default) to be sure that the requirement cannot be filled when another user completes a task and releases a server process. If no server process becomes available in this way, ACMS creates new server processes at intervals of 10 seconds (default) until the maximum is reached or no tasks are waiting. Similarly, when a server process becomes inactive, ACMS delays 30 seconds (default) to be sure no other task requires it. ACMS then deletes server processes at intervals of 15 seconds (default) until the minimum is reached or there are no more inactive server processes.

ACMS provides the ability to adjust these delays and intervals according to the pattern of demand for your application. Four server control subclauses are available for this purpose:

- CREATION DELAY
- CREATION INTERVAL
- DELETION DELAY
- DELETION INTERVAL

For example, a stock broker's office runs an ACMS application, recording transactions with the New York Stock Exchange as well as customer transactions. Both tasks use the same server. Customer transactions are steady throughout the day. Stock exchange transactions, however, begin at 10 a.m. and end at 4 p.m. The application definition uses these server control subclauses:

```
SERVER ATTRIBUTES ARE
  BROKER_SERVER:  CREATION DELAY IS 5;
                  CREATION INTERVAL IS 2;
                  DELETION DELAY IS 15;
                  DELETION INTERVAL IS 5;
                  MAXIMUM SERVER PROCESSES IS 10;
                  MINIMUM SERVER PROCESSES IS 0;
END SERVER ATTRIBUTES;
```

The application runs the NYSE task and the CUSTOMER task in the BROKER_SERVER. The CREATION DELAY of 5 seconds means that ACMS waits 5 seconds before starting to create new

processes of BROKER_SERVER, to see if old processes become available. Before 10 a.m., the requirement for server processes is usually filled in this way.

After the New York Stock Exchange opens, demand for server processes increases sharply, and tasks are kept waiting for longer than 5 seconds. ACMS then begins to create new server processes at a CREATION INTERVAL of 2 seconds (up to the maximum of 10), quickly filling the need. Demand continues steadily until 4 p.m. when instances of the NYSE task are no longer required. ACMS then waits 15 seconds, the DELETION DELAY specified, before beginning to delete inactive server processes at a DELETION INTERVAL of 5 seconds, until there are no inactive BROKER_SERVER processes (minimum number).

ACMS does not monitor the queue of waiting tasks continuously; there is a monitoring interval which can be set with the SERVER MONITORING INTERVAL application clause. The actual delay at run time includes a waiting period of up to the monitoring interval. For example, if the definition specifies a CREATION DELAY of 10 seconds and if the monitoring interval is 5 seconds, the actual delay can be anywhere from 10 to 15 seconds. Use the SERVER MONITORING INTERVAL clause with caution, however. Setting the monitoring interval too low can affect performance, causing ACMS to use its resources monitoring its own requirements.

The use of the clauses controlling minimum and maximum server processes and the associated delays and intervals enables ACMS to adapt the supply of server processes to the demands of your business environment. In using these server control subclauses, you need to strike a balance between having inactive server processes and keeping your users waiting. In most environments, the default settings provided with ACMS strike this balance. You can handle unusual situations with the server control subclauses.

11.4.8. Replacing an Active Server

Once an application is running, you may want to make code changes to a server. ACMS gives an application manager the capability to dynamically replace a server within an active application without affecting task execution or availability of the application.

To dynamically replace a server in an active application, follow these steps:

1. Make the necessary changes to the new server's source file.
2. Compile and relink the new server image.
3. Fully test the new image.
4. Copy the server image to the location specified by the SERVER IMAGE IS statement in the application's task definition.
5. Issue the **ACMS/REPLACE SERVER** command.

For example, an application manager might be integrating some code changes to an existing server called DEPRMSCOB.EXE, whose server image definition is:

```
SERVER IS
  DEPARTMENT_SERVER:
    PROCEDURE SERVER IMAGE IS "ACMS$EXAMPLES:DEPRMSCOB.EXE"
            .
            .
            .
END SERVER;
```

The logical name ACMS$EXAMPLES points to the directory in which ACMS expects to find the server image.

After compiling, relinking, and testing the server image, the application manager must copy the server image to the correct directory:

```
$ COPY DEPRMSCOB.EXE ACMS$EXAMPLES:DEPRMSCOB.EXE
```

Finally, the application manager uses the **ACMS/REPLACE SERVER** command to activate the new server using the server name defined in the application.

```
$ ACMS/REPLACE SERVER DEPARTMENT_SERVER/APPLICATION=PERSONNEL
```

When the application controller receives the **ACMS/REPLACE SERVER** command, it runs down all free server processes for the specified server and requests all active servers to run down when they are free. A server retaining context does not run down until it releases context. ACMS creates new server processes to meet the MIN and MAX requirements of the application.

For more information on modifying an active server, consult *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/].

11.4.9. SERVER ATTRIBUTES and SERVER DEFAULTS Clauses

When ADU begins processing an application definition, ACMS assigns default values to all characteristics of servers. You can change these default values by assigning different characteristics to the servers of an application with the SERVER ATTRIBUTES or SERVER DEFAULTS clauses.

A characteristic assigned with the SERVER DEFAULTS clause can become the value of the characteristic or it can be overridden by a value supplied in the task group definition or a value supplied in a SERVER ATTRIBUTES clause.

ADU uses the following order of defaulting to find values for server attributes:

1. SERVER ATTRIBUTES clause in the application definition

   If a characteristic is defined for a server in a SERVER ATTRIBUTES clause, ADU uses that value for the characteristic for that server.

2. SERVERS clause in the task group definition

   If a server characteristic is not defined for a server in a SERVER ATTRIBUTES clause, and if the characteristic is one that can be assigned in a task group definition, ADU looks in the task group database that defines implementation for that server. The two control characteristics that can be defined in a task group definition are USERNAME OF TERMINAL USER and FIXED/DYNAMIC USERNAME.

3. SERVER DEFAULTS clause in the application definition

   ADU looks at the SERVER DEFAULTS clauses in the application definition for any characteristic not defined in the application or task group definition. The SERVER DEFAULTS clause sets up default values for server characteristics. An application definition can include more than one of these clauses. The position of the SERVER DEFAULTS, SERVER ATTRIBUTES, and TASK GROUP clauses in the application definition determines which server defaults apply to which servers.

4. ACMS-supplied defaults

ADU uses the default value it derives only if no value is assigned for the characteristic in the SERVER ATTRIBUTES or SERVER DEFAULTS clause of the application definition, or the task group database.

You can include more than one server in a SERVER ATTRIBUTES clause and you can also include more than one SERVER ATTRIBUTES clause in an application definition.

In a SERVER ATTRIBUTES clause, you must always name the server to which you want a subclause to apply. This name must be unique for each server in the application and it must conform to the rules for identifiers. For example:

```
SERVER ATTRIBUTE IS
   UTILITY_SERVER : SERVER UTILITY_SERVER IN DEPARTMENT_COBOL_TASK_GROUP;
                    USERNAME IS DEPARTMENT;
 END SERVER ATTRIBUTE;
```

This SERVER ATTRIBUTES clause assigns the name UTILITY_SERVER to the UTILITY_SERVER. A colon separates the server name from the SERVER keyword and the USERNAME subclause. You must end the subclause with a semicolon (;).

The SERVER keyword points to a server in a task group. In the example, the SERVER keyword points to the UTILITY_SERVER. The task group name must be the same as the name used in the TASK GROUPS clause in the application definition. You can use SERVER DEFAULTS clauses with SERVER ATTRIBUTES clauses to simplify your application definition.

The SERVER DEFAULTS clause resets the ACMS defaults for server control characteristics. These new defaults apply until the end of the definition, or until they are changed again with another SERVER DEFAULTS clause. You can override the defaults by assigning a value assigned in a task group definition or a SERVER ATTRIBUTES clause.

Several servers may have one or more control attributes in common that are different from the ACMS-supplied defaults. In this case, one way to simplify your application definition is to use a SERVER DEFAULTS clause.

The SERVER DEFAULTS clause allows you to define, in a single subclause, an attribute that several servers have in common. Using the SERVER ATTRIBUTES clause, you have to name each server and the identical attribute for each. For example, using the SERVER DEFAULTS clause, you can assign all the servers in the DEPARTMENT task group the user name DEPARTMENT in a single subclause:

```
SERVER DEFAULT IS
  USERNAME IS DEPARTMENT;
END SERVER DEFAULT;

TASK GROUP IS
  DEPARTMENT_COBOL_TASK_GROUP : TASK GROUP FILE IS
      "ACMS$EXAMPLES:DEPRMSCOB.TDB";
END TASK GROUPS;
```

When you build the application database, ADU assigns the user name Department to every server in the Department task group. ADU uses the defaults it derives for all other server control characteristics for those servers.

Remember that control characteristics assigned either in SERVER ATTRIBUTES clauses or in the task group definition override values assigned with the SERVER DEFAULTS clause. Also, the SERVER DEFAULTS clause must precede the TASK GROUPS clause to which you want it to apply.

Example 11.7, "Application Definition Using Server Defaults" shows an application definition that uses a SERVER DEFAULTS clause to define control attributes for all the servers in the application. The application includes only one task group.

Example 11.7. Application Definition Using Server Defaults

```
USERNAME IS PERSONNEL;
SERVER DEFAULTS ARE
  AUDIT;
  USERNAME IS DEPARTMENT;
  MAXIMUM SERVER PROCESSES IS 5;
  MINIMUM SERVER PROCESSES IS 1;
END SERVER DEFAULTS;

TASK GROUP IS
   DEPARTMENT_TASK_GROUP: TASK GROUP FILE IS
                          "ACMS$EXAMPLES:DEPART.TDB";
END TASK GROUP;
```

If an application contains only one task group and if all servers in the application use the same control attributes, the application definition can be as simple as this, even if the application includes many tasks.

11.4.10. Defaulting Server and Task Group Names

Depending on the position of SERVER ATTRIBUTES clauses in the application definition, you do not need to name explicitly the server or task group to which you want a control characteristic to apply. ACMS provides some defaulting of server and task group names in the application definition.

The following SERVER ATTRIBUTE clause includes both the server and the task group name of the server to which the user name DEPARTMENT applies:

```
SERVER ATTRIBUTE IS
  UTILITY_SERVER : SERVER UTILITY_SERVER IN DEPARTMENT_COBOL_TASK_GROUP;
                   USERNAME IS DEPARTMENT;
END SERVER ATTRIBUTE;
```

In the SERVER ATTRIBUTES clause, there are two phrases: the server name and the task group name. In some cases, one of these phrases can be omitted. If the server has the same name in that application as it has in the task group, you do not have to use the server name phrase. For example:

```
SERVER ATTRIBUTE IS
  UTILITY_SERVER : IN DEPARTMENT_COBOL_TASK_GROUP;
                   USERNAME IS DEPARTMENT;
END SERVER ATTRIBUTE;
```

If the task group containing the server immediately precedes the SERVER ATTRIBUTES clause, you do not have to use the task group phrase. For example:

```
TASK GROUP IS
  DEPARTMENT_COBOL_TASK_GROUP : TASK GROUP FILE IS
      "ACMS$EXAMPLES:DEPRMSCOB.TDB";
END TASK GROUPS;

SERVER ATTRIBUTE IS
  UTILITY_SERVER : SERVER UTILITY_SERVER;
                   USERNAME IS DEPARTMENT;
END SERVER ATTRIBUTE;
```

If you do not specify a task group name in a SERVER ATTRIBUTES clause, the task group name is defaulted from the last task group name in the immediately preceding TASK GROUPS clause. If you name the task group in the SERVER ATTRIBUTES clause, then you do *not* have to place the SERVER ATTRIBUTES clause *after* the TASK GROUPS clause to which it applies.

11.4.11. Positioning SERVER ATTRIBUTES and SERVER DEFAULTS Clauses

The way you place SERVER ATTRIBUTES and SERVER DEFAULTS clauses in an application definition affects how ACMS assigns control characteristics to the servers in the application.

Example 11.8, "Application Using Multiple Server Defaults Clauses" shows an application definition that uses multiple SERVER DEFAULTS clauses to define different server control attributes for the servers in two task groups.

Example 11.8. Application Using Multiple Server Defaults Clauses

```
USERNAME IS PERSONNEL;
SERVER DEFAULTS ARE
  AUDIT;
  USERNAME IS DEPARTMENT;
  LOGICAL NAME
     PERS_FILE  = "ACMS$EXAMPLES:PERSFILE.DAT";
  MAXIMUM SERVER PROCESSES IS 5;
  MINIMUM SERVER PROCESSES IS 1;
END SERVER DEFAULTS;

TASK GROUP IS
   DEPARTMENT_TASK_GROUP: TASK GROUP FILE IS
                          "ACMS$EXAMPLES:DEPART.TDB";
END TASK GROUP;

SERVER DEFAULTS ARE
  USERNAME IS PERSONNEL;
 END SERVER DEFAULTS;

TASK GROUP IS
  PERSONNEL_TASK_GROUP:  TASK GROUP FILE IS
                         "ACMS$EXAMPLES:PERSONNEL.TDB";
END TASK GROUP;

SERVER ATTRIBUTES
  UTILITY_SERVER : DYNAMIC USERNAME;
END SERVER ATTRIBUTES;
END DEFINITION;
```

Because none of the servers in the Department task group is named in a SERVER ATTRIBUTES clause, the defaults defined by the first SERVER DEFAULTS clause apply to all the servers in that task group. Most of those default values also apply to all the servers in the Personnel task group, except that:

- The second SERVER DEFAULTS clause sets up a different default user name for the Personnel servers.
- The SERVER ATTRIBUTES clause defines the Utility server in the Personnel task group as the only one with a dynamic user name.

Any defaults set in a SERVER DEFAULTS clause remain in effect unless changed by a later SERVER DEFAULTS clause. Any control attributes not named in a SERVER DEFAULTS clause retain their ACMS-supplied defaults. You can also override any default value by explicitly assigning an attribute in a SERVER ATTRIBUTES clause.

When you write an application definition, use the order of SERVER DEFAULTS, SERVER ATTRIBUTES, and TASK GROUP clauses that lets you take the best advantage of defaulting. Your goal is always to make the application definition as simple and as clear as possible.

11.4.12. Auditing Servers

ACMS provides an auditing facility to record events occurring when a task is running in a server process. Use the AUDIT subclause to control whether events such as the unexpected stopping of a server process are recorded in the audit trail log. For example:

```
SERVER DEFAULT IS
  AUDIT;
END SERVER DEFAULT;

TASK GROUP IS
  ADMINISTRATION_COBOL_TASK_GROUP : TASK GROUP FILE IS
    "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUP;
```

In this example, the audit trail log records server events for the Administration task group whenever a task in that group is run and at other times. The default value for the AUDIT subclause is NOAUDIT. If you want a record of task events, be sure to include the AUDIT subclause in the SERVER ATTRIBUTES or SERVER DEFAULTS clause. For a list of the events written to the audit trail log, see *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/]. Even if you do not specify the AUDIT subclause, ACMS records all failure statuses.

11.4.13. Enabling Procedure Server Process Dumps

Although a task executes successfully under the ACMS Task Debugger, it can sometimes encounter problems when it is running in the ACMS run-time environment. You can request a server process dump in the event that a server abnormally terminates processing. Using the server process dump, you can trace the location of a software error that occurs while the server is executing in a production environment.

To enable server process dumps, use the SERVER PROCESS DUMP clause in the application definition:

```
SERVER ATTRIBUTES ARE
  SMITHSRV:
    SERVER SMITHSRV IN ACMSTEST_GROUP;
    SERVER PROCESS DUMP;
  END SERVER ATTRIBUTES;
```

If the server process terminates abnormally and you have enabled server dumps, ACMS saves the context of the process at the point where the error occurs and writes it to a dump file.

To analyze the output from a server process dump file, use the command **ANALYZE/PROCESS_DUMP**. For information on using this command, consult *VSI OpenVMS Debugger Manual* [https://docs.vmssoftware.com/vsi-openvms-debugger-manual/].

For more information on requesting server process dumps, see *VSI ACMS for OpenVMS Writing Server Procedures* [https://docs.vmssoftware.com/vsi-acms-writing-server-proc/].

11.5. Controlling Applications

In addition to controlling an application by assigning server and task attributes, you can assign control characteristics to the application itself. These characteristics determine processing for the application including:

- User name for the EXC
- Whether or not application events are audited
- Default directory for the EXC
- Logical names and name tables for the EXC
- Default file name for the application database
- Maximum number of server processes the application can have active at one time
- Maximum number of task instances the application can have active at one time

11.5.1. Assigning an Application Execution Controller User Name

ACMS assigns default values to all user names except the EXC user name. Every EXC has a user name that you assign in the application definition with the APPLICATION USERNAME clause. The execution controller user name determines:

- How the work done by the execution controller process is logged by the OpenVMS ACCOUNTING facility
- The privileges, priority, and quotas assigned to the execution controller

The following application definition assigns the user name PERSONNEL to the execution controller:

```
APPLICATION USERNAME IS
  PERSONNEL;

TASK GROUP IS
  ADMINISTRATION_COBOL_TASK_GROUP : TASK GROUP FILE IS
    "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUP;
```

The user name that you assign to the Application Execution Controller must be a valid OpenVMS user name consisting of 1 to 12 alphanumeric characters, including underscores. The user whose user name you assign to the Application Execution Controller must be an authorized OpenVMS user.

11.5.2. Auditing Applications

ACMS provides the audit trail log to record application events such as application starts and stops, and to write out reports on application events. You use the AUDIT clause to control whether events are recorded in the audit trail log. For example:

```
AUDIT;
```

```
APPLICATION USERNAME IS PERSONNEL;

TASK GROUPS ARE
  DEPARTMENT_COBOL_TASK_GROUP : TASK GROUP FILE IS
      "ACMS$EXAMPLES:DEPRMSCOB.TDB";
  ADMINISTRATION_COBOL_TASK_GROUP : TASK GROUP FILE IS
      "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUPS;
END DEFINITION;
```

In this example, the audit trail log records application events for the Personnel application whenever that application is run. The default value for the AUDIT subclause is NOAUDIT. For a list of the events written to the audit trail log, see *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/]. Even if you do not specify the AUDIT clause, ACMS records all failure statuses.

11.5.3. Assigning Application Default Directories

ACMS assigns a default directory to every Application Execution Controller process directory. The default value comes from the directory that you assign to the user name of an application.

You can assign a default directory for an execution controller if you want to override the default directory assignment because the execution controller needs a different default directory from the one assigned to the application user name in the SYSUAF file. By default, ACMS assigns USERNAME DEFAULT DIRECTORY for the DEFAULT DIRECTORY clause. When the default is in effect, ACMS assigns the execution controller the default directory that is in the SYSUAF entry for the application user name.

If you do not supply full file specifications for task group databases, request libraries, or message files in the application and task group definition, ACMS uses the default directory assigned to the application user name in the SYSUAF file to find them.

You can name a default device and directory with the DEFAULT DIRECTORY subclause. For example:

```
DEFAULT DIRECTORY IS SYS$SAMPLE;
USERNAME IS PERSONNEL;

TASK GROUP IS
  ADMINISTRATION_COBOL_TASK_GROUP : TASK GROUP FILE IS
    "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUP;
```

This definition assigns the device and directory pointed to by the logical name SYS$SAMPLE as the default directory for the execution controller in the Personnel application. You can name a directory specification and device name rather than a logical name, but using logical names lets you change directory locations without redefining and rebuilding the application. You can also use the LOGICAL NAMES subclause to define the name.

11.5.4. Assigning Application Logical Names

It is important to define logical names for an EXC process if that process needs to find task group databases, request libraries, or message files by logical name. Logical name translations must be available to the EXC. There are several ways to make the names available:

- Define the logical names as system logicals.

- Define the logical names as group logicals. Make sure the logical names are available to the group UIC for the EXC process.
- Use the APPLICATION LOGICAL NAMES subclause to define process logical names for the execution controller process.
- Use logical name tables.

Logicals assigned with the LOGICAL NAMES subclause are available only to the execution controller. List each logical name and its equivalent name. For example:

```
APPLICATION LOGICAL NAME IS EMPLOYEE_MESSAGES = "ACMS$SAMPLE:EMPMSG.EXE";
APPLICATION USERNAME IS PERSONNEL;

TASK GROUP IS
  ADMINISTRATION_COBOL_TASK_GROUP : TASK GROUP FILE IS
    "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUP;
END DEFINITION;
```

In this example, the LOGICAL NAMES clause defines the logical name EMPLOYEE_MESSAGES for the file ACMS$SAMPLE:EMPMSG.EXE. If a logical name or equivalent name does not conform to the rules for identifiers, enclose it in quotation marks.

ACMS assigns the logical names SYS$LOGIN, SYS$DISK, and SYS$SCRATCH by default.

For information on using logical name tables, see *Section 11.4.5, "Creating Logical Name Tables for Application Servers"*.

11.5.5. Assigning Application Database Files

When you build an application definition, you can include the name of the application database file with the **BUILD** command. The **BUILD** command translates the CDD version of the definition into the application database file. For example:

```
ADU> BUILD APPLICATION PERSONNEL_APPL1 SYS$SAMPLE:PERSONNEL
ADU>
```

In this example, PERSONNEL_APPL1 is the CDD application object. The application database file is SYS$SAMPLE:PERSONNEL.ADB.

To simplify the **BUILD** command, you can include the default database file name in the application definition instead of putting it on the command line. You name the file in the application definition with the DEFAULT APPLICATION FILE clause. For example:

```
DEFAULT APPLICATION FILE IS PERSONNEL;
USERNAME IS PERSONNEL;
TASK GROUP IS
  ADMINISTRATION_COBOL_TASK_GROUP : TASK GROUP FILE IS
      "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUP;
END DEFINITION;
```

The application database file you are naming for the default is PERSONNEL. The default file type is .ADB. If you do not include a device or directory, ADU uses your default device and directory when you build the application. The default device and directory are those of the process at work when the application is built, rather than those in effect when the application definition is created.

If ADU does not find an application database file on the **BUILD** command line, it looks for the name in the application definition. If you have not used the DEFAULT APPLICATION FILE clause, ADU derives the file name from the full CDD given name of the application, including underscores (_) and dollar signs ($), and a default file type of .ADB, to create the application database file name.

The DEFAULT APPLICATION FILE clause accepts a full file specification so that ADU can control the placement of the application database file. Once you build the application, you must move the application database file to ACMS$DIRECTORY or use the **INSTALL** command to copy the application database into ACMS$DIRECTORY before you can start the application.

11.5.6. Controlling the Number of Server Processes

It is important to control the number of server processes active in your application in order to make the best use of your system resources. Because every application requires a slightly different allocation of system resources, experiment with the number of server processes that is best for your application. For more information on determining the best number of server processes for your application, see *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/].

You can control the number of server processes allowed in an ACMS application from two places in the application definition:

- MAXIMUM SERVER PROCESSES subclause in the SERVER DEFAULTS or SERVER ATTRIBUTES clauses
- MAXIMUM SERVER PROCESSES clause at the application level

For example:

```
DEFAULT APPLICATION FILE IS "ADRMSCAPP.ADB";
MAXIMUM SERVER PROCESSES IS 2;

APPLICATION USERNAME IS PERSONNEL;

SERVER DEFAULTS ARE
 AUDIT;
 MAXIMUM SERVER PROCESSES IS 2;
 MINIMUM SERVER PROCESSES IS 0;
END SERVER DEFAULTS;

TASK GROUP IS
  ADMINISTRATION_COBOL_TASK_GROUP : TASK GROUP FILE IS
      "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUP;

END DEFINITION;
```

In the SERVER DEFAULTS clause, the MAXIMUM SERVER PROCESSES is set to 2. In the MAXIMUM SERVER PROCESSES clause at the application level, the number of server processes is also set to 2.

When an application is started, ACMS checks three values to determine the actual maximum number of server processes:

- Value your system manager sets for the maximum number of OpenVMS processes OpenVMS can create (MAXPROCESSCNT)

- Value of the MAXIMUM SERVER PROCESSES clause at the application level
- Sum of the values of all the MAXIMUM SERVER PROCESSES subclauses for all the servers in the application

ACMS first compares the value of all the MAXIMUM SERVER PROCESSES subclauses with the value of the MAXIMUM SERVER PROCESSES clause at the application level. ACMS takes the smaller value of the two and compares it to the maximum number of OpenVMS processes available. The smaller of these two values is the number assigned for the overall maximum number of server processes in the application.

The number you assign with the MAXIMUM SERVER PROCESSES clause for the application must be greater than the sum of the MINIMUM SERVER PROCESSES subclauses for all the servers in the application.

When you start an application, the EXC determines the smallest value for the maximum server processes. Then it creates a table for each server. The tables can become large enough to affect the size of the EXC, and the size of the EXC affects your system.

You can make sure that ACMS does not use up all of the process slots on your operating system table by setting a MAXIMUM SERVER PROCESSES value in the application that is less than the maximum number of OpenVMS processes allowed on your system.

If the execution controller cannot start a server process because no more OpenVMS processes are available on the system, the execution controller cancels tasks. If there are no available server processes in the ACMS system, the execution controller holds a task until a server process becomes available.

You can experiment with the number of server processes to get an idea of how many server processes are best for your application and system. The default value for MAXIMUM SERVER PROCESSES is unlimited. Try the default value and then decrease the number of server processes. Too many server processes can degrade the performance of your system. Too few may decrease throughput. Set a limit that does not degrade system performance but still lets users complete their work as quickly as possible.

11.5.7. Controlling the Number of Task Instances

The value you define with the MAXIMUM INSTANCES clause controls the number of task instances that can be active at one time. The number of task instances allowed on your system affects the system's performance. For example:

```
DEFAULT APPLICATION FILE IS "ADRMSCAPP.ADB";
MAXIMUM SERVER PROCESSES IS 2;
MAXIMUM TASK INSTANCES IS 25;

TASK GROUPS ARE
  DEPARTMENT_COBOL_TASK_GROUP : TASK GROUP FILE IS
      "ACMS$EXAMPLES:DEPRMSCOB.TDB";
  ADMINISTRATION_COBOL_TASK_GROUP : TASK GROUP FILE IS
      "ACMS$EXAMPLES:ADMRMSCOB.TDB";
END TASK GROUPS;
```

By default, ACMS allows as many task instances as there are task selections. In most cases, you can use the default value, but there are two reasons to define a lower value for MAXIMUM TASK INSTANCES:

- To prevent the EXC from using up its OpenVMS quotas
- To improve performance if a large number of active tasks are causing poor performance

The execution controller assigns quotas to each task from the pool of quotas that it has available. If the number of task instances is greater than the value of MAXIMUM TASK INSTANCES, ACMS cancels additional task selections. To avoid overloading your system and thereby reducing performance, set a reasonable task limit with the MAXIMUM TASK INSTANCES clause. Because the needs of different applications vary, you must experiment with this value to find the best setting for the applications you have.

After you write an application definition, you must store the definition in the CDD and then process the definition, using ADU, to create an application database that ACMS can use at run time.

11.6. Modifying an Active Application

ACMS allows you to modify certain environmental attributes for the following components of an active ACMS application:

- Application attributes
- Server attributes
- Task attributes

Using the **ACMS/MODIFY APPLICATION** command, you can perform the following functions:

- Enable or disable application, server, and task auditing
- Enable or disable tasks
- Adjust the minimum and maximum number of server processes for a particular server
- Adjust server creation and deletion intervals
- Adjust server creation and deletion delays
- Adjust the maximum number of task instances and server processes for an application
- Adjust server monitoring intervals
- Adjust transaction timeout limits

Changes you make with the **ACMS/MODIFY APPLICATION** command affect only the current application and do not permanently affect the ADB. Parameters changed with the **ACMS/MODIFY APPLICATION** command are reset when you restart an application. To change attributes permanently, you must modify and rebuild the application definition.

For more information on modifying active applications, consult *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/].

11.7. Controlling Application Failover

ACMS is capable of keeping applications available in the event of a system failure. The **ACMS/REPROCESS APPLICATION_SPEC** command redirects all subsequent task selections to the application pointed to by the applications specification.

For example, suppose that:

- The application specification payroll points to an application in which tasks are being selected
- You have defined PAYROLL as a logical name for A::PAYROLL

To force subsequent task selections for PAYROLL to go to node B, you must:

1. Redefine PAYROLL as B::PAYROLL
2. Issue the command to redirect future task selections:

```
$ ACMS/REPROCESS APPLICATION_SPEC PAYROLL
```

By using search lists for application specifications, you can provide for automatic change to one or more backup applications in the event of a system failure. When the original system becomes available, you can use the **ACMS/REPROCESS APPLICATION_SPEC** command to change back to the original application.

For more information on application failover, see *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/].

Chapter 12. Defining Menus

After you define the tasks, task groups, and application as described in the previous chapters, you need to write menu definitions to describe how your application looks to users.

You can define two types of entries to display on a menu: tasks and menus. Tasks do the work of an application; menus display tasks and other menus for selection by users. Because a user can select one menu from another menu, you can create a menu hierarchy or tree. There are three parts to creating a menu hierarchy for your application:

- Planning the menu structure of your application
- Deciding whether to use DECforms or TDMS to display each menu in your menu structure
- Writing definitions for menus in the application

You can create and use a menu tree before any applications exist. This feature is useful for early tests of how your menus and menu tree will look to users. Before you write menu definitions, plan the menu structure that best represents the work users need to do with your application.

For each menu in a menu tree, you need to decide whether you want to use DECforms or TDMS to display your menu. You can use a combination of DECforms and TDMS to display the menus in one menu tree.

12.1. Planning the Menu Structure

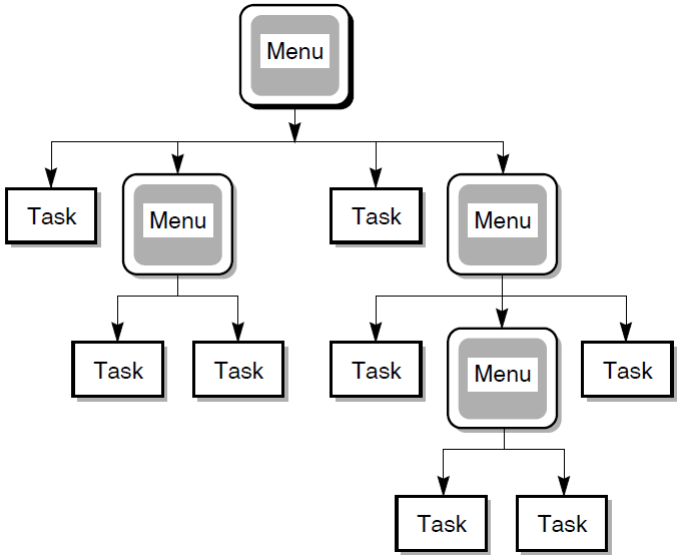
The ACMS menu structure provides a great deal of flexibility for presenting your application to users. ACMS menus are organized in a hierarchical structure, much like the OpenVMS directory structure. A top menu points to menus and tasks. Users can select either a menu or a task to run from the top menu.

You can display the tasks of an application on a single menu or on more than one menu. This choice depends on how you want to present the tasks to users.

One menu or menu tree can point to tasks in several applications. Several menus or menu trees can point to tasks in a single application. Finally, you can create any combination of these two cases.

Figure 12.1, "The ACMS Menu Structure" shows the hierarchical ACMS menu structure.

Figure 12.1. The ACMS Menu Structure



Using a fictional personnel application as an example, suppose you had two kinds of tasks in the application:

- REVIEW_HISTORY and REVIEW_SCHEDULE tasks, which are restricted tasks that deal with employee performance reviews
- Utility tasks, such as an LSEDIT task to run the LSE editor and a DATR task to run the DATATRIEVE procedure DUE

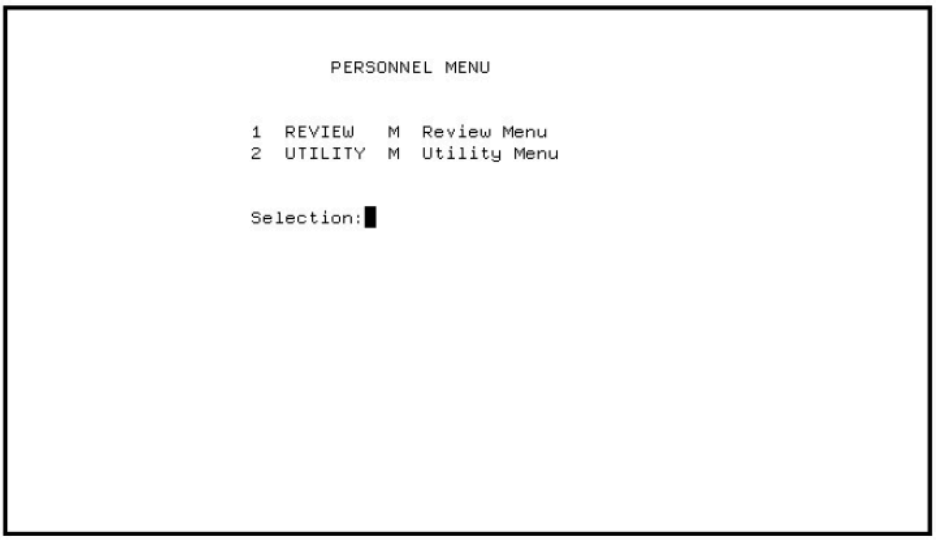
You can create a menu hierarchy that includes three menus: one to display the REVIEW_HISTORY and REVIEW_SCHEDULE tasks, one to display the utility tasks, and one from which the other two menus can be selected. *Example 12.1, "Menu Definition for the Personnel Menu"* shows the menu definition for the last of these menus.

Example 12.1. Menu Definition for the Personnel Menu

```
HEADER IS   "                        PERSONNEL MENU";
ENTRIES ARE
  REVIEW   : MENU IS EXAMPLES_DEFINITIONS.REVIEW_MENU;
             TEXT IS "Review Menu";
  UTILITY  : MENU IS EXAMPLES_DEFINITIONS.UTILITY_MENU;
             TEXT IS "Utility Menu";
END ENTRIES;
END DEFINITION;
```

Figure 12.2, "Personnel Menu" shows the menu that is displayed by the top menu definition after it is processed. The letter M indicates that the menu selection is another menu; if the letter T appears it indicates that the menu selection is a task.

Figure 12.2. Personnel Menu



12.2. Defining Menus

Once you plan a menu structure for your application, you can write menu definitions to describe what the user sees on the menus. In the menu definition you:

- Assign a title, or header, to a menu
- Name tasks and menus to be listed on the menu
- Include descriptive text for each task and menu
- Assign default application files for tasks on the menu
- Assign default menu database files for the build process
- Assign WAIT/DELAY characteristics for tasks
- Reference DECforms control text responses or TDMS requests for customized menus

ACMS provides both a DECforms form and a TDMS request for ACMS to use when displaying menus. ACMS takes the specific information that you enter about a menu (heading, titles of menus and tasks listed, and so on) from your menu database (.MDB) and displays that information in the format of either the DECforms or TDMS request. See *Section 12.2.9, "Defining a Menu Forms Product"* for details on how to determine which forms product ACMS uses.

Both the DECforms and TDMS menu formats allow you to create menus that serve the needs of your users. By specifying menu elements in your menu definition, you can create a menu to suit almost any application. After you define a menu, you use ADU to create a menu database (.MDB) to use in your application. See *Section 12.3, "Processing the Menu Definition"* for an explanation of how to build a menu database.

If you need to use a menu format different from the one provided with either DECforms or TDMS requests, you can modify the standard menu format. See *Appendix A, "Changing the ACMS Menu Format Using VSI DECforms"* and *Appendix B, "Changing the ACMS Menu Format Using TDMS"* for information on modifying menus displayed by DECforms and TDMS requests.

12.2.1. Creating a Title for a Menu

The first step in creating a menu definition is to enter a title for the menu. The HEADER clause lets you create either a 1-line or 2-line title. ACMS displays the title at the top of a menu screen. By default, ACMS leaves the lines at the top of a screen blank. However, it is a good idea to include a title for each menu so your users always know where they are working within the menu structure.

To define a menu title, type the HEADER clause in your source file, putting quotation marks around the title. For example:

```
HEADER IS   "                       PERSONNEL MENU";
```

You center the title by including spaces before the text of the title, inside the quotation marks. End the clause with a semicolon (;). If you include a second line in a HEADERS clause, end the first line with a comma (,) and end the second line with a semicolon (;). For example:

```
HEADER IS   "               HISTORY AND SALARY MENU",
             "              FOR PERSONNEL APPLICATION";
```

Caution

If you are using TDMS to display your menu, do not use tabs in your menu definition. TDMS does not allow tabs; therefore, using tabs can cause a fatal error at run time.

If you are using DECforms to display your menu, however, you can use tabs in your menu definition.

12.2.2. Naming Entries on a Menu

You use the ENTRIES clause to name the entries on a menu and to describe characteristics of each entry. The ENTRIES clause is the only required clause in the menu definition. The descriptions of an entry include the entry name, entry description, and the type of entry.

The ENTRIES clause begins with the keywords ENTRIES ARE and ends with END ENTRIES. Entry names can be up to 10 characters long and should describe the task or menu that they name. An entry name can be enclosed in quotation marks. If your entry name is a quoted string, do not include spaces, tabs, or periods in the name. A colon (:) separates an entry name from the subclauses that define the entry. For example:

```
HEADER IS   "                       PERSONNEL MENU";
ENTRIES ARE
  REVIEW :

  UTILITY:

END ENTRIES;
END DEFINITION;
```

You can define two types of entries in a menu definition: tasks and menus. If you define an entry as a task and a user who has access to it selects that entry, ACMS runs that task. When you define an entry as a menu, and a user selects that entry, ACMS displays another menu. You use either the TASK or MENU subclause in the ENTRIES clause to indicate the type of entry. Every ENTRIES clause must include at least one TASK or MENU subclause.

12.2.3. Naming Menus

You use the MENU subclause to define every menu entry on a menu. In the menu subclause, you include the keywords MENU IS and the CDD path name of the definition for the menu you want to display. For example:

```
HEADER IS   "                        PERSONNEL MENU";
ENTRIES ARE
  REVIEW   : MENU IS EXAMPLES_DEFINITIONS.REVIEW_MENU;
  UTILITY  : MENU IS EXAMPLES_DEFINITIONS.UTILITY_MENU;
END ENTRIES;
END DEFINITION;
```

In this example, the REVIEW entry is a menu whose definition is in the CDD directory EXAMPLES_DEFINITIONS. The menu is named REVIEW_MENU. The second entry in the menu definition is UTILITY. This entry is also a menu that would be found in the CDD directory EXAMPLES_DEFINITIONS. The menu is named UTILITY_MENU.

Both entries in the menu definition are menus. Tasks and menus can be named on the same menus or on different ones. Naming tasks is very similar to naming menus.

12.2.4. Naming Tasks on a Menu

You use the TASK subclause to define every task entry on a menu. In the TASK subclause, you include the keywords TASK IS, the name of the task in the application database, and the **application specification** of the application database file that contains the task. For a more complete discussion of application specifications, see *Section 12.2.7, "Application Specifications and Remote Tasks"*.

If the task is defined with the TASK ATTRIBUTES clause in the application definition, the task name must be the one used in the ATTRIBUTES clause. If the task is not defined in the application definition, the task name must be the one used in the task group definition.

For the application database file name, include only the file name, not a device, directory, or file type specification. For example:

```
HEADER IS   "                     REVIEW MENU";
ENTRIES ARE
  SCHEDULE : TASK IS REVIEW_SCHEDULE IN PERSONNEL;
  HISTORY  : TASK IS REVIEW_HISTORY IN PERSONNEL;
END ENTRIES;
END DEFINITION;
```

In this example, the HISTORY entry is a task named REVIEW_HISTORY in the Personnel application. The SCHEDULE entry is a task named REVIEW_SCHEDULE, running in the Personnel application.

12.2.5. Specifying WAIT or DELAY Action

You can specify the WAIT/DELAY attribute on tasks in the menu definition. The WAIT/DELAY attribute causes the Command Process (CP) to wait or delay before displaying the menu. The WAIT/DELAY attribute specified in the menu definition overrides the WAIT/DELAY value specified for the task in the application database.

```
HEADER IS   "                     REVIEW MENU";
ENTRIES ARE
  HISTORY  : TASK IS REVIEW_HISTORY IN PERSONNEL;
             DELAY;
  SCHEDULE : TASK IS REVIEW_SCHEDULE IN PERSONNEL;
```

```
            WAIT;
END ENTRIES;
END DEFINITION;
```

In this example, the **WAIT** command causes the CP to wait until the user signals to continue, and the **DELAY** command causes the CP to wait 3 seconds. This section describes how to create short descriptions of tasks and menus that are displayed with the entry on an ACMS menu.

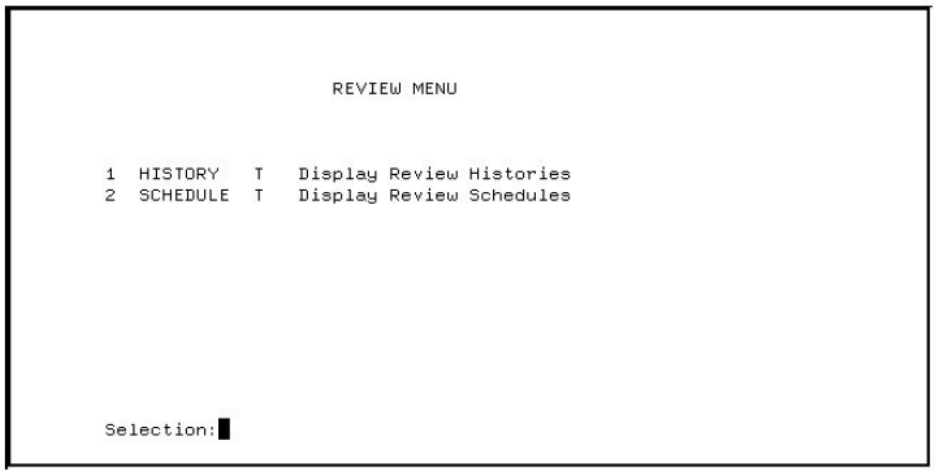
The TEXT subclause lets you create a short description of an entry that is displayed on an ACMS menu. By default, ACMS does not include descriptive text on the screen. However, it is a good idea to include descriptive text for each task and menu in your application to help users identify the item they are selecting, rather than relying just on the name of a selection.

To include descriptive text on a menu, use the TEXT IS subclause, followed by the text itself. The text can be up to 50 characters long, must be enclosed in quotation marks and must end with a semicolon (;). For example:

```
HEADER IS   "                    REVIEW MENU";
ENTRIES ARE
  HISTORY  : TASK IS REVIEW_HISTORY IN PERSONNEL;
             TEXT IS "Display Review Histories";
  SCHEDULE : TASK IS REVIEW_SCHEDULE IN PERSONNEL;
             TEXT IS "Display Review Schedules";
END ENTRIES;
END DEFINITION;
```

For each task on the Review Menu, descriptive text is included to the right of the task selection. *Figure 12.3, "The Review Menu"* shows the menu created by the Review Menu definition.

Figure 12.3. The Review Menu



So far, every menu definition shown in this chapter has named the application for each individual task or menu. The next section explains the menu definition clause that allows you to avoid this repetition.

12.2.6. Naming Default Application Files

It is not necessary to name the application for every task and menu in a menu definition. Instead, you can assign a default application specification in a menu definition. ADU uses the application specification by default when you build the application.

The DEFAULT APPLICATION clause includes the keywords DEFAULT APPLICATION IS and the application specification. When you use the DEFAULT APPLICATION clause, the application name

should not include the device, directory, or file type specification. For example, the menu definition for the Review Menu includes a default application name:

```
DEFAULT APPLICATION IS PERSONNEL;
HEADER IS   "                    REVIEW MENU";
ENTRIES ARE
  SCHEDULE : TASK IS REVIEW_SCHEDULE;
             TEXT IS "Display Review Schedules";
  HISTORY  : TASK IS REVIEW_HISTORY;
             TEXT IS "Display Review Histories";
END ENTRIES;
END DEFINITION;
```

In this example, Personnel is the default application specification for the application. The TASK subclauses in the definition do not need to specify the application.

The application specification that you assign in the TASK subclause overrides the one that you assign with the DEFAULT APPLICATION clause.

12.2.7. Application Specifications and Remote Tasks

You can select a task running in an application on another node in a distributed environment, whether that environment is a local area network, wide area network, or OpenVMS Cluster system. There are two methods for selecting a task in a distributed environment:

- You can use a logical name for the node name or for the application name. Using a logical name means that you do not need to rebuild the application if the application node changes for some reason. For a more complete discussion of remote tasks and application specifications, see *VSI ACMS for OpenVMS ADU Reference Manual* [https://docs.vmssoftware.com/vsi-acms-adu-ref-manual/].

- In your menu definition you can also specify the node that the application is on. If the application and the task are on the same node, the application specification consists of the application name. If the application is on a remote node, the application specification consists of the node name followed by two colons (::) followed by the application name.

  Using the example of two applications, Personnel and Employee, suppose the department using these applications has an OpenVMS Cluster with two nodes, RAVEN and MAGPIE. The Personnel application runs on the node RAVEN; Employee runs on MAGPIE. Rather than duplicate tasks from the Employee application for users on RAVEN, you can provide remote access to Personnel tasks on RAVEN for users on MAGPIE using the menu definition. *Example 12.2, "Example of a Menu with a Remote Task"* shows the menu definition for the node MAGPIE.

Example 12.2. Example of a Menu with a Remote Task

```
HEADER IS   "                    EMPLOYEE MENU";
ENTRIES ARE
  SCHEDULE : TASK IS REVIEW_SCHEDULE IN RAVEN::PERSONNEL;
  EMPLOYEE : TASK IS EMPLOYEE IN EMPLOYEE;
END ENTRIES;
END DEFINITION;
```

Remember that the EMPLOYEE menu definition is on MAGPIE. The SCHEDULE entry is a task named REVIEW_SCHEDULE, in the Personnel application running on node RAVEN. The EMPLOYEE entry is the task named EMPLOYEE in the Employee application on node MAGPIE. The SCHEDULE entry provides remote access to the task named REVIEW_SCHEDULE in the Personnel application on RAVEN.

12.2.8. Naming Default Menu Files

You can assign a default menu database file specification in a menu definition. If you specify a file specification, the Application Definition Utility (ADU) uses the name by default for the menu database created when you build a menu tree.

The DEFAULT MENU FILE clause includes the keywords DEFAULT MENU FILE and the name you assign to the database file. If the file name you assign to the database file does not fit the specifications for an ACMS identifier, enclose the name in quotation marks. By default, ADU assigns the file type .MDB to the menu database file. If you do not include a device or directory specification for the file, ADU uses your current device and directory. For example:

```
DEFAULT APPLICATION IS PERSONNEL;
DEFAULT MENU FILE IS "PERSONNEL.MDB";
HEADER IS   "                       PERSONNEL MENU";
ENTRIES ARE
  REVIEW   : MENU IS EXAMPLES_DEFINITIONS.REVIEW_MENU;
             TEXT IS "Review Menu";
  UTILITY  : MENU IS EXAMPLES_DEFINITIONS.UTILITY_MENU;
             TEXT IS "Utility Menu";
END ENTRIES;
END DEFINITION;
```

In this example, the default menu file is PERSONNEL.MDB in the current default device and directory. When you build this menu, ACMS creates a menu database called PERSONNEL.MDB.

A menu database file assignment you make with the **BUILD** command overrides an assignment you make with the DEFAULT MENU FILE clause. If you do not include a file specification for the menu database with the **BUILD** command, ADU uses the menu database file name assigned with the DEFAULT MENU FILE clause in the menu definition. If you do not name a menu database file either with the **BUILD** command or in the menu definition, ADU derives the file name from the full CDD given name of the menu definition, including dollar signs and underscores.

ADU ignores the DEFAULT MENU FILE clause if the clause is in any menu definition other than the menu definition specified in the **BUILD** command. For more information on how ADU derives the menu database file name from a CDD path name, see *VSI ACMS for OpenVMS ADU Reference Manual* [https://docs.vmssoftware.com/vsi-acms-adu-ref-manual/].

12.2.9. Defining a Menu Forms Product

ACMS uses either DECforms or TDMS to display menus for users. In cases where DECforms or TDMS is not available (if it is not installed on the system, or the user is logged in to an unsupported terminal), ACMS uses OpenVMS QIOs to prompt the user to enter a selection.

If DECforms or TDMS is available, ACMS uses the following information, in the order specified, to determine which forms product will be used to display the menu:

1. If the MDB was built using ACMS Version 3.0 or lower, TDMS will be used.

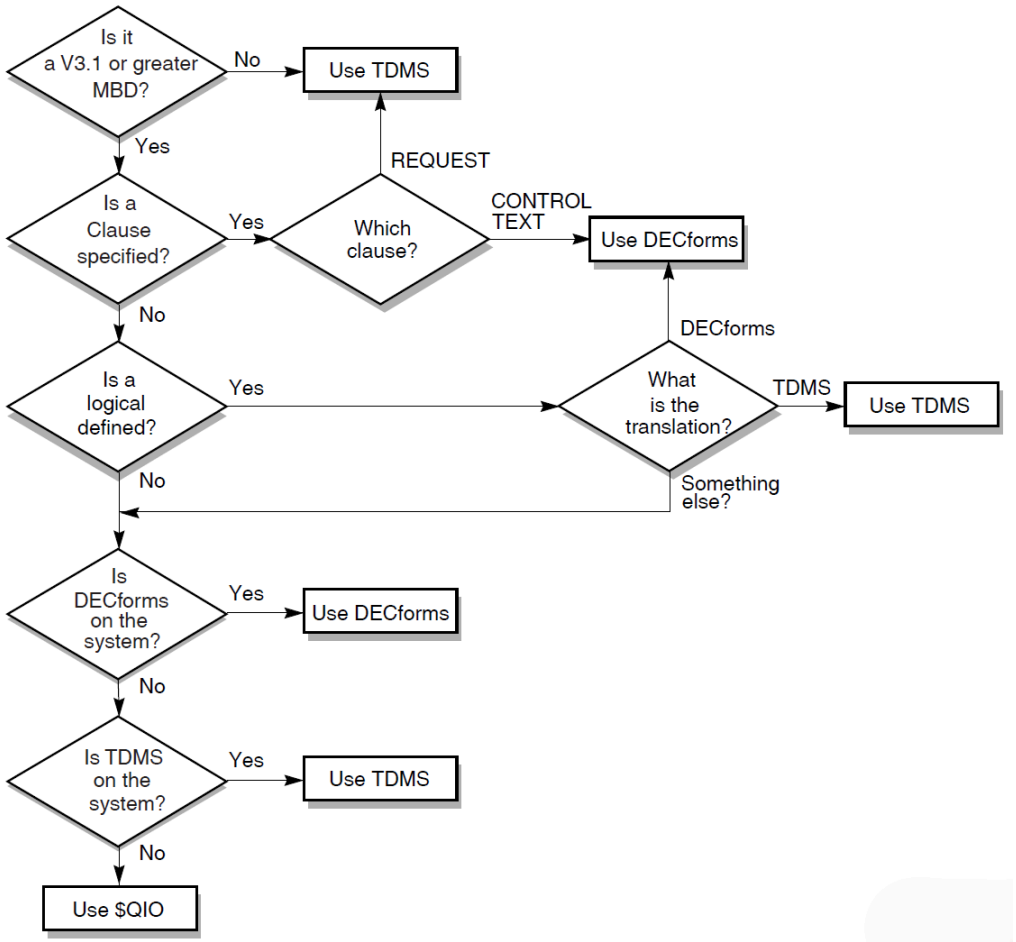
2. You can choose to customize your menu by specifying the REQUEST IS clause for TDMS or the CONTROL TEXT IS clause for DECforms. If one of these clauses is specified in the menu definition, the corresponding forms product will be used. See *Appendix A, "Changing the ACMS Menu Format Using VSI DECforms"* and *Appendix B, "Changing the ACMS Menu Format Using TDMS"* for specific information on customizing ACMS menus displayed by DECforms or TDMS.

3. You can define the logical ACMS$DEFAULT_MENU_FORMS_PRODUCT to be TDMS or DECforms. This logical forces the forms product for all users in the CP.

4. If DECforms is installed on the system, it will be used to display the menu. Otherwise, TDMS will be used.

Figure 12.4, "ACMS Menu Choices" is a diagram of the decisions ACMS makes in selecting DECforms or TDMS to display an ACMS menu.

Figure 12.4. ACMS Menu Choices



If you need to modify the standard ACMS menu format, an advantage to using DECforms is that you do not need to make modifications in both the menu request and in the menu definition. You can make all your changes in the menu form source (.IFDL) file. See *Appendix A, "Changing the ACMS Menu Format Using VSI DECforms"* for information about customizing DECforms menus.

To use DECforms to display menus for versions of ACMS prior to ACMS Version 3.1, you must rebuild MDBs.

If you remove TDMS from your system, ACMS does not automatically use DECforms for menus displayed by the CP. Unless you rebuild your MDB, the task submitter sees only a selection prompt. After rebuilding your MDB, you can use DECforms to display menus.

If you want to use two identical menus, one using DECforms and another using TDMS, in your application, you must build two separate MDBs. You can, however, use DECforms for one menu and TDMS for another menu in the same menu tree.

Following are explanations of how to select DECforms or TDMS as a menu forms product in a menu definition or in a logical:

- Selecting a menu displayed by TDMS in a menu definition

  If your ACMS application requires a menu format different from the ACMS menu format displayed by TDMS, you can use TDMS to modify the menu format. However, this requires a thorough knowledge of TDMS. See *Appendix B, "Changing the ACMS Menu Format Using TDMS"* for more information on how to modify ACMS menus using TDMS.

  To specify a TDMS menu, you must include the REQUEST IS clause in your menu definition. In the REQUEST IS clause, you must include the given name of the request you want ACMS to use for that menu. This given name is not the CDD path name of the request but, rather, the name by which the request is listed in the request library definition. For example:

```
HEADER IS   "                        PERSONNEL MENU";
REQUEST IS PERSONNEL_MENU_REQUEST WITH 12 ENTRIES PER SCREEN;
DEFAULT APPLICATION IS PERSONNEL;
ENTRIES ARE
  REVIEW   : MENU IS EXAMPLES_DEFINITIONS.REVIEW_MENU;
             TEXT IS "Review Menu";
  UTILITY  : MENU IS EXAMPLES_DEFINITIONS.UTILITY_MENU;
             TEXT IS "Utility Menu";
END ENTRIES;
END DEFINITION;
```

  In this example, the REQUEST clause names the TDMS request PERSONNEL_MENU_REQUEST for the menu format. This definition must be in the request library, ACMSREQ.RLB, which is used by ACMS for displaying menus.

  The definition also specifies that the Personnel request display 12 entries per screen. The number in the ENTRIES PER SCREEN phrase must correspond to the number of form fields to which the request writes entry information. The default number of entries for each screen is 16. If you want more or fewer than 16 entries on each screen, you must use the WITH ENTRIES PER SCREEN phrase to define the number of entries. You must also define a request to handle that number of entries.

  If you are not customizing the menu format and you want to use TDMS only to display the menu, specify the request name provided by ACMS:

```
REQUEST IS MENU_REQUEST;
```

- Selecting a menu displayed by DECforms in a menu definition

  If your ACMS application requires a menu format different from the ACMS menu format displayed by DECforms, you can use DECforms to modify the menu format. However, this requires a thorough knowledge of DECforms forms source files. See *Appendix A, "Changing the ACMS Menu Format Using VSI DECforms"* for more information on how to modify ACMS menus using DECforms.

  To specify a DECforms menu that you have customized, you must include the CONTROL TEXT IS clause in your menu definition. In the CONTROL TEXT IS clause, you must include a five-character control text item. In the menu form source (IFDL) file, you must define a control text response with the same name. In the menu definition, for example, you enter the CONTROL TEXT IS clause, followed by a one- to five-character string:

```
HEADER IS   "                        PERSONNEL MENU";
```

```
CONTROL TEXT IS "MYMNU";
DEFAULT APPLICATION IS PERSONNEL;
ENTRIES ARE  REVIEW   : MENU IS EXAMPLES_DEFINITIONS.REVIEW_MENU;
             TEXT IS "Review Menu";
  UTILITY  : MENU IS EXAMPLES_DEFINITIONS.UTILITY_MENU;
             TEXT IS "Utility Menu";
END ENTRIES;
END DEFINITION;
```

In this example, the CONTROL TEXT IS clause names the control text item MYMNU.

You must enter a reference to MYMNU in the form menu source file; for example:

```
Control Text Response "MYMNU"
      Activate MY_PANEL
End Response
```

In this example, the control text response identifies the control text item MYMNU and directs DECforms to display the customized menu panel MY_PANEL to terminal users.

The default number of entries on each screen of menus displayed by DECforms is 16. With menus displayed by DECforms, you specify a different number of entries per screen in the menu form source file. *Appendix A, "Changing the ACMS Menu Format Using VSI DECforms"* contains instructions for customizing menus displayed by DECforms.

If you are not customizing the menu format and you want to use DECforms only to display the menu, specify the control text response provided by ACMS:

```
CONTROL TEXT IS "DFMENU";
```

- Selecting a forms product by defining a Logical

  If you do not use a REQUEST IS or a CONTROL TEXT IS clause in a menu definition to specify a customized menu displayed by TDMS or DECforms, ACMS attempts to translate this logical:

  ```
  ACMS$DEFAULT_MENU_FORMS_PRODUCT
  ```

  You can define this logical to be either TDMS or DECforms and thus specify a forms product for all users on the CP. For example:

  ```
  $ DEFINE/SYSTEM ACMS$DEFAULT_MENU_FORMS_PRODUCT TDMS

   $ DEFINE/SYSTEM ACMS$DEFAULT_MENU_FORMS_PRODUCT DECFORMS
  ```

  This logical name should be defined before you start ACMS. To change the value of the logical after you have started ACMS, stop and restart the ACMS terminal subsystem.

12.3. Processing the Menu Definition

You use the ADU **CREATE** or **REPLACE** command to store menu definitions in CDD. After you store a definition in CDD, you use the ADU **BUILD** command to create a database from the definition that ACMS uses at run time.

As explained in *Chapter 1, "Writing Definitions with ADU"*, processing definitions can be simpler if a **REPLACE** command is included in the source definition file. Include a **REPLACE** command at the top of each menu source definition file. For example, the command for the Personnel menu is:

```
REPLACE MENU PERSONNEL_MENU/LOG/LIST
```

With the **REPLACE** command in each source file, type the at sign (@) command and the name of each menu definition source file in response to the ADU> prompt:

```
$ ADU
ADU> @PERSMEN1.MDF

%ACMSCDU-S-MENREPLAC,
Menu UDISK:[CDDPLUS]EXAMPLES_DEFINITIONS.PERSONNEL_MENU replaced
ADU>
```

The errors you can get when replacing menu definitions are explained in the online error message documentation contained in the file SYS$HELP:ACMSADU.MEM. If you get an error, correct the definition and use ADU to process it again.

Once the corrected definition is stored in CDD, you can use the **BUILD** command to create menu databases. The **BUILD** command includes:

- The MENU keyword.
- The CDD relative or full path name of the menu definition of the top-level menu of your menu tree.
- A menu database file specification (optional). If you do not provide a file name, and if the top-level menu definition does not contain a DEFAULT MENU FILE clause, ADU creates one from the full CDD path name including dollar signs and underscores, and the .MDB file type.

Use the **BUILD** command to produce menu databases. You need to build only the top menu in an application. The other menus are built automatically at the same time. To build the top menu, enter the **BUILD** command:

```
ADU> BUILD MENU PERSONNEL_MENU PERSMEN1.MDB/LOG

%ACMSCDU-I-MENUNAME, Menu named 'REVIEW'
%ACMSCDU-I-LODMENNAM, Loading menu
%ACMSCDU-I-MENPTHLOD, Menu CDD object
'UDISK:[CDDPLUS]EXAMPLES_DEFINITIONS.REVIEW_MENU' loaded
%ACMSCDU-I-PROCMENU, Processing menu 'REVIEW_MENU'
%ACMSCDU-I-PROCTASK, Processing task 'REVIEW_SCHEDULE'
%ACMSCDU-I-PROCTASK, Processing task 'REVIEW_HISTORY'
%ACMSCDU-I-MENUNAME, Menu named 'UTILITY'
%ACMSCDU-I-LODMENNAM, Loading menu
%ACMSCDU-I-MENPTHLOD, Menu CDD object
'CDD$TOP.EXAMPLES_DEFINITIONS.UTILITY_MENU' loaded
%ACMSCDU-I-PROCMENU, Processing menu 'UTILITY_MENU'
%ACMSCDU-I-PROCTASK, Processing task 'EDIT'
%ACMSCDU-I-PROCTASK, Processing task 'DATR'
%ACMSCDU-I-WRITEMDB, Writing MDB
-ACMSCDU-I-BYTESWRIT,   1336 bytes (3 blocks)
```

This command produces a menu database file, PERSMEN1.MDB, containing the run-time version of all three menu definitions.

All submenus in the menu tree must exist in CDD before the **BUILD MENU** command can complete successfully.

Chapter 13. Defining Existing Applications as ACMS Tasks

ACMS applications are made up of tasks. Typically, these tasks are defined using the ACMS Application Definition Utility (ADU), as described in the preceding chapters. However, it is possible that you have existing programs or applications that you did not define using ACMS. Such applications might include OpenVMS images, DATATRIEVE commands and procedures, or DCL commands and command procedures. You can include these existing programs as tasks in your ACMS application.

This chapter explains how to include these types of applications in an ACMS application.

13.1. Defining Single-Step Tasks in ACMS Task Groups

To include existing OpenVMS images, DATATRIEVE commands or procedures, or DCL commands or procedures in your application, you must define them as single-step tasks in the task group definition.

You specify single-step tasks using the TASKS clause in the same way you specify a multiple-step task. However, rather than specifying a CDD pathname for a multiple-step task definition, you specify the executable image, DCL command, or DATATRIEVE procedure that you want to run.

The following sections explain how to include the following as single-step tasks in ACMS task group definitions:

- OpenVMS images
- DCL commands and command procedures
- DATATRIEVE commands and procedures

13.1.1. Defining OpenVMS Images as Tasks

Many applications include programs that perform certain tasks. Suppose you have a program, written in a language that adheres to the OpenVMS Calling Standard, that you want to include as a task in your ACMS application. You use, as part of the TASKS clause, the IMAGE processing subclause to define an OpenVMS image as a task in a task group. For example:

```
TASKS ARE
   EMPLOYEE :  PROCESSING IMAGE "SYS$SAMPLE:EMPLOYEE.EXE";
           .
           .
           .
```

The name you assign to the task can contain up to 31 alphanumeric characters, dollar signs, and underscores, but no embedded blanks. In this example, the task named EMPLOYEE runs an OpenVMS image with the file specification SYS$SAMPLE:EMPLOYEE.EXE. If you do not specify a default device and directory, ACMS uses the current default device and directory by default. You begin the IMAGE subclause with the keyword PROCESSING. Follow the keyword with IMAGE and, in quotation marks, the image name. End the subclause with a semicolon (;).

13.1.2. Defining DCL Commands and Command Procedures as Tasks

It is common for applications to include editing and mail functions that users can select and run. For example, you may want to include the DCL commands **EDIT** and **MAIL** as tasks in your ACMS application. You use the DCL COMMAND subclause to describe tasks that consist of DCL commands or command procedures.

```
TASKS ARE
  EMPLOYEE :  PROCESSING IMAGE "SYS$SAMPLE:EMPLOYEE.EXE";
  EDIT     :  PROCESSING DCL COMMAND "$EDIT/EDT 'P1'";
  MAIL     :  PROCESSING DCL COMMAND "$MAIL";
          .
          .
          .
```

When a user selects the EDIT task, ACMS invokes the **EDIT/EDT** command. When selecting the task, the user can supply a string that ACMS passes to the task as the parameter, P1. For the **EDIT** command, this parameter is the name of the input file for the editing session. The MAIL task invokes the DCL **MAIL** command, allowing the user to read and send mail.

When you use the DCL COMMAND subclause, precede the subclause with the PROCESSING keyword. Be sure to enclose the command in quotation marks ("") and begin the command with the dollar sign ($). You must also end the subclause with a semicolon (;).

13.1.3. Defining DATATRIEVE Commands and Procedures as Tasks

If you have DATATRIEVE commands and procedures that you want to include in your ACMS application, use the DATATRIEVE COMMAND subclause to name them in a task group definition. For example, suppose you have a DATATRIEVE procedure that produces a report outlining all the performance reviews that are due for a department. The following TASKS clause names the DUE task that runs a DATATRIEVE procedure.

```
TASKS ARE
  EMPLOYEE :  PROCESSING IMAGE "SYS$SAMPLE:EMPLOYEE.EXE";
  EDTR     :  PROCESSING DCL COMMAND "$EDIT/EDT 'P1'";
  MAIL     :  PROCESSING DCL COMMAND "$MAIL";
  DUE      :  PROCESSING DTR COMMAND IS
                "DISK1:[CDDPLUS]ACMS$DIR.ACMS$EXAMPLES_RMS.DUE";
END TASKS;
```

The DUE task uses a DATATRIEVE procedure stored in the directory. You must enclose the command string in quotation marks ("") and end the subclause with a semicolon (;).

When you use the DTR COMMAND processing subclause, ACMS uses the image DTR32 by default. If you want to use another image, you must define DTR32 as a logical that points to the DATATRIEVE image you want to run.

Once you have named all the tasks for a TASKS clause, use the END TASKS keywords followed by a semicolon (;). You can use more than one TASKS clause in a single task group definition and can include one or more tasks in each TASKS clause.

After deciding which tasks to include in a task group, you must define servers to handle processing for those tasks.

13.2. Defining Servers to Handle Processing

Servers handle the processing work for the tasks in an application. There are two kinds of servers: procedure servers and DCL servers. Procedure servers handle calls to subroutines and are typically used for processing steps of multiple-step tasks defined with ADU. DCL servers handle the processing work for tasks that run OpenVMS images, DCL commands or command procedures, and DATATRIEVE commands or procedures.

If you are including single-step tasks in your ACMS application, you must define one or more DCL servers to handle the processing work for those tasks. If the task group already includes a DCL server, that server may be able to handle the work for any tasks you are including. However, you may need to name one or more additional servers to handle the processing work for tasks you add to the task group.

You can use the SERVERS clause and its subclauses to name and describe servers for the tasks in a task group. The syntax for the server subclauses is similar to the syntax of the SERVER ATTRIBUTES clause you use in an application definition. For example:

```
SERVER IS
  EMPLOYEE_SERVER : DCL PROCESS;
END SERVER;
```

Here the SERVERS clause names one server, EMPLOYEE_SERVER. The server subclause DCL PROCESS indicates that the server is a DCL server rather than a procedure server. The DCL PROCESS subclause is the only required subclause of the SERVERS clause.

You can also define other server attributes in the SERVERS clause such as the DYNAMIC USERNAME and USERNAME OF TERMINAL USER. However, because these are control attributes, they can be overridden in the SERVER ATTRIBUTES clause of the application definition. See *Chapter 11, "Defining Applications"* for more information.

To enable terminal users to receive their own mail, edit their own files, or use DATATRIEVE against files in their own directories, you must define a dynamic user name for the server, EMPLOYEE_SERVER. When you use the DYNAMIC USERNAME subclause to define a server process, ACMS changes the user name, UIC, and default directory of the task to those of the user selecting a task processed by that server. *Chapter 11, "Defining Applications"* explains in more detail how to use the DYNAMIC USERNAME subclause.

There are two ways to assign servers to tasks in a task group. One way is to position the TASKS clause under the SERVERS clause containing the servers you want the tasks to use. In this case, the tasks in the TASKS clause will be assigned the server defined above them in the SERVERS clause, which is the default server in effect.

In *Example 13.1, "A Task Group Definition"*, the EMPLOYEE_SERVER is assigned to the EMPLOYEE, EDIT, MAIL, and DUE tasks because the TASKS clause follows the SERVERS clause in the task group definition.

Example 13.1. A Task Group Definition

```
REPLACE GROUP ADMINISTRATION_GROUP
 DEFAULT TASK GROUP FILE IS "ACMS$EXAMPLES:ADMRMSCOB.TDB";

 SERVER IS
   EMPLOYEE_SERVER : DCL PROCESS;
                     DYNAMIC USERNAME;
 END SERVER;
```

```
 TASKS ARE
   EMPLOYEE :  PROCESSING IMAGE "SYS$SAMPLE:EMPLOYEE.EXE";
   EDIT     :  PROCESSING DCL COMMAND "$EDIT/EDT 'P1'";
   MAIL     :  PROCESSING DCL COMMAND "$MAIL";
   DUE      :  PROCESSING DTR COMMAND IS
                "DISK1:[CDDPLUS]ACMS$DIR.ACMS$EXAMPLES_RMS.DUE";
 END TASKS;
 END DEFINITION;
```

The other way to define servers for tasks is to name specific servers for specific tasks in the TASKS clause. For example:

```
TASKS ARE
   EMPLOYEE :  PROCESSING IMAGE "SYS$SAMPLE:EMPLOYEE.EXE"
                 IN EMPLOYEE_SERVER;
   EDIT     :  PROCESSING DCL COMMAND "$EDIT/EDT 'P1'"
                 IN EMPLOYEE_SERVER;
   MAIL     :  PROCESSING DCL COMMAND "$MAIL"
                 IN EMPLOYEE_SERVER;
   DUE      :  PROCESSING DTR COMMAND IS
                 "DISK1:[CDDPLUS]ACMS$DIR.ACMS$EXAMPLES_RMS.DUE"
                   IN EMPLOYEE_SERVER;
END TASKS;
```

In the TASKS clause, EMPLOYEE_SERVER is individually assigned to the EMPLOYEE, EDIT, MAIL, and DUE tasks in the IN SERVER phrase of the TASKS clause in the processing subclause for each task.

The task group definition in *Example 13.1, "A Task Group Definition"* lists only tasks that use a DCL server and that were not defined using ADU. However, you can include ACMS multiple-step tasks as well as definitions for already existing tasks in a single task group definition.

13.3. Using the Task Group in an Application

Once you have included existing tasks in a task group, you must consider the effect of those tasks and that task group on the application. You need to change the application definition if:

- You want to change control attributes for tasks or servers named in the modified task group
- You created a new task group definition and must include the name of that group in the TASK GROUPS clause of the application definition

Whether or not you make changes to the application definition, you must rebuild it to create a run-time application database that includes the new tasks. You must also:

- Add new tasks to existing menu definitions or create new menu definitions
- Rebuild modified menu definitions and build new menu definitions

Chapter 14. Using the ACMS Request Interface

ACMS provides great flexibility in collecting user input for processing in an application. You can use DECforms, TDMS, the ACMS Request Interface, the ACMS Systems Interface, or a combination of these tools to capture data for use in an ACMS application. This chapter discusses the ACMS Request Interface and how to use it in an ACMS application.

The examples shown in this chapter are taken from the request interface examples located in the SYS$COMMON:[SYSHLP.EXAMPLES.ACMS.RI] directory, which the logical ACMS$RI_EXAMPLES points to. For ease of use, the components of the ACMS$RI_EXAMPLES directory are listed in *Appendix E, "Request Interface Kit Components"*.

14.1. Overview of the ACMS Request Interface

The ACMS Request Interface (RI) provides an alternate method of doing request I/O from an ACMS task during processing. You can use a combination of DECforms I/O, TDMS Request I/O and/or the RI, Stream I/O, or Terminal I/O in different tasks within the same application. You can use a combination of TDMS Request I/O and the RI in a single task definition on a per-request-library basis. However, you cannot use the RI in conjunction with DECforms I/O or Stream I/O in the same task.

While DECforms and TDMS limit you to collecting user input from DECforms- and TDMS-supported terminals, the RI allows use of other I/O methods such as FMS, SMG, third-party forms products, OpenVMS mailboxes, DECnet task-to-task communication, and OpenVMS QIOs. Using the RI allows you to take advantage of several interactive facilities that are not available with TDMS or DECforms:

- 3270 terminal support
- Multiwindow support
- Graphics support

The RI provides the flexibility of choosing an interface to ACMS without impacting the multiple-step task structure or ease-of-maintenance features inherent in an ACMS application. The RI simply provides a mechanism for executing **user-written request procedures (URPs)** that perform their own I/O in place of TDMS requests. You write a URP in any OpenVMS-supported, high-level language and include the necessary facility calls (for example, FMS, SMG, QIO) for interfacing ACMS with the preferred device.

ACMS multiple-step tasks call URPs from a task definition using the same syntax necessary to call TDMS requests. The RI translates a logical name and uses the resulting translation to determine if it should process a TDMS request or execute a URP from a shareable image.

ACMS supplies a multi-user agent program called the command process (CP), through which DECforms and TDMS terminal users can access ACMS tasks. With the RI, you use a single-user agent to access ACMS tasks. ACMS supplies an agent called ACMS$RI_AGENT that you can use to select tasks that use the RI. Either use the ACMS-supplied RI agent, or, if an application requires it, write a customized, single-threaded RI agent. Code an RI agent in any high-level language that adheres to the OpenVMS calling standard.

Never use the RI if an application requires asynchronous processing or multithreaded processing (one process handling the needs of several users). Instead, you must use the SI services to write a multithreaded agent.

The Request Interface consists of four major components:

- Request Interface agent

  You use an RI agent in place of the ACMS-supplied command process. The RI agent enables the RI so that you can interface non-TDMS devices with ACMS. Note that the standard ACMS menu interface is available only with CP. However, the ACMS-supplied RI agent program ACMS$RI_AGENT allows you to develop your own menu interface.

- User-written request procedures (URPs)

  The RI calls URPs to perform the exchange work for an ACMS task. A URP is a procedure that replaces a TDMS request in an exchange step and is responsible for performing the I/O to the user's device. A group of URPs is linked together into a shareable image that replaces a TDMS request library (.RLB). You can write URPs in any high-level language; include the appropriate facility calls (for example, FMS, SMG, QIO) to interface ACMS with the preferred device.

- ACMRRSHR

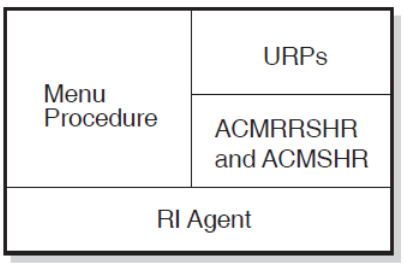
  This ACMS shareable image is responsible for calling a TDMS request or a URP based on the translation of an ACMS$RI_LIB logical name. You can, optionally, define an ACMS$RI_LIB logical to point to a TDMS request library (RLB) or to a URP shareable image (EXE). At run time, ACMRRSHR translates the ACMS$RI_LIB logical to determine if the agent should or perform TDMS or RI I/O during the exchange step.

- Menu interface

  You use this component to develop a menu interface that can be processed by the RI ACMS$RI_AGENT program. For example, you can develop a routine that uses FMS calls to display a menu on an FMS-supported device and to accept a task selection from the user.

Figure 14.1, "Request Interface Run-Time Components" shows the four RI run-time components and their relationship with ACMS.

Figure 14.1. Request Interface Run-Time Components



This chapter describes these components in more detail and explains how to incorporate the Request Interface in an ACMS application.

14.2. The Request Interface and the ACMS Run-Time System

The ACMS Command Process (CP) uses DECforms or TDMS to interact with the terminal user. If you use the RI, you cannot use the CP; you must, instead, use either the ACMS-supplied RI agent or develop an RI agent that does the following:

1. Signs in the user to ACMS
2. Enables the Request Interface
3. Repeatedly prompts the user for task selection information and then calls the task, until the user wants to exit ACMS
4. Disables the Request Interface
5. Signs the user out of ACMS

Note that the URPs that ACMS calls to perform the exchange I/O are not linked into the RI agent image. URPs are called by the ACMRRSHR shareable image, not by the RI agent program.

This section describes how the RI interacts with the ACMS run-time system and the impact it has on ACMS system performance.

When you use the RI (either locally or with distributed applications), the RI agent process always handles the I/O defined in exchange steps. Since the RI is a synchronous interface, it can be used only in a single-threaded agent. Therefore, users must each have their own process that executes the RI agent. Contrast this with the ACMS-supplied CP agent that is a multithreaded, asynchronous agent that can support multiple users in a single process.

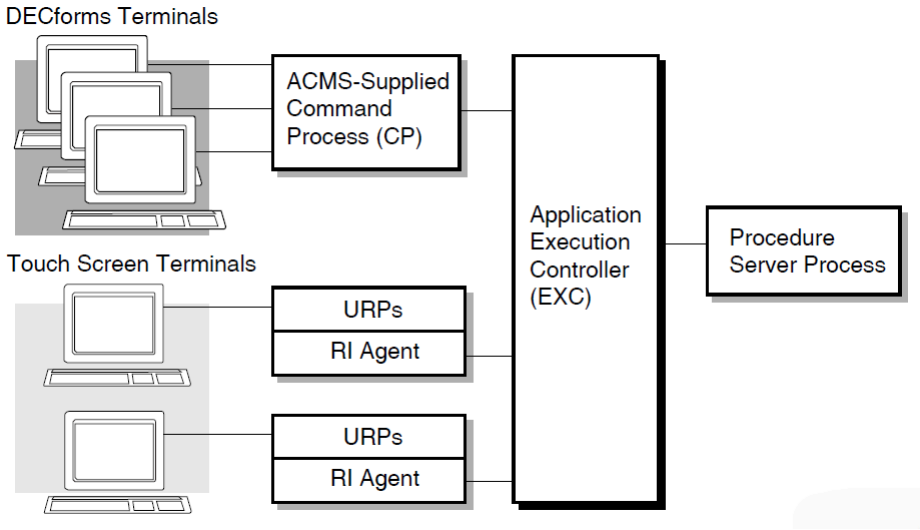
Figure 14.2, "Request Interface Mode" shows one multithreaded CP agent and two single-threaded RI agents interfacing with the same ACMS application execution controller.

As *Figure 14.2, "Request Interface Mode"* shows, an ACMS system can include more than one agent process. Also, a single Application Execution Controller (EXC) can pass request information and handle flow control and scheduling for different types of agent programs. The EXC is a multithreaded process for flow control and scheduling. It interprets the definitions of the tasks in an application. The same performance benefits are realized from ACMS on the processing end (back end) of the system regardless of the I/O method (DECforms, TDMS, or RI) used.

An application can lose performance when using the RI on the front-end system because of the synchronous nature of the Request Interface. Since a single CP agent process supports multiple users, it uses less system resources than a single-user RI agent, where individual users must have their own process. For example, 40 users each running their own copy of a single-user RI agent will often use more memory than two CP agent processes, each handling 20 users.

If an application requires a multithreaded and asynchronous agent, you must use the ACMS Systems Interface to develop an SI agent; you cannot use the RI for multithreaded, asynchronous processing. However, you cannot use asynchronous processing if the interface you want to use is synchronous (such as FMS or SMG). See *VSI ACMS for OpenVMS Systems Interface Programming* [https://docs.vmssoftware.com/vsi-acms-sys-interface-prog/] for more information about the Systems Interface.

Figure 14.2. Request Interface Mode



14.3. Defining Tasks and Task Groups

You define a task the same way whether the task calls TDMS requests or URPs. When you want existing TDMS applications to take advantage of the RI, you do not need to change any definitions in those applications. In fact, you can enable or disable the RI on a per-request-library and/or user-by-user basis, letting an exchange step in a task use TDMS requests for one user and the RI for another user.

ACMS determines whether to call TDMS requests or URPs at run time. If you use the Request Interface, you can define a logical name that ACMS translates to determine whether to use a TDMS request or a user-request procedure to do the work of an exchange step.

At run time, ACMS does the following to determine if it should call a TDMS request or a user-request procedure:

1. At application startup time, the EXC attempts to open library files with both .RLB and .EXE file types. If the EXC cannot open a library with a file type of .RLB, it logs an error message and the application does not start. However, if the EXC cannot open a library file with the file type of .EXE, the application will start. This is because EXC needs .RLB files to perform local TDMS request I/O, but it does not need .EXE files.

2. When the task step begins executing, the EXC passes the name of the TDMS request library (.RLB) file or the user request procedure (.EXE) shareable image to the Request Interface in the RI agent (ACMRRSHR component).

3. If you have defined the ACMS$RI_LIB_library-name logical name, the RI translates the logical and uses the file type of the resulting translation to determine whether or not to process a request from a request library or a user-request procedure from a shareable image. Library-name is the file-name portion of the request library or shareable image file specification named in the task group definition.

   In other words, the ACMS$RI_LIB_library-name logical overrides the request library named in the task group definition. If no logical name exists, the RI uses the file specification passed by the EXC. *Section 14.3.3, "How and When to Use the* ACMS$RI_LIB *Logical Name"* describes how to define the ACMS$RI_LIB logical name.

   - If the ACMS$RI_LIB logical name translation is a file with an .RLB file type, the Request Interface processes a TDMS request from the named request library.

- If the ACMS$RI_LIB logical name translation is a file with an .EXE file type, the RI calls the user-request procedure in the named shareable image file.

This run-time determination means that your application has full front-end independence; the EXC need not be aware of the front-end interface used by the application.

14.3.1. Task Definition

The task definition in *Example 14.1, "Simple Inquiry Task"* is an example of a simple inquiry task. This task definition is generic in that it can be used to call a TDMS request or a URP in a shareable image. This discussion provides the task definition for the sake of example. This example is part of the ACMS-supplied example that is provided in the ACMS$RI_EXAMPLES directory. It is also referred to in several subsequent sections.

The task definition consists of two exchange steps and a processing step. In this task, the user enters a customer number and presses **Return**. The task uses the customer number to read the customer data file and retrieve the appropriate record, displaying that record on the screen.

Example 14.1. Simple Inquiry Task

```
REPLACE TASK RI_INQ_TASK
DEFAULT SERVER IS RI_SERVER;
WORKSPACES ARE RI_EMPLOYEE_RECORD;
BLOCK WORK
   EX1:
       EXCHANGE
       REQUEST IS RI_INQ_REQUEST IN RI_REQUEST_LIBRARY1
          USING RI_EMPLOYEE_RECORD;
   SP1:
       PROCESSING
       CALL RI_GET_PROCEDURE IN RI_SERVER
           USING RI_EMPLOYEE_RECORD;
   EX2:
       EXCHANGE
       REQUEST IS RI_INQ_DISP_REQUEST IN RI_REQUEST_LIBRARY2
          USING RI_EMPLOYEE_RECORD;
END BLOCK WORK;
END DEFINITION;
```

In this example, the task calls a request in a library identified as REQUEST_LIBRARY1. Note that this does not indicate whether ACMS should use a TDMS request or a URP. This decision is not made until run time, when the RI calls a TDMS request or a URP depending on how the library is defined in the group and whether or not an ACMS$RI_LIB logical has been defined for this library.

In a single task, one exchange step can call a URP to do terminal I/O while another can use a TDMS request to do terminal I/O. ACMS passes workspaces to URPs in the same way that it passes workspaces to routines in procedure servers.

See *Chapter 10, "Defining Task Groups"* for more information on defining a task.

14.3.2. Defining a Task Group

When defining a task group whose tasks call only URPs, you can use the REQUEST LIBRARY IS clause to name the shareable image containing URP procedures. *Example 14.2, "Task Group Definition"* shows an example of a task group definition. Alternatively, you can name an .RLB file and use an

ACMS$RI_LIB logical name to point to a shareable image at run time. See *Chapter 10, "Defining Task Groups"* for more information on defining a task group.

Example 14.2. Task Group Definition

```
REPLACE GROUP RI_PASSED_GROUP
REQUEST LIBRARY ARE  "ACMS$RI_EXAMPLES:RI_REQUEST_LIBRARY1.EXE"
                          WITH NAME RI_REQUEST_LIBRARY1,
                     "ACMS$RI_EXAMPLES:RI_REQUEST_LIBRARY2.RLB"
                          WITH NAME RI_REQUEST_LIBRARY2;

DEFAULT TASK GROUP FILE
    IS "ACMS$RI_EXAMPLES:RI_PASSED_GROUP.TDB";

   SERVER IS
       RI_SERVER:
          PROCEDURE SERVER IMAGE IS
                  "ACMS$RI_EXAMPLES:RI_SERVER.EXE";
          INITIALIZATION PROCEDURE IS RI_INIT_PROCEDURE;
          TERMINATION PROCEDURE IS RI_TERM_PROCEDURE;
          PROCEDURES ARE       RI_ADD_PROCEDURE,
                               RI_GET_PROCEDURE;
   END SERVER;

   TASK IS
       RI_ADD_TASK     : TASK DEFINITION IS RI_ADD_TASK;
       RI_INQ_TASK     : TASK DEFINITION IS RI_INQ_TASK;
   END TASK;
END DEFINITION;
```

Notice that this task group names both a URP shareable image file and a request library; the tasks in the group can use both user request procedures and TDMS requests.

However, be aware that you cannot use CP to select a task that uses a request in a library that is defined as an .EXE file. This is because CP cannot call URPs and does not translate ACMS$RI_LIB logical names in order to determine if an .RLB file is also available.

Consider the steps ACMS takes when processing the task and task group shown in *Example 14.1, "Simple Inquiry Task"* and *Example 14.2, "Task Group Definition"*:

1. At application startup time, the EXC determines the file type of the request libraries (.RLB or .EXE). It opens the ACMS$RI_EXAMPLES:RI_REQUEST_LIBRARY2.RLB file but does not open the ACMS$RI_EXAMPLES:RI_REQUEST_LIBRARY1.EXE file.

   If there is an ACMS$RI_LIB logical name that redefines the .EXE file to be an .RLB file, the Request Interface opens that request library at run time. See *Section 14.3.3, "How and When to Use the* ACMS$RI_LIB *Logical Name"* for information on defining the ACMS$RI_LIB logical name.

2. When the task step begins executing, the EXC passes the following names to the RI agent:

   - RI_REQUEST_LIBRARY1
   - RI_REQUEST_LIBRARY2

     The image activation for a shareable image (.EXE) occurs only the first time a URP in that image is needed. After that, the image stays active; only a call to the procedure is necessary.

3. The Request Interface does the following:

- Translates the logical name ACMS$RI_LIB_RI_REQUEST_LIBRARY1 and uses the file type of the resulting translation to determine whether to call a user request procedure from a shareable image or a request from a TDMS request library. In this example, if no logical name has been defined, the Request Interface calls the RI_INQ_REQUEST user request procedure from the RI_REQUEST_LIBRARY1.EXE shareable image file.
- Translates the logical name ACMS$RI_LIB_RI_REQUEST_LIBRARY2 and uses the file type of the resulting translation to determine whether to call a user request procedure from a shareable image or a request from a request library. In this example, if no logical name has been defined, the Request Interface calls the RI_DISP_INQ_REQUEST request from the RI_REQUEST_LIBRARY2.RLB request library.

14.3.3. How and When to Use the ACMS$RI_LIB Logical Name

The TDMS request libraries (.RLB) files or the URP (.EXE) shareable image files can be defined and referenced in the REQUEST LIBRARY IS clause in the task group definition or with the ACMS$RI_LIB_libraryname logical. In either case, the ACMS$RI_LIB logical name can replace or override whatever is defined in the task group definition.

The ACMS shareable image, ACMRRSHR (in the RI agent process), translates the ACMS$RI_LIB logical name at run time to determine whether the task should use TDMS or a URP. Define the ACMS$RI_LIB logical name in any logical name table that is accessible by the RI agent program. For example, defining it as a process logical name means that only a specific user has access to the request library or shareable image defined by the logical. However, defining it as a group logical name means that all users with the same UIC group can have access to the request library or shareable image file defined by the logical.

You use the **DEFINE** command to define the logical name ACMS$RI_LIB_file-name for a shareable image or request library. It is important that the request library that is referenced by the ACMS$RI_LIB logical name has the same library name as defined in the task group definition, and it must include the .RLB or .EXE file extension:

```
$ DEFINE ACMS$RI_LIB_RI_REQUEST_LIBRARY1 -
_$ ACMS$RI_EXAMPLES:RI_REQUEST_LIBRARY1.EXE/GROUP
```

This example defines a group logical for the shareable image named RI_REQUEST_LIBRARY1.EXE. When defining the ACMS$RI_LIB logical, provide the full file specification. Otherwise, ACMS assumes the file is located in SYS$SHARE.

If you use TDMS applications with the RI, you do not need to make any modifications to existing task, task group, or application definitions. However, try to avoid misleading other users by what you have defined in the task definition. Consider the following options to improve the readability of a task group definition:

- When defining a task whose exchange steps call only URPs, name the URP executable images (.EXE) in the task group.
- When defining a task whose exchange steps call only TDMS, name the TDMS request libraries (.RLB) in the task group.
- When defining a task whose exchange steps call TDMS requests and also call URPs, name the TDMS request library in the task group, and use the ACMS$RI_LIB logical name to point to URP executable images where necessary.

Remember that ACMS makes the final determination at run time to call a TDMS request or a URP. You can use the ACMS$RI_LIB logical to override what is defined in the task group definition. Use this logical to enable or disable the RI on a user-by-user basis so that you can have one person use TDMS and another use a URP in the same ACMS application. This makes the RI attractive for testing an application because you can use logicals to redefine the RLB that is hard-coded in the task group definition.

14.4. Writing User Request Procedures

Write user request procedures, URPs, to perform I/O for ACMS tasks in place of TDMS requests. You can write a URP in any OpenVMS-supported high-level language and include the necessary facility calls (for example, FMS, SMG, QIO) that allow you to interface ACMS with the appropriate device.

A URP can use all the facilities available to a normal program and extend the terminal I/O capabilities of the ACMS application without impacting the task definition or task group definition. Use the following guidelines when writing RI procedures:

- The name of the URP must correspond exactly to the name used for the request in the task definition. For example, the function name in BASIC or FORTRAN or a COBOL PROGRAM-ID must match the request name in the task definition. In *Example 14.1, "Simple Inquiry Task"*, the URP request name is RI_INQ_REQUEST.
- The workspaces named in the URP parameter list must be in the same order as those listed for the exchange step in the task definition.
- As an option, include an initialization procedure in an RI shared image. If you provide one, ACMS calls this procedure the first time it calls a URP in the shared image. For example, you can use an initialization procedure to open a channel to a device or open an FMS form file.
- You can also optionally provide a cancellation procedure in an RI shared image. If you provide one, ACMS will call it if it must cancel a task that is calling a URP in the shared image at the time of the cancellation.
- Considerations for the URP shareable image:
  - Link the URP into a shared image and make sure the image is position independent.
  - Set the protection of the shareable image file to have read and execute access.

  The RI agent activates the URP shareable image only once, and it remains mapped into the agent's memory until the agent image exits.
- Each URP procedure must return a status value or you will receive the message "Error processing request interface call".
- ACMS only caches .RLB files that are defined in the task group. It does not cache .RLB files defined by an ACMS$RI_LIB logical. ACMS never caches RI .EXE files.

Example 14.1, "Simple Inquiry Task" and *Example 14.2, "Task Group Definition"* depict a task and a task group in which the first exchange step in the task calls a URP. *Example 14.3, "FORTRAN User Request Procedure"* shows RI_INQ_REQUEST, a FORTRAN URP that displays the inquiry form and prompts the user to enter the customer number.

Example 14.3. FORTRAN User Request Procedure

```
!********************** RI_INQ_REQUEST ******************
```

```
!
       INTEGER FUNCTION RI_INQ_REQUEST (DATA_REC)
       INCLUDE '($IODEF)'              !OpenVMS I/O definitions
!
!   Declaration of variables
!
!       Declare external routines
!
        INTEGER               SYS$QIOW
!
!       Declare the workspaces that will be passed as parameters
!
        STRUCTURE /RI_EMPLOYEE_RECORD/
          INTEGER*4   EMPLOYEE_ID
          CHARACTER*10 EMPLOYEE_FIRST_NAME
          CHARACTER*10 EMPLOYEE_LAST_NAME
        END STRUCTURE
        RECORD /RI_EMPLOYEE_RECORD /DATA_REC
!
!       Declare the IOSB to be used in the OpenVMS QIO system service
!
        STRUCTURE /IOSTAT_BLOCK/
          INTEGER*2   IOSTAT
          INTEGER*2   TERM_OFFSET
          INTEGER*2   TERMINATOR
          INTEGER*2   TERM_SIZE
        END STRUCTURE
        RECORD /IOSTAT_BLOCK/ IOSB
!
!       Declare input channel number to be shareable
!       by all User Request Procedures (URP).
!
        INTEGER*2      INPUT_CHAN
       COMMON /INPUT_CHANNEL/INPUT_CHAN
!
!       Declare local variables
!
       INTEGER*4       CODE,STATUS

        CHARACTER*1    ESC
        DATA           ESC /27/
        CHARACTER*3    CLEAR
        DATA           CLEAR /'[2J'/
        CHARACTER*7    POSITION
        DATA           POSITION /'[00;24f'/

       CHARACTER*80    PROMPT
        CHARACTER*10    INPUT_STRING

!
!   Perform the RI_INQ_REQUEST User Request Procedure.  This
!   routine will duplicate the work done by the RI_INQ_REQUEST
!   TDMS request, which displays a form to accept employee id from
!   the terminal user.
!
!       Set return status success
!
        RI_INQ_REQUEST = 1
```

```
!
!       Screen is cleared and cursor is positioned at 6th line,
!       24th column then user is requested to input Employee ID.
!
        POSITION(2:3) = '06'
       PROMPT = ESC//CLEAR//ESC//POSITION//'EMPLOYEE ID: '
       CODE = IO$_READVBLK.OR.IO$_READPROMPT
       STATUS = SYS$QIOW (,%VAL(INPUT_CHAN), %VAL(CODE),IOSB,,,
       1                %REF(INPUT_STRING), %VAL(6),,,
       1                %REF(PROMPT), %VAL(25))
       IF (.NOT. STATUS) THEN
            RI_INQ_REQUEST = STATUS
            RETURN
        END IF
!
!       Employee ID is converted from a string to the integer
!       field in the workspace record.
!
       DECODE (IOSB.TERM_OFFSET,100,INPUT_STRING) DATA_REC.EMPLOYEE_ID
100    FORMAT (I6)

       END
```

Example 14.4, "TDMS Request" shows RI_INQ_DISP_REQUEST, a TDMS request used in the second exchange step of the task shown in *Example 14.1, "Simple Inquiry Task"*.

Example 14.4. TDMS Request

```
REPLACE REQUEST RI_INQ_DISP_REQUEST

FORM   IS RI_INQ_DISP_FORM;
RECORD IS RI_EMPLOYEE_RECORD;

CLEAR SCREEN;
DISPLAY FORM RI_INQ_DISP_FORM;

OUTPUT  EMPLOYEE_ID             TO EMPLOYEE_ID,
        EMPLOYEE_FIRST_NAME     TO EMPLOYEE_FIRST_NAME,
        EMPLOYEE_LAST_NAME      TO EMPLOYEE_LAST_NAME;

WAIT;
END DEFINITION;
```

14.4.1. Writing an ACMS$RI_LIB_INIT Initialization Procedure

As an option, you can write an initialization procedure that sets up the necessary data structures for the RI shared image. For example, if the URP use an FMS-supported device, the initialization procedure might set up FMS workspaces, open an I/O channel to the terminal, open forms libraries, and do other initialization work necessary to perform I/O to the terminal. The initialization procedure must return a status value, or the RI reports a failure and cancels the current task.

The ACMSRRSHR component calls the ACMS$RI_LIB_INIT procedure only once (when it first maps the shared image) before it calls a URP in the shared image for the first time. Always name this procedure ACMS$RI_LIB_INIT. Link the initialization procedure with the shared image.

Example 14.5, "FORTRAN Initialization Procedure" shows an initialization procedure written in FORTRAN.

Example 14.5. FORTRAN Initialization Procedure

```
!****************  ACMS$RI_LIB_INIT **********************
!
        INTEGER FUNCTION ACMS$RI_LIB_INIT
!
!   This procedure is the initialization routine for the TDMS_LIB
!   library file.  The initialization procedure is not required,
!   but if one exists it must be named ACMS$RI_LIB_INIT.
!
!   Since this library file will perform QIOs to the terminal
!   instead of executing the normal TDMS request for each exchange
!   step, the initialization procedure will open a channel to
!   the user terminal.

!
!   Declaration of variables
!
!       Declare external routines
!
        INTEGER        SYS$ASSIGN
!
!       Declare input channel number to be shareable
!       by all User Request Procedures (URP).
!
        INTEGER*2      INPUT_CHAN
       COMMON /INPUT_CHANNEL/INPUT_CHAN
!
!       Set up local variables
!
       INTEGER*4      STATUS
!
!   **IMPORTANT:
!   A status must be passed back from the initialization procedure
!   or the RI will report a failure and cancel the current task.
!
!       Set the return status equal to success,
!
        ACMS$RI_LIB_INIT = 1
!
!   Perform the initialization procedure for the QIO
!   library file.
!
!       Assign the terminal pointed to by SYS$INPUT and provide
!       the I/O channel number to perform subsequent QIO calls.
!
       STATUS = SYS$ASSIGN ('SYS$INPUT',INPUT_CHAN,,)
!
!       Check the status that was return from the
!       SYS$ASSIGN system service call.  If an error
!       occurred, then return the error to the Request
!       Interface which will report the error in
!       ACMS audit log and cancel the task.
!
        IF (.NOT. STATUS) THEN
```

```
            ACMS$RI_LIB_INIT = STATUS
            RETURN
        END IF

        END
```

The name of the URP in this example is in a function statement. The URP does not pass any parameters, but it defines a COMMON area with which to open a channel to a user terminal. This initialization URP sets the return status to success. When using the RI, it is important to return a status from the URP whether it is an initialization URP or any other URP. The RI agent checks the status returned. If there is an error, the message is returned to the RI. The RI reports the error to ACMS, cancels the current task, and logs the message in the ACMS audit trail log. If you do not return status information from a URP, the RI agent:

- Reports an error ("Error Processing request interface call") and cancels the task
- Is unable to record the exact error message in the ACMS audit trail log

Example 14.6, "Example of Audit Trail Error Messages" shows an example of the error messages that would be recorded in the audit trail log if the RI agent was unable to open the I/O channel for the RI_INQ task.

Example 14.6. Example of Audit Trail Error Messages

```
Task canceled in STEP EX1 task RI_INQ_REQUEST in Group RI_PASSED_GROUP
Error processing request interface call
Error executing initialization routine for request library file
      DEVICE:$[$DIR]LIBRARY1.EXE

PROGRAM ERROR:

Error processing request interface call
Error in Request RI_INQ_REQUEST, error returned from URP in
      DEVICE:$[$DIR]LIBRARY1.EXE
SYSTEM-F-INVCHAN, invalid I/O channel
```

14.4.2. Writing an ACMS$RI_LIB_CANCEL Cancellation Procedure

You can optionally write a cancel procedure to abort any I/O in progress and do any necessary cleanup work for a task. If a task is canceled for any reason such as by the **ACMS/CANCEL** operator command (discussed in *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/]), the Request Interface searches for the ACMS$RI_LIB_CANCEL routine in the shareable image and uses that routine as the cancellation procedure for the task. *Example 14.7, "FORTRAN Cancel Procedure"* shows a cancellation procedure written in FORTRAN.

Example 14.7. FORTRAN Cancel Procedure

```
!****************** ACMS$RI_LIB_CANCEL  ***********************
!
        INTEGER FUNCTION ACMS$RI_LIB_CANCEL
!
!   This procedure is the cancellation routine for the TDMS_LIB
!   library file.  The cancellation procedure is not required,
```

```
!   but if one exists it must be named ACMS$RI_LIB_CANCEL.
!
!   This routine will get called when the current task gets
!   canceled either by the operator command "ACMS/CANCEL TASK"
!   or the using typing Ctrl\C during an exchange step.  It
!   signals an ACMS$_CANCELD error, which will abort any
!   QIO's that were in progress and will unwind the call!   stack.

!
!   Declaration of variables
!
!       Declare external symbols and routines
!
        EXTERNAL    ACMS$_CANCELD
        INTEGER     LIB$SIGNAL,SYS$CANCEL
!
!       Declare input channel number to be shareable
!       by all User Request Procedures (URP).
!
        INTEGER*2      INPUT_CHAN
       COMMON /INPUT_CHANNEL/INPUT_CHAN
!
!       Declare local variables
!
        INTEGER*4   STATUS
!

!   Perform the cancellation procedure for the TDMS_LIB
!   library file.
!
!       Cancel any QIO that is outstanding
!
        STATUS = SYS$CANCEL(%VAL(INPUT_CHAN))
        CALL LIB$SIGNAL(ACMS$_CANCELD)
!
        END
```

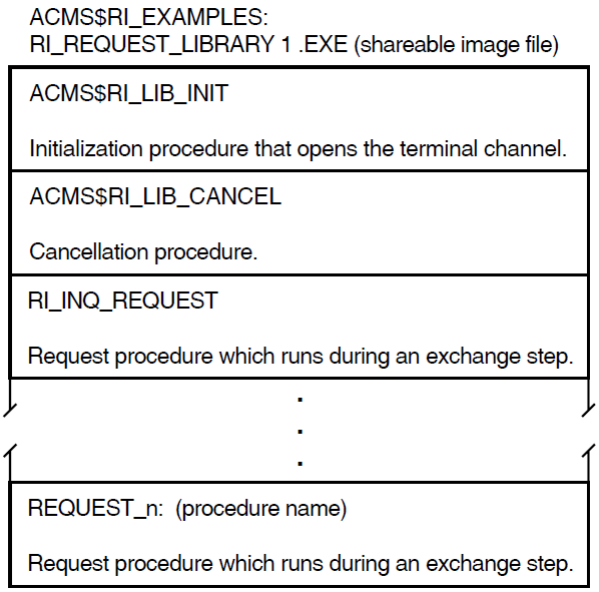
14.4.3. Compiling and Linking URPs

After writing the user request procedures for all the tasks in a task group and any initialization and cancellation procedures, compile those procedures and link them into a shareable image file. Reference the shareable image file in either of the following ways:

- In the REQUEST LIBRARY IS clause of the definition of each task group whose tasks call URPs
- By defining the ACMS$RI_LIB logical name to point to the URP shareable image file

An application can use multiple shareable image files. Each shareable image file contains any number of URPs, an optional URP initialization procedure to set up the necessary work areas, and an optional URP cancellation procedure. *Figure 14.3, "User-Written Shareable Image File"* shows the contents of a shareable image file.

Figure 14.3. User-Written Shareable Image File



The following command compiles request procedures RI_INQ_REQUEST, ACMS$RI_LIB_INIT, and ACMS$RI_CANCEL, and produces three object modules:

```
FORTRAN RI_INQ_REQUEST, ACMS$RI_LIB_INIT, ACMS$RI_LIB_CANCEL/DEBUG
```

When linking your request procedures, you must:

- Name the ACMS$RI_DEBUG_MODULE object module in the **LINK** command if you want to test a URP in the debugger.
- Use an options file to declare all URP names as universal and common data as nonshared. See *Example 14.8, "REQPROCS.OPT Options File"* for an example of an options file on OpenVMS.

Example 14.8. REQPROCS.OPT Options File

```
SYMBOL_VECTOR = (RI_INQ_REQUEST = PROCEDURE,-
                 ACMS$RI_DEBUG_ROUTINE = PROCEDURE ,-
                 ACMS$RI_LIB_INIT = PROCEDURE ,-
                 ACMS$RI_LIB_CANCEL = PROCEDURE)
PSECT_ATTR = INPUT_CHANNEL,PIC,USR,OVR,REL,GBL,NOSHR,NOEXE,RD,WRT,NOVEC
```

The RI uses the OpenVMS Run-Time Library (RTL) routine LIB$FIND_IMAGE_SYMBOL to locate the URP to call. Therefore, the request library executable image must meet the following requirements of the LIB$FIND_IMAGE_SYMBOL routine:

- The image must be a shareable image. Therefore, always link URPs as a shareable image.
- The names of all URPs, including ACMS$RI_DEBUG_ROUTINE, ACMS$RI_LIB_INIT, and ACMS$RI_LIB_CANCEL, must be universal symbols.
- All common (global) areas defined as shareable between the URPs must not be shareable by different processes, and must be declared nonshareable in an options file. Every process can share the RI request library shareable image code but must have its own copy of common data areas.

You can use any OpenVMS file names for your procedures and for the shareable image file.

Example 14.9, "Linking Shareable Images and Using an Options File" links into a shareable image the four object modules. It also shows how to use the options file in *Example 14.8, "REQPROCS.OPT Options File"* and how to meet the requirements of the LIB$FIND_IMAGE_SYMBOL RTL service.

Example 14.9. Linking Shareable Images and Using an Options File

```
$ LINK/DEBUG/SHARE/EXECUTABLE=REQPROCS.EXE RI_INQ_REQUEST, -
_$ ACMS$RI_LIBINIT, ACMS$RI_LIB_CANCEL, -
_$ SYS$LIBRARY:ACMS$RI.OLB/INCLUDE=ACMS$RI_DEBUG_MODULE, -
_$ REQPROCS.OPT
```

In this list of commands:

1. The **/SHARE** qualifier creates an OpenVMS shareable image. If you forget this qualifier and attempt to use the image, the RI logs the error "Reserved addressing fault" in the ACMS audit trail log to notify you that the image has not been linked as a shareable image.

2. The **SYMBOL_VECTOR** command in the options file sets up the global symbols as keys into the image. The symbols referenced must be global to be defined as universal. You define a symbol as universal by using the **SYMBOL_VECTOR** command. All URPs, including the initialization, debugger, and cancellation URPs, must be defined as universal symbols. If you try to access a URP that was not declared as a universal symbol, the RI logs the following error in the ACMS audit trail log:

   ```
   %LIB-F-KEYNOTFOU, key not found in tree
   ```

3. The **PSECT_ATTR** command in the option file sets the URP common area as not shareable. By default, all common areas have the PSECT attribute SHR. This attribute lets multiple processes share the same data area if it is installed (using the OpenVMS Install Utility) as writable.

   Be sure that all URPs for a given RI request library can share the data areas. However, those data areas should not be shareable across different processes. If you define the PSECT attribute as NOSHR, each user (process) can have a separate copy of the data area. If you try to run the RI with the PSECT attribute set to SHR, the RI logs an error in the ACMS audit trail log, notifying you that the image must be installed as WRITABLE. This message indicates that the PSECT attribute is set to SHR. Remember, if you install the image as writable, the data areas are shared by many processes and the URPs may access corrupted data.

14.5. Providing an RI Agent

When using the RI, do not use the ACMS-supplied agent (CP). Instead, use an RI agent to interface the task submitter to ACMS. An RI agent is provided as part of the ACMS software. You can use either this ACMS-supplied agent, called ACMS$RI_AGENT, or you can write your own agent to meet the specific needs of an application.

The ACMS$RI_AGENT provides an example of how an agent program can be developed to utilize the RI interface. An RI agent uses the Systems Interface (SI) services to sign users in to ACMS, enable the RI agent to perform I/O through the request interface, and call tasks in an ACMS application. The ACMS$RI_EXAMPLES directory contains a listing of the source code of the ACMS-supplied RI agent. See *VSI ACMS for OpenVMS Systems Interface Programming* [https://docs.vmssoftware.com/vsi-acms-sys-interface-prog/] for more information about the Systems Interface and its services.

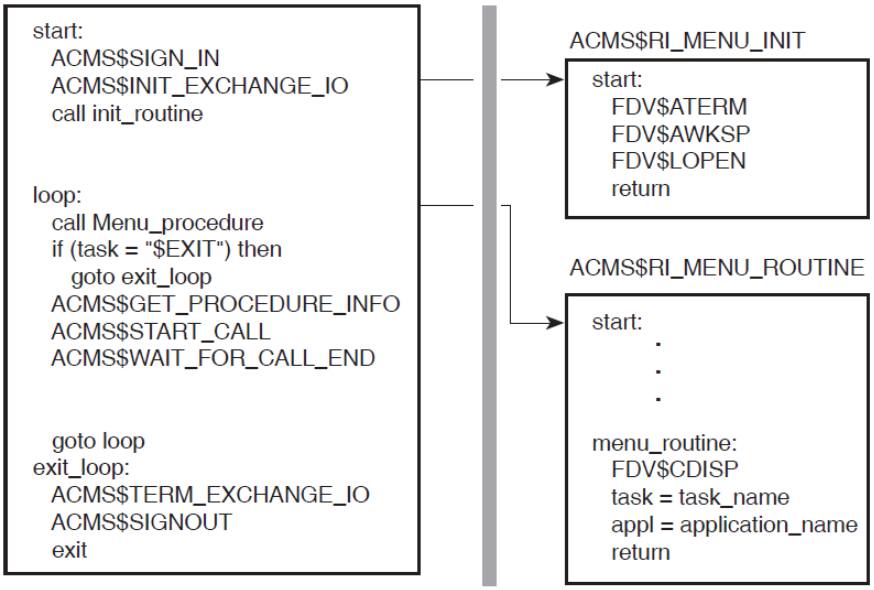
Always run an RI agent from a user process so there is one RI agent for each user process. An RI agent submits tasks to ACMS under the OpenVMS user name of the user process. The RI agent signs in each user process to identify it to the ACMS system. Just as in a standard ACMS system, only users

authorized with the ACMS User Definition Utility can gain access to ACMS. Also, all terminals or devices accessing ACMS must be authorized with the ACMS Device Definition Utility. See *VSI ACMS for OpenVMS Managing Applications* [https://docs.vmssoftware.com/vsi-acms-managing-applications/] for more information about authorizing users and devices.

The RI agent ACMS$RI_AGENT (by default) prompts the user for an application and task name. An RI agent can also call a menu interface through which the user can enter data. You can write your own menu interface modules, or if you are interfacing to FMS, you might want to use the ACMS-supplied FMS menu interface. *Section 14.5.1, "Providing a Menu Interface"* describes more about how to provide a menu interface.

Figure 14.4, "Pseudocode for an RI Agent Using an FMS Menu Interface" shows the pseudocode for the Request Interface agent and two menu interface routines. The routines in the figure use FMS calls to display the menu and retrieve the data.

Figure 14.4. Pseudocode for an RI Agent Using an FMS Menu Interface



In this figure, the RI agent:

1. Signs in the user to the ACMS system. The agent uses the ACMS$SIGN_IN service to sign in the user. By default, the user name signed in is the agent's OpenVMS user name.

2. Calls the ACMS$INIT_EXCHANGE_IO service to prepare the agent for use with the RI. The RI agent uses the ACMS$M_IO_ENABLE_SYNC_RI flag to indicate to the ACMS EXC that the RI agent performs all task I/O. See *VSI ACMS for OpenVMS Systems Interface Programming* [https://docs.vmssoftware.com/vsi-acms-sys-interface-prog/] for more information on flags used by the ACMS$INIT_EXCHANGE_IO service.

3. Determines whether or not the agent calls a user-written menu. The user-written menu routine can be linked into the agent, or into a shared image that is pointed to by the ACMS$RI_MENU logical name. The RI agent uses the ACMS$RI_MENU logical to locate the menu interface image, if any, that contains the ACMS$RI_MENU_INIT and ACMS$RI_MENU_ROUTINE user interface routines. If there is no menu, ACMS prompts the user for the application name and task name.

4. Performs the exit processing starting at step 7 if the user enters $EXIT in response to the "Enter task name:" prompt. Otherwise, the agent calls the ACMS$GET_PROCEDURE_INFO service to find out what I/O method (terminal, request, stream, or none) and procedure ID to use for the task.

5. Uses the ACMS$START_CALL service to submit an ACMS task if the task and application names are valid. Because the ACMS$START_CALL service only starts the task, the agent must also include an ACMS$WAIT_FOR_CALL_END service to wait for the task to end.

6. Loops back to step 4 to prompt for another task selection.

7. Calls the ACMS$TERM_EXCHANGE_IO service to terminate the exchange I/O enabled earlier.

8. Calls the ACMS$SIGNOUT service to sign the user out of ACMS.

While taking these steps to develop an agent, you may need to define several logical names and routines for use at run time. *Table 14.1, "Logical Names Used by the Request Interface"*, *Table 14.2, "Routines Used by the Request Interface"*, and *Table 14.3, "Menu Interfaces Used by the Request Interface"* summarize these names and routines.

Table 14.1. Logical Names Used by the Request Interface

Logical Name	Used by ACMS at Run Time
ACMS$RI_LIB_library-name	A logical defined by the user that identifies the request library or shareable image associated with the name used in a REQUEST IS USING clause.
ACMS$RI_MENU	A logical defined by the user that identifies the shareable image created by linking the ACMS$RI_MENU_INIT menu initialization procedure and the ACMS$RI_MENU_ROUTINE menu interface procedure.

Table 14.2. Routines Used by the Request Interface

Routine Name	Identifies
ACMS$RI_LIB_CANCEL	An optional user-written cancel procedure that the RI calls to do cleanup work when a task is canceled.
ACMS$RI_DEBUG_ROUTINE	An ACMS-supplied routine that the Request Interface starts up when you debug URP object modules.
ACMS$RI_LIB_INIT	An optional user-written initialization procedure that sets up the necessary data structures for the RI executable image.

Table 14.3. Menu Interfaces Used by the Request Interface

Routine Name	Identifies
ACMS$RI_MENU_INIT	An optional user-written initialization procedure that sets up to use the ACMS$RI_MENU_ROUTINE menu interface.
ACMS$RI_MENU_ROUTINE	A user-written (URP) menu interface procedure.

14.5.1. Providing a Menu Interface

There are two ways to obtain task selection information from the user: write your own menu interface, or code the RI agent to prompt you automatically for information. If you choose not to write a menu interface and you use the ACMS-supplied RI agent, by default the ACMS$RI_AGENT prompts the user to enter the task and application selections. If you choose to use a menu interface, the

ACMS$RI_AGENT can optionally call special-purpose menu interface (URP) procedures with which the user can enter the task and application information.

With a menu interface in place, the application is easier to use because you do not have to select an application and the specific task each time you want to run a task. To use a menu interface with the ACMS$RI_AGENT, a programmer must write a menu initialization procedure and a menu interface procedure. These procedures must be named ACMS$RI_MENU_INIT and ACMS$RI_MENU_ROUTINE, respectively.

Include these procedures in the RI agent code in one of the following ways:

- Link them into a shared image and then dynamically activate them at run time. Activate the shareable image in the RI agent process by using the LIB$FIND_IMAGE_SYMBOL RTL routine. In order to activate these procedures in the ACMS$RI_AGENT, define the logical ACMS$RI_MENU to point to the shared image file that contains the two menu interface procedures.

- Link them directly into the RI agent code and call them as required. This is the preferred and recommended method because it results in better performance.

If interfacing to FMS, you may want to use the ACMS-supplied FMS menu interface. ACMS supplies an FMS-based menu interface as part of the RI software. The sample FMS form library for the menu interface is located in the ACMS$RI_EXAMPLES directory.

As with the user-written menu interface described previously, you can include the ACMS-supplied FMS menu interface in an RI agent by either defining the logical ACMS$RI_MENU to point to a shared image that contains the ACMS$RI_MENU_INIT and ACMS$RI_MENU_ROUTINE procedures, or relinking the RI agent with the ACMS$RI_MENU_INIT and ACMS$RI_MENU_ROUTINE procedures.

The layout of the FMS menu form is similar to the ACMS menu format. The FMS menu form contains a menu header, a selection list, a prompt line and a message line. The menu header is the name of the menu. The selection list shows the tasks to run and the menus for display. The prompt line includes both the SELECTION: prompt and the blank spaces after the prompt. You type the number or keyword for the task or menu you want in the blank space after the prompt. Press **Return** after your selection.

Example 14.10, "FMS Initialization Procedure" and *Example 14.11, "FMS Menu Procedure"* show FMS-supplied initialization and menu interface definitions.

Example 14.10. FMS Initialization Procedure

```
FUNCTION LONG ACMS$RI_MENU_INIT

        RET_STATUS = FDV$ATERM(TCA%(),12%,12%)

        RET_STATUS = FDV$AWKSP(FMS_WORKSPACE%(),2000%)

        RET_STATUS = FDV$LOPEN('ACMS$RI_FMS_MENU_LIB'm10%)

END FUNCTION
```

Example 14.11. FMS Menu Procedure

```
FUNCTION LONG ACMS$RI_MENU_ROUTINE ( STRING TASK_NAME ,        &
                                     STRING APPLICATION_NAME,  &
                                     LONG   EXECUTION_STATUS )
```

```
        RET_STATUS = FDV$CDISP(TRM$(CURRENT_FORM))

        RET_STATUS = FDV$GET ( MENU_OPTION, TERMINATOR, 'OPTION')

        RET_STATUS = FDV$RETDI(VAL%(DATA_INDEX),NAMED_DATA)

END FUNCTION
```

To change FMS menus, change only the form definition. You do not need to make code changes to change FMS menus. The NAMED_DATA syntax is part of FMS forms definition and is explained in the FMS documentation. For more information about how to modify the ACMS-supplied FMS menu interface to work with your application, see *Appendix F, "Modifying the FMS Menu Interface"*.

14.5.2. Compiling and Linking Menu Interface URPs with the RI Agent

After writing the menu initialization and interface procedures, compile those procedures and link the resulting object modules with the RI agent in one of the following ways:

- Link the ACMS$RI_MENU_INIT and ACMS$RI_MENU_ROUTINE menu interface object modules with the RI agent object module.
- Link the ACMS$RI_MENU_INIT and ACMS$RI_MENU_ROUTINE menu interface object modules (using the **/SHARE** qualifier) into a menu interface image. At run time, the RI agent uses the ACMS$RI_MENU logical name to locate that image.

For example, type the following command to link the RI agent to include the FMS menu interface:

```
$ LINK/EXE=fmsagent.exe  sys$input/option
        sys$library:acms$ri.olb/include=acms$ri_agent_module
        sys$library:acms$ri_fms_menu.olb/include=
        (acms$ri_menu_init, acms$ri_menu_routine)

$ DEFINE ACMS$RI_LIB_libraryname ri_request_library.exe

$ DEFINE ACMS$RI_FMS_MENU_LIB fmsformmenulib.flb

$ RUN fmsagent
```

At run time, the ACMS$RI_AGENT uses the following steps to run the menu interface procedures:

1. If the menu interface procedures have been linked with the agent object module, it runs those procedures.
2. If the menu interface procedures have not been linked with the agent object module, the ACMS$RI_AGENT uses the logical name ACMS$RI_MENU to locate the menu interface image containing the menu initialization and interface procedures.
3. If the menu interface procedure is not available, then the ACMS$RI_AGENT prompts the user for an application and task name.

To debug menu initialization and menu interface procedures, follow the instructions for debugging URPs in *Section 14.6, "Debugging Applications that Call URPs"*.

14.5.3. User-Written Menus for the ACMS$RI_AGENT

When you write a user-written menu routine for the ACMS$RI_AGENT, it is important to be aware of how your chosen language handles dynamic string descriptors. The task name and application name arguments in the ACMS$RI_AGENT are passed to the ACMS$RI_MENU_ROUTINE as dynamic descriptors. If the language you have chosen does not support dynamic string descriptors, you must use an OpenVMS run-time library routine to return the task and application names to the ACMS$RI_AGENT.

The following FORTRAN example accepts an application name and a task name into two fixed-length strings. The STR$TRIM OpenVMS RTL routine is then used to remove trailing spaces and copy the names into the arguments supplied by the ACMS$RI_AGENT.

```
!
!   THIS ROUTINE GETS THE TASK AND APPLICATION
!   NAMES FROM THE USER....
!
INTEGER FUNCTION ACMS$RI_MENU_ROUTINE(TASK,APPL,TASK_STS)
!
! Addresses of appl and task name dynamic string descriptors
INTEGER*4 TASK, APPL
!
! Completion status of previous task (0 if 1st time through)
INTEGER*4 TASK_STS
!
! Local strings to input application and task names
CHARACTER*32 TNAME, ANAME
!
! RTL completion status
INTEGER*4 STATUS
!
! RTL routine to trim spaces from a string
INTEGER*4 STR$TRIM
!
WRITE( UNIT=5,FMT='(A,$)' ) ' INPUT APPLICATION SELECTION: '
READ ( UNIT=5,FMT='(A32)' ) ANAME
WRITE( UNIT=5,FMT='(A,$)' ) ' INPUT TASK SELECTION: '
READ ( UNIT=5,FMT='(A32)' ) TNAME
!
STATUS = STR$TRIM( %REF(TASK), %DESCR(TNAME) )
IF (STATUS) THEN
STATUS = STR$TRIM( %REF(APPL), %DESCR(ANAME) )
END IF
!
ACMS$RI_MENU_ROUTINE = STATUS
RETURN
END
```

14.6. Debugging Applications that Call URPs

The method used to debug the RI user request procedures depends on whether or not:

- You have a running ACMS application (with the application started and menus defined) and you want to debug just the RI agent and the URPs
- You need to debug the complete application including the RI agent, ACMS tasks, and URPs

Section 14.6.1, "Using the OpenVMS Debugger to Debug URPs Using a Running Application" discusses using the OpenVMS Debugger to debug applications that are already up and running, but are currently using TDMS requests. *Section 14.6.2, "Using the ACMS Task Debugger to Debug URPs and Their Tasks"* describes using the ACMS Task Debugger to debug URPs and the tasks that they call.

Do the following to prepare to debug:

1. Link the RI agent with the debugger:

   ```
   $ LINK/DEBUG/EXE=debug_agent -
    _$ sys$library:acms$ri.olb/include=acms$ri_agent_module
   ```

   This command creates an image named DEBUG_AGENT.EXE in your default directory. You can use the optional **/EXE** qualifier to assign a name to the image. If you do not use the qualifier, the linker uses the file name of the first object module in the **LINK** command as the name of the server image. The default file type is .EXE. Make sure that the name of the server image is the same as the name in the IMAGE clause of the task group definition.

   Include the **/DEBUG** qualifier in the **LINK** command to set up the table of symbols used for debugging.

2. Include the ACMS$RI_DEBUG_MODULE object module in the RI shareable image file by adding the following line to the **LINK** command in *Example 14.9, "Linking Shareable Images and Using an Options File"*:

   ```
   sys$library:acms$ri.olb/include=acms$ri_debug_module
   ```

   ACMS supplies the ACMS$RI_DEBUG_MODULE to debug URPs. Link the ACMS$RI_DEBUG_MODULE in the shared image that contains the URPs so that when the shared image is activated, the RI agent (ACMRRSHR component) will invoke the debugger. The image is activated by the LIB$FIND_IMAGE_SYMBOL RTL routine in the RI agent. The ACMS$DEBUG_ROUTINE is then invoked to signal the debugger. The user sets the module to the URP module to set breaks and so on.

To protect business data, set up test files to run against the task. If your procedures use logical names to identify the files used, create a set of data files in another directory and temporarily redefine the logical names to point to that directory. If defining these logical names from DCL command mode, be sure to define the names as group or system logicals using the **/GROUP** or **/SYSTEM** logical on the **DEFINE** or **ASSIGN** command.

14.6.1. Using the OpenVMS Debugger to Debug URPs Using a Running Application

Start with a running ACMS application that is currently working by using TDMS requests. Then define the ACMS$RI_LIB_library-name logical name to point to the URP shareable image file. Then run the RI agent you linked with the debugger. For example:

```
$ ACMS/START APPLICATION test_application
$ DEFINE ACMS$RI_LIB_request_library1  request_lib_image.exe
$ RUN debug_agent
DBG> GO
Task Name:
Application Name:
DBG>
```

In this example, test_application is a fully developed ACMS application that contains a TDMS request library named request_library1.rlb. The shareable image created to replace that TDMS request library is named request_lib_image.exe.

14.6.2. Using the ACMS Task Debugger to Debug URPs and Their Tasks

This section provides an outline of the steps to debug URPs and their tasks by using the ACMS Task Debugger. This method of debugging is very similar to the method for debugging tasks submitted by user-written agents that is discussed in *VSI ACMS for OpenVMS Writing Server Procedures* [https://docs.vmssoftware.com/vsi-acms-writing-server-proc/]. By following these instructions and supplementing this discussion with the one in *VSI ACMS for OpenVMS Writing Server Procedures* [https://docs.vmssoftware.com/vsi-acms-writing-server-proc/], you can debug URPs with the RI debugger and debug tasks with the ACMS Task Debugger. To use this debugging method, do the following:

1. Type the following commands to start the ACMS Task Debugger process:

   ```
   $ SET PROCESS/PRIVILEGE=SHARE
   $ ACMS/DEBUG/AGENT_HANDLE=dummyapplicationname testgroup.tdb
   ACMSDBG>
   ```

   In response to the ACMSDBG prompt, start servers needed by the task and set breakpoints before selecting the task (see *VSI ACMS for OpenVMS Writing Server Procedures* [https://docs.vmssoftware.com/vsi-acms-writing-server-proc/]). Then type the **ACCEPT** command to allow the ACMS Task Debugger to accept calls from the RI agent program:

   ```
   ACMSDBG> ACCEPT
   ```

2. Make tasks selected by the RI agent program run in the ACMS Task Debugger by defining the following before you run the RI agent:

   ```
   $ DEFINE ACMS$RI_LIB_request_library1  request_lib_image.exe
   $ DEFINE ACMS$DEBUG_AGENT_TASK "TRUE"
   $ RUN debug_agent
   DBG> GO
   Task Name:
   Application Name: dummyapplicationname
   DBG>
   ```

   When you run the RI agent linked with the OpenVMS debugger, the debugger prompt appears. In these commands, the `dummyapplicationname` is the agent handle that was used in the **ACMS/DEBUG/AGENT_HANDLE** command in the debugger process (see *VSI ACMS for OpenVMS Writing Server Procedures* [https://docs.vmssoftware.com/vsi-acms-writing-server-proc/]).

After setting up the debugger process, start the agent process and enter the **GO** command at the DBG> prompt. Because a menu interface was not linked into the RI agent in this example, the ACMS$RI_AGENT prompts you for a task name and an application name.

After you enter the task and application names, the RI determines if the RI request library shareable image file contains the ACMS$RI_DEBUG_MODULE procedure (URP). If the debugger procedure exists, the OpenVMS debugger (DBG>) prompt appears.

At this point, use the **SET IMAGE** command to tell the debugger that you are going to use the RI request library shareable image file. The name of the image specified with this command must be the

same as the logical name used to point to the RI request library image. Then you can set breakpoints at any URPs, including the initialization and cancellation URPs. For example:

```
DBG> SET IMAGE ACMS$RI_LIB_requestlibraryname
 DBG> SET BREAK urp1
 DBG> SET BREAK urp2
 DBG> GO
```

Type $EXIT in response to the "Enter Task Name:" prompt to exit from the RI agent.

14.7. Defining an Application that Uses the Request Interface

There are no differences between an application definition that uses DECforms or TDMS and one that uses the Request Interface. *Example 14.12, "Example Application Definition"* shows the application definition that uses the task group and tasks defined in this document.

Example 14.12. Example Application Definition

```
REPLACE APPLICATION RI_SAMPLE_APPL
  AUDIT;
APPLICATION USERNAME IS adf$exc;
DEFAULT APPLICATION FILE IS
              "ACMS$RI_EXAMPLES:RI_SAMPLE_APPL.ADB";

SERVER DEFAULTS ARE
  AUDIT;
  USERNAME IS adf$server;
END SERVER DEFAULTS;

TASK DEFAULTS ARE
   AUDIT;
END TASK DEFAULTS;

TASK GROUP IS
   RI_LOGICAL_GROUP:
       TASK GROUP FILE IS "ACMS$RI_EXAMPLES:RI_PASSED_GROUP.TDB";
END TASK GROUP;
END DEFINITION;
```

14.8. Running the Agent

To run your agent in a production environment, relink the request procedures, omitting the **/DEBUG** qualifier and the RI debug object module ACMS$RI_DEBUG_MODULE. Then use the DCL **RUN** command followed by the name of the RI agent image. For example:

```
$ RUN ACMS$RI_AGENT
```

Part II. Writing and Migrating Applications to OpenVMS

This part describes how to write ACMS applications for a VSI OpenVMS (OpenVMS) system.

Chapter 15. Introduction

ACMS supports features on OpenVMS systems, including DECforms support. This part of the manual describes the following:

- Writing applications for OpenVMS
- I/O options and restrictions
- Managing applications on OpenVMS

Chapter 16. Writing Applications for OpenVMS

This chapter describes how to write applications using ACMS, focusing on information that is specific to ACMS on OpenVMS.

16.1. Writing an ACMS Application for OpenVMS

An ACMS application typically consists of the following source components:

- Task definitions
- Task group definitions
- Menu definition
- Application definition
- Server procedures
- User-written agents
- Programs that call ACMS Queued Task Services
- Message definition files
- CDD record definitions
- Forms

Follow these steps to build an ACMS application for OpenVMS:

1. Invoke the CDD Dictionary Operator utility on OpenVMS to define the record definitions.
2. Invoke ADU on the OpenVMS system to build the task, task group, menu, and application definitions.
3. Invoke the DECforms IFDL translator on the OpenVMS system to build the form files.
4. Invoke the OpenVMS 3GL compiler of choice and the OpenVMS linker to compile and link the server procedures, user-written agents, and programs that call the ACMS Queued Task Services.
5. Invoke the OpenVMS Message utility to build the message definitions.

16.2. Form Changes and Form File Caching

All form files (DECforms form image files and TDMS request libraries) used by an application must be on the same node on which the application is started because the application execution controller (EXC) checks that all files are present when the application starts. This is necessary in order for ACMS to cache the form files or request libraries to the submitter node when a user selects a task that uses DECforms or TDMS in exchange steps.

The form image files or request libraries must be in the location specified in the task group definition.

The following line from a task group definition file shows that the form file is in the directory pointed to by the AVERTZ_DEFAULT logical name:

```
.
.
.
FORM IS vr_form IN "avertz_default:vr_form.form";
.
.
.
```

In this example, the VR_FORM.FORM form file must be in the AVERTZ_DEFAULT directory.

The following line from a task group definition file shows that the request library is in the directory pointed to by the AVERTZ_DEFAULT logical name:

```
.
.
.
REQUEST LIBRARY IS "avertz_default:deprmsrlb.rlb";
.
.
.
```

In this example, the DEPRMSRLB.RLB request library must be in the AVERTZ_DEFAULT directory on OpenVMS.

16.2.1. Formatting and Naming VSI DECforms Form Image Files

The DECforms form image files (.EXE) on the application node must have the image format expected on the submitter node:

- If the submitter node is an OpenVMS node, the form image file on the application node must be an OpenVMS image.

16.2.2. Caching with Multiple Submitter Platforms

To support forms caching, the application execution controller (EXC) checks that all form image files are present when the application starts. Therefore, if you use DECforms form image files on multiple submitter node platforms, you must define the following logical name on the application node:

```
ACMS$MULTIPLE_SUBMITTER_PLATFORMS
```

This notifies the EXC that there are multiple submitter node platforms. The scope of the logical name can be systemwide. However, if you have multiple applications, you may want to define this logical name at the application level by using the APPLICATION LOGICALS clause in your application definition.

At startup, the EXC translates the logical name. If the value of the logical name is set to "T", "t", "Y", "y", or "1" and the file type in the FORMS clause of the task group definition specified is .EXE, the EXC looks for the presence of both form image files (form_name.EXE_VAX and

form_name.EXE_AXP). If either file is missing, the application does not start. If the logical name is not defined or translates to something other than "T", "t", "Y", "y", or "1", the EXC looks for the file that is specified in the task group definition. If you have multiple submitter platforms and FORM image files are copied to the submitted nodes manually, the file type must be retained (for example, form_name.EXE_VAX and form_name.EXE_AXP).

Note

1. The platform-specific form image files with the extensions .EXE_VAX and .EXE_AXP must be present in the specified directory on the application node. In addition, the logical name ACMS$MULTIPLE_SUBMITTER_PLATFORMS must be defined for all applications that have submitters on mixed platforms, even if the form image files are copied to the submitter nodes manually rather than being cached.

2. When using multiple submitter platforms, a form name cannot be a logical name. If the task group definition uses a logical name for the form name, the EXC cannot locate the form because no logical name translation is performed.

16.3. Using Logical Names

Use logical names instead of hardcoded names to make your application more flexible and easier to maintain. If an application is moved to another directory or node, you simply redefine the logical names to reflect the new configuration. If you do not use logical names, you must modify and rebuild your application. You can use logical names to define the following:

- File names
- Node names
- Device names
- CDD path names

Using logical names also makes distributing your application easier, because fewer changes in the source code are required.

Note

When used in a multiplatform environment, form names cannot be logical names. See *Section 16.2.2, "Caching with Multiple Submitter Platforms"* for more information.

Table 16.1, "ADU Clauses in Which You Can Specify a Logical Name" lists the ADU clauses in which you can specify a logical name, and the ACMS definitions containing the clauses.

Table 16.1. ADU Clauses in Which You Can Specify a Logical Name

Clause	Definition
IMAGE	Task, task group
WORKSPACES	Task, task group
DEFAULT OBJECT FILE	Task group

Clause	Definition
DEFAULT TASK GROUP FILE	Task group
FORMS	Task group
MESSAGE FILES	Task group
PROCEDURE SERVER IMAGE	Task group
REQUEST LIBRARIES	Task group
APPLICATION DEFAULT DIRECTORY	Application
DEFAULT APPLICATION FILE	Application
DEFAULT DIRECTORY	Application
TASK GROUPS	Application
DEFAULT APPLICATION	Menu
DEFAULT MENU FILE	Menu
MENU	Menu
TASK	Menu

Chapter 17. I/O Options and Restrictions

This chapter describes the following:

- I/O options and restrictions when distributing an application
- I/O options and restrictions when one of the nodes in a distributed environment is an OpenVMS node
- Task and menu selection on OpenVMS

17.1. Restrictions for Distributing an Application

It is important to know the I/O methods that the tasks in your application use when you are determining whether or not you can distribute the application. There are defaults for the I/O attributes, and, in some cases, it is not obvious by looking at the task definition to see what the I/O attributes of the task are.

You can use the ADU **DUMP GROUP** command to display the contents of the task group database. The output of this command shows the I/O type of each task in the task group. The ADU **DUMP GROUP** command lists the I/O type of each block, exchange, and processing step for each task in the task group. For example:

```
ADU> DUMP GROUP VR_TASK_GROUP.TDB/OUTPUT=VR_TASK_GROUP_DUMP.DMP
```

You cannot distribute tasks that have the following:

- TERMINAL I/O in a processing step
- REQUEST I/O in a processing step
- Clauses that run in a DCL server and require terminal I/O

  Tasks that run in a DCL server and require terminal I/O can be selected only from the system where the application is started. The following task definition clauses run in a DCL server:

  - DATATRIEVE COMMAND
  - DCL COMMAND
  - IMAGE

  The OpenVMS image specified in the IMAGE clause must be compatible with the platform on which the application is executing.

17.2. Selecting Tasks and Menus on OpenVMS

You can use the **ACMS/ENTER** command on OpenVMS to select the following kinds of tasks:

- Tasks that require no terminal interaction
- Tasks that specify FORM I/O in the task definition

- Tasks that specify STREAM I/O in the task definition
- Some local tasks that specify TERMINAL I/O in the task definition

Because TDMS is not on the OpenVMS platform, a menu that uses this request option appears as a command line menu when you issue the **ACMS/ENTER** command on OpenVMS. The following sections describe the commands and keys that are available in the command line menu interface.

17.2.1. ACMS Menu Commands

If you issue the **ACMS/ENTER** command on an OpenVMS system, ACMS returns the Selection prompt as the following example shows:

```
$ ACMS/ENTER
Selection:
```

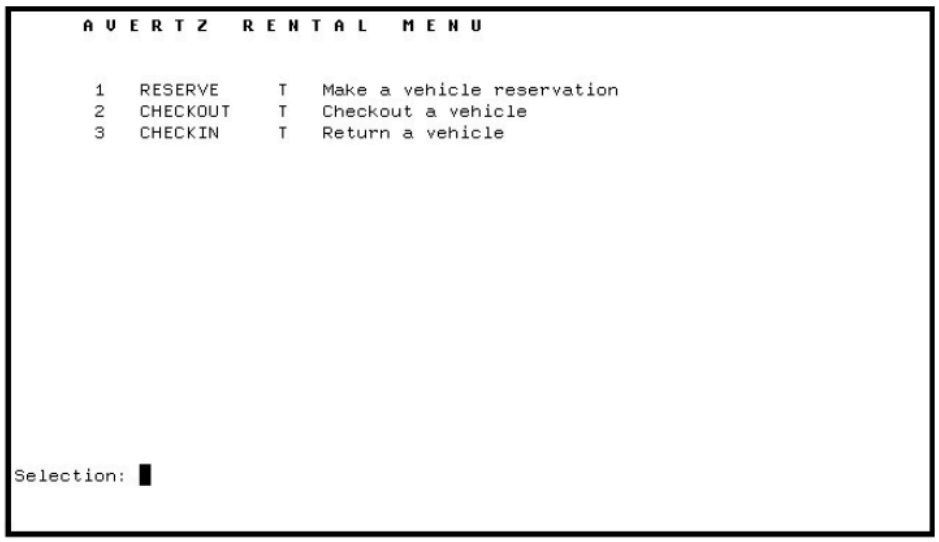
Enter a dollar sign ($) at the Selection prompt to enable the ACMS command menu prompt. *Table 17.1, "ACMS Menu Commands"* lists the ACMS commands that are available at the selection prompt.

Table 17.1. ACMS Menu Commands

Command	Description
CONTINUE	Returns to a selection menu from a command menu
EXIT	Ends your ACMS session and signs you out of ACMS
HELP	Provides information on ACMS commands and menus
MENU	Displays selection or command menu
NOMENU	Displays a Command: or Selection: prompt without displaying the command or selection menu
SELECT	Selects a task that is not in your menu tree without including the menu path for the task

A selection menu is application-specific. Issue the **MENU** command at the Selection prompt to display menu items. *Figure 17.1, "AVERTZ Rental Menu"* shows the selection menu from the AVERTZ sample application.

Figure 17.1. AVERTZ Rental Menu



You select an item from the menu by either name or number. For example, to select the reservation task, type RESERVE or 1.

Menus can display tasks and menus. The letter (T or M) following the task name in the selection menu indicates whether the menu item is a task or another menu (called a **submenu**). If you know the name of the task you want to select from a submenu, you can enter both the menu keyword and the task keyword at the Selection prompt.

You can also select a task without specifying the menu path by including the application name and the task name on the command line. For example, suppose that you want to access the DELETE task, which is an item on a submenu of the PAYROLL application. You can select it directly by issuing the following command:

```
Selection: SELECT SAMPLE::PAYROLL DELETE
```

If you know the menu path of the task you want to select, you can access the task directly from the Selection prompt. For example:

```
Selection: 1 2 1
```

This allows you to select the task without going through every menu in the menu path.

17.2.2. ACMS Function Keys

Table 17.2, "ACMS Function Keys" lists the keys that are available in the command line menu interface.

Table 17.2. ACMS Function Keys

Key	Description
Asterisk	Enables the default menu
Ctrl/Y	Cancels the current ACMS task
DELETE	Erases the last character typed
Dollar sign	Enables the command menu
Hyphen	Enables the menu one level above the current menu

Part III. Appendixes

The appendixes provide the following supplemental information:

- *Appendix A, "Changing the ACMS Menu Format Using VSI DECforms"* shows how to modify the standard ACMS menu format using by DECforms.
- *Appendix B, "Changing the ACMS Menu Format Using TDMS"* shows how to modify the standard ACMS menu format by using TDMS.
- *Appendix C, "Using CDO for Pieces Tracking "* shows how to use CDO to track relationships between ACMS entities in the CDD dictionary.
- *Appendix D, "Using LSE with ACMS"* demonstrates how to use the optional Language-Sensitive Editor productivity tool to enter ACMS code on line.
- *Appendix E, "Request Interface Kit Components"* lists the files included with the ACMS Request Interface examples and software supplied by ACMS.
- *Appendix F, "Modifying the FMS Menu Interface"* explains how to modify the FMS menu interface that you use with the ACMS Request Interface.
- *Appendix G, "Checklist of References to Platform-Specific Files"* contains a list of references to platform-specific files in an ACMS application.

Appendix A. Changing the ACMS Menu Format Using VSI DECforms

Do not attempt to make changes in the standard ACMS menu unless you are thoroughly familiar with DECforms. Errors in the DECforms records and definitions that ACMS uses for menu work can produce fatal exceptions that cause other parts of the ACMS system to fail; avoid changing the ACMS menu format except when there is a serious need.

This appendix also describes how to disable the **SELECT** command, which gives the user the ability to select a task by application name and task name from the ACMS Command Menu. Disabling the **SELECT** command does not require using DECforms.

A.1. Modifying Menu Appearance Without Changing the Default Format

There are two ways to revise the ACMS menu format without changing the default format. First, you can include OpenVMS images that use option lists or menus in your application. Second, when you set up a menu using ADU, you can make a number of choices in the menu format that ACMS supplies. For example, you can define the text that appears at the top of the menu. You define the entries that are displayed on the screen and the descriptive text for these entries.

By using the HEADER, DEFAULT APPLICATION, and ENTRIES clauses (and the subclauses under ENTRIES), you can change what is displayed on the menu. The format of the menu, however, is the same from menu to menu:

- The two lines of header text are the top two lines of the screen.
- Each page of the menu can contain up to 16 entries.
- Each entry line consists of a number, a keyword, a task/menu flag, and descriptive text.
- The Selection: prompt displays on the third line from the bottom.
- The bottom line is used for all messages except "Press <RET> for more" and "Press <RET> for first page". These messages are displayed on the line above the selection prompt.

A.2. Modifying the ACMS Menu Using VSI DECforms

You can change some parts of this format just by changing the DECforms panel definitions that ACMS uses. For example, you can change the Selection: prompt or the "Press <RET> …" message lines by changing the Default and Command panel definitions in the ACMS_MENU.IFDL file. Other changes can require you to change other parts of the ACMS_MENU.IFDL source file that is the source of the menu form. The rest of this appendix explains how to change the source file, ACMS_MENU.IFDL, in order to modify the ACMS menu format.

ACMS uses two panel definitions contained in the ACMS_MENU.IFDL file to control the display of menus on the system:

- Panel DEFAULT_PANEL is the complete ACMS menu, which includes 16 entries, a 2-line menu header, the Selection: prompt, and a 2-line selection input field.
- Panel EXPERT_DEF_PANEL contains the Selection: prompt and the 2-line selection input field; this form is for users who want to select tasks without seeing menus.

There are two corresponding definitions in the ACMS_MENU.IFDL for the ACMS Command Menu: panel COMMAND_PANEL, and panel EXPERT_COM_PANEL. The latter displays only the Command: prompt while the former displays eight entries plus the Command: prompt.

To modify the default ACMS menu format, you need to change the ACMS_MENU.IFDL file that displays and controls the menu form. For example, if you want to change the number of entries that can appear on a menu, you must change the ACMS_MENU.IFDL file. After it has been modified, you use DECforms to create a new .EXE file to produce the menu format. For more information on creating a new .EXE file after you have modified the ACMS_MENU.IFDL file, see *VSI DECforms Guide to Developing an Application* [https://docs.vmssoftware.com/vsi-decforms-for-openvms-guide-to-developing-an-application/].

To change the menu, you must either modify the ACMS_MENU.IFDL file supplied by ACMS or create a new ACMS_MENU.IFDL file.

You can modify the ACMS menu ACMS_MENU.IFDL file and keep the same name for it. Alternatively, you can make a copy of the ACMS_MENU.IFDL file and make your changes in the copy. In either case, you use DECforms procedures to create a new .EXE file.

ACMS always looks for the menu form file in the same place with a fixed name: SYS$SHARE:ACMS_MENU.EXE. So after you have modified the ACMS_MENU.IFDL file or a copy of it and created a new .EXE file, it must always be copied to the SYS$SHARE directory with the ACMS_MENU.EXE file name. All menus on your ACMS system then use the modified default format.

A.2.1. Obtaining the ACMS VSI DECforms Default Menu File

To create a new default menu file, you can make a copy of the source file, ACMS_MENU.IFDL, and alter it as explained in this appendix in order to meet your needs. Or, you can alter the ACMS_MENU.IFDL file itself if you are sure the changes you are making are permanent. The ACMS_MENU.IFDL is also stored in the SYS$SHARE directory.

Caution

Do not, under any conditions, change the HEADER RECORD DATA or the MENU CONTROL RECORD DATA definitions. There is only one entry in the MENU ENTRY RECORD DATA definition that can be changed, as explained in the following sections. Changes in any other fields of these definitions can cause the ACMS system to fail.

A.2.2. How ACMS Uses Menu Form Record Definitions

Five form record definitions in the ACMS menu definition file control how the information you supply in the menu definitions you write as part of an application is used.

The first is the menu header record. *Example A.1, "Definition for ACMS Menu Header"* shows how this form record definition appears in the ACMS_MENU.IFDL file.

Example A.1. Definition for ACMS Menu Header

```
Form Record MENU_HEADER
    NUM_ENTRIES Unsigned Longword
    MENU_PATH Character(70)
    MENU_HEADER_1 Character(80)
    MENU_HEADER_2 Character(80)
End Record
```

ACMS uses the NUM_ENTRIES field to pass to the form the number of entries in the MENU_ENTRIES record.

ACMS uses the two menu header fields to pass the menu header, or title, to the form file. It takes this text from the HEADER clause of the menu definition.

The default ACMS menu does not use the MENU_PATH field of this record. The menu path is the sequence of menus, identified by keyword, that the user followed in reaching the current menu; ACMS maintains this information. But this field can be used to display the user's current location in the menu tree.

Example A.2, "Definition for Menu Entries Record" shows the ACMS_MENU.IFDL file definition for the second record used by the ACMS form file, MENU_ENTRIES.

Example A.2. Definition for Menu Entries Record

```
Form Record MENU_ENTRIES
    Group ENTRIES
        Occurs 16
        ENTRY_ENABLE Character(1)
        ENTRY_NUMBER Character(6)
        ENTRY_FILL Character(3)
        ENTRY_KEY Character(10)
        ENTRY_FLAG_TEXT Character(56)
    End Group

End Record
```

ACMS uses the form record MENU_ENTRIES to pass the number, keyword, identifying flag, and descriptive text to the form file. It derives all this information for each entry from the menu database. The number for an entry is derived from the sequence in which the entry occurs in the ENTRIES clause of the menu definition you write as part of an application. The keyword is derived from the name used for the entry in the ENTRIES clause; the text is taken from the TEXT subclause for the entry. The ENTRY_ENABLE field contains one of two values: T (for tasks) or M (for menus).

Example A.3, "Definition for Menu Control" shows MENU_CONTROL, the third form record used by the form file.

Example A.3. Definition for Menu Control

```
Form Record MENU_CONTROL
    CTL_EXPERT_MENU Character(1)
    CTL_NEW_ENTRIES Character(1)
    CTL_MULTI_PAGE Character(1)
    CTL_LAST_PAGE Character(1)
End Record
```

The information in the menu control record does not come directly from the menu database. Rather, it is information that ACMS maintains for each user and for the user's current menu. The value of

CTL_EXPERT_MENU determines whether the user sees the complete ACMS menu, in which case the field is set to F (False), or sees only the selection prompt, in which case the field is set to T (True). The initial value is set from the user definition file (ACMSUDF.DAT). Each time the user types in the terminal user **MENU** or **NOMENU** command, that field is updated.

ACMS uses the MULTI_PAGE and LAST_PAGE fields in the record to tell the form file whether more menu entries are available than would fit on a single screen. It uses the NEW_ENTRIES field to tell the form file whether the entries to be displayed differ from the entries last displayed.

Example A.4, "Definition for Menu Selection Record" shows MENU_SELECTION_RECORD, the fourth record used by the form file.

Example A.4. Definition for Menu Selection Record

```
Form Record MENU_SELECTION
    Group APPL_SELECT_LINE_GROUP
        APPL_SELECTION_STRING_1 Character(68)
        Transfer APPL_SELECT_LINE_GROUP.APPL_SELECTION_STRING_1
                Source SELECT_LINE_GROUP.SELECTION_STRING_1
        APPL_SELECTION_STRING_2 Character(187)
        Transfer APPL_SELECT_LINE_GROUP.APPL_SELECTION_STRING_2
                Source SELECT_LINE_GROUP.SELECTION_STRING_2
    End Group

End Record
```

This record consists of two fields which accept the selection strings from the menu. The APPL_SELECTION_STRING_1 and APPL_SELECTION_STRING_2 fields are the actual fields received by the ACMS Command Process (CP). The user input is made in the SELECTION_STRING fields. The two APPL_SELECTION_STRING fields accept up to a maximum of 255 characters. If the actual input is less, the strings are padded with blanks. The size of the APPL_SELECTION_STRING fields cannot be changed.

Example A.5, "Control Text Response Found Record" shows the fifth record, which tells the ACMS CP if any control text response with an accept phase has been executed.

Example A.5. Control Text Response Found Record

```
Form Record MENU_CTRL_TEXT_FOUND
    CTRL_TEXT_RESP_FOUND Longword Integer
End Record
```

A.2.3. Instructions Performed by the Form File

The .EXE file built from the ACMS_MENU.IFDL file carries out a number of functions when ACMS calls a menu you defined in the course of constructing an application. This section explains those functions.

First, either a complete menu or the selection prompt only is displayed.

After the user has typed in a selection keyword or number (and, optionally, a selection string) and pressed **Return**, the request moves that information to the menu selection record.

The form file uses the value passed in the CTL_EXPERT_MENU field to determine whether to display the complete menu or only the expert menu. If the CTL_EXPERT_MENU field is set to T, the expert menu is displayed, the selection is entered in the menu selection record, and the process ends.

If the CTL_EXPERT_MENU field is not set to T, the form file then checks whether the entries to be displayed are the same ones as the last entries the user saw. If the value of the CTL_NEW_ENTRIES field of the control record is set to a value other than T, then the menu is the same as the last one. In this case, the form file displays the last menu, accepts the selection string, and ends.

If the menu to be displayed does contain new entries, then ACMS has set the value of CTL_NEW_ENTRIES to T. In this case, the form file displays the menu form and outputs the menu header. It then begins displaying the entries to the indexed fields on the menu form. As long as there is an entry, which the form file checks by looking at one of the fields to be displayed, it continues to produce the entry information. As soon as there are no more entries, or when the sixteenth entry has been displayed, the process stops.

If there are more entries than the number of entries allowed per screen, ACMS sets the value of CTL_MULTI_PAGE to "T". This indicates that the menu has multiple pages.

If there are more than 16 entries, which is the default number of entries for each screen, then the request tests the CTL_LAST_PAGE field to determine whether or not this is the last page of the menu displayed to the user. ACMS indicates the last page of the menu by setting the CTL_LAST_PAGE field to T. If this is a multiple page menu and it is the last page, the form file displays the message "Press <RET> for first page". If it is not the last page of the menu, the form file outputs the message "Press <RET> for more". It then accepts the selection from the user and ends.

A.2.4. Modifying the Menu Appearance Only

If you are changing only the appearance of the menu without changing the number of entries, you can make the changes by modifying the panel definitions in the ACMS_MENU.IFDL file. By modifying the panel definitions, you can change:

- The background text that displays on the menu. The only background text on the ACMS-supplied form is the Selection: or Command: prompt. You can change this prompt or add other background text.
- The text of the instruction for multiple-page menus.

For a detailed explanation of how to modify DECforms form definitions, see *VSI DECforms Guide to Developing an Application* [https://docs.vmssoftware.com/vsi-decforms-for-openvms-guide-to-developing-an-application/]. If you are changing only the Selection: or Command: prompts or adding more background text, or want to change the text of the instructions for multiple page menus, you will be altering the panel definitions only.

The ACMS_MENU.IFDL file supplied with ACMS includes a number of DECforms panel definitions. To make the changes detailed in the following paragraphs, you must alter two of these panel definitions, those for DEFAULT_PANEL and COMMAND_PANEL.

These two definitions are very similar, but you must be sure to make the changes you want in both definitions so that your menus will be consistent. *Example A.6, "Panel Definition"* shows how the DEFAULT_PANEL definition appears in the ACMS_MENU.IFDL file.

Example A.6. Panel Definition

```
Panel DEFAULT_PANEL
             Viewport DEFAULT_VIEW
             Field MENU_HEADER_1
                 Line 1
                 Column 1
                 Display
```

```
            Bold
        Output Picture X(80)
    End Field

    Field MENU_HEADER_2
        Line 2
        Column 1
        Display
            Bold
        Output Picture X(80)
    End Field

    Literal Text
        Next Line
        Value ""
    End Literal

    Group ENTRIES
        Vertical
            Displays 16
        Entry Response
            Reset
                INFO_LINE_1
        End Response

        Exit Response
            Message
                ""
            If (RECALL_HOLD = 1) Then
                Let RECALL_HOLD = 0
            Else
                If (PREVIOUS_PAGE = 1) Then
                    Let SELECT_LINE_GROUP.SELECTION_STRING_1 = "\"
                    Reset
                        SELECT_LINE_GROUP.SELECTION_STRING_2
                Else
                    If (NEXT_PAGE = 1) Then
                        Reset
                            SELECT_LINE_GROUP.SELECTION_STRING_1
                        Reset
                            SELECT_LINE_GROUP.SELECTION_STRING_2

                    Else
                        Let SELECT_LINE_GROUP.SELECTION_STRING_1
=
                            ENTRIES(MENU_ENTRIES_INDEX).ENTRY_KEY
                        Reset
                            SELECT_LINE_GROUP.SELECTION_STRING_2
                        Include RESET_RECALL_LIST
                    End If
                End If
            End If
        End Response

        Function Response CHANGE
            Let ON_ENTRIES = 0
            Let RECALL_HOLD = 1
            Deactivate
```

```
                        All
                    Activate
                        Field SELECT_LINE_GROUP.SELECTION_STRING_2 on
                            DEFAULT_PANEL
                    Activate
                        Field SELECT_LINE_GROUP.SELECTION_STRING_1 on
                            DEFAULT_PANEL
                    Position To Field SELECT_LINE_GROUP.SELECTION_STRING_1
On
                        DEFAULT_PANEL
                End Response

                Function Response GO_UP
                    If (MENU_ENTRIES_INDEX = 1) Then
                        Let PREVIOUS_PAGE = 1
                        Return
                    Else
                        Position To Up Occurrence
                    End If
                End Response

                Function Response GO_DOWN
                    If (MENU_ENTRIES_INDEX = NUM_ENTRIES) Then
                        Let NEXT_PAGE = 1
                        Return
                    Else
                        Position To Down Occurrence
                    End If
                End Response

                Field ENTRY_NUMBER
                    Next Line
                    Column 3
                    Active Highlight
                        Reverse
                    Output Picture X(6)
                    No Data Input
                End Field

                Field ENTRY_KEY
                    Same Line
                    Column 12
                    Output Picture X(10)
                End Field

                Field ENTRY_FLAG_TEXT
                    Same Line
                    Column 22
                    Output Picture X(56)
                End Field

            End Group

            Field INFO_LINE_1
                Line 21
                Column 1
                Output Picture X(80)
            End Field
```

```
            Group SELECT_LINE_GROUP
                Entry Response
                    If (CTL_MULTI_PAGE = "T") Then
                        If (CTL_LAST_PAGE = "T") Then
                            Let INFO_LINE_1 = "                    .......
"-
                                "Press <RET> for first page ......."
                        Else
                            Let INFO_LINE_1 = "
 ....... "-
                                "Press <RET> for more ......."
                        End If
                    Else
                        Reset
                            INFO_LINE_1
                    End If
                End Response

                Exit Response
                    Message
                        ""
                    If (RECALL_HOLD = 1) Then
                        Let RECALL_HOLD = 0
                    Else
                        If (PREVIOUS_PAGE = 1) Then
                            Let SELECT_LINE_GROUP.SELECTION_STRING_1 = "\"
                            Reset
                                SELECT_LINE_GROUP.SELECTION_STRING_2
                            Let PREVIOUS_PAGE = 0
                        Else
                            If (NEXT_PAGE = 1) Then
                                Reset
                                    SELECT_LINE_GROUP.SELECTION_STRING_1
                                Reset
                                    SELECT_LINE_GROUP.SELECTION_STRING_2
                                Let NEXT_PAGE = 0
                            Else
                                Include RESET_RECALL_LIST
                            End If
                        End If
                    End If
                End Response

                Function Response CHANGE
                    Let ON_ENTRIES = 1
                    Let RECALL_HOLD = 1
                    Reset
                        SELECT_LINE_GROUP.SELECTION_STRING_1
                    Reset
                        SELECT_LINE_GROUP.SELECTION_STRING_2
                    Deactivate
                        All
                    Activate
                        Field ENTRIES.ENTRY_NUMBER on DEFAULT_PANEL
                    Position To Field ENTRIES(1).ENTRY_NUMBER
                        On DEFAULT_PANEL
                End Response
```

```
                Function Response GO_UP
                    Include GET_PREVIOUS_RECALL_ITEM
                End Response

                Function Response GO_DOWN
                    Include GET_NEXT_RECALL_ITEM
                End Response

                Literal Text
                    Line 22
                    Column 1
                    Value "Selection: "
                End Literal

                Field SELECTION_STRING_1
                    Line 22
                    Column 12
                    Autoskip
                    Output Picture X(68)
                End Field

                Field SELECTION_STRING_2
                    Line 23
                    Column 1
                    Output Picture X(79)
                End Field

            End Group

        End Panel
```

If you make changes to add background text, you are adding literals, in DECforms terminology. Be sure that your new background text areas do not conflict with areas on the panel that are already defined for use in the ACMS_MENU.IFDL file.

To change the prompt, you must alter the wording in two panel definitions. These definitions begin with:

```
Panel DEFAULT_PANEL
```

and

```
Panel COMMAND_PANEL
```

Example A.7, "Default Panel Field Definition" shows the section of the default panel definition in the ACMS_MENU.IFDL file which needs to be altered to change the prompt.

Example A.7. Default Panel Field Definition

```
Literal Text
     Line 22
     Column 1
     Value "Selection: "
```

Example A.8, "Command Panel Field Definition" shows the section of the ACMS_MENU.IFDL file which needs to be altered to change the Command: prompt in the Command Panel definition.

Example A.8. Command Panel Field Definition

```
Literal Text
     Line 22
     Column 1
     Value "Command: "
```

You can replace the quoted string in either definition with text which suits your needs.

To change the text of the instructions which appear on multiple-page menus, you also make changes in the DEFAULT_PANEL and COMMAND_PANEL panel definitions in the ACMS_MENU.IFDL file.

To change these instructions, you must find the record group SELECT_LINE_GROUP in the ACMS_MENU.IFDL file. *Example A.9, "Record Group SELECT_LINE_GROUP"* shows how the group entry appears in the ACMS_MENU.IFDL file supplied with ACMS.

Example A.9. Record Group SELECT_LINE_GROUP

```
Group SELECT_LINE_GROUP
                  Entry Response
                      If (CTL_MULTI_PAGE = "T") Then
                          If (CTL_LAST_PAGE = "T") Then
                              Let INFO_LINE_1 = "                    ....... 
 "-
                                  "Press <RET> for first page ......."
                          Else
                              Let INFO_LINE_1 = "
  ....... "-
                                  "Press <RET> for more ......."
                          End If
```

To change the instructions, change the text within the quotation marks. Remember to change the text in both the DEFAULT_PANEL panel definition and the COMMAND_PANEL definition or the instructions the user sees will vary depending on what panel is in use.

If you want to change any other characteristics of the ACMS menu format, you must modify the other areas in the ACMS_MENU.IFDL file beyond the panel definitions.

A.2.5. Changing SELECTION_STRING Field Lengths

The SELECTION_STRING is an ACMS-supplied system workspace. It allows a user to enter input after making a menu selection. ACMS places this input in the SELECTION_STRING workspace and ACMS has access to it within the task called by the menu selection. It could be used, for example, to pass an employee number to an employee record update task.

In the ACMS_MENU.IFDL file, the selection string fields are defined as a 68-character field beginning after the Selection: or Command: prompt and a 79-character field beginning in column 1 of the line below the Selection: or Command: prompt.

If you need SELECTION_STRING fields of different lengths than these, you must alter the ACMS_MENU.IFDL file in four places.

First, you must change two record groups, SELECT_LINE_GROUP and RECORD_LIST. *Example A.10, "SELECTION_STRING Record Groups"* shows the two groups as they appear in the ACMS_MENU.IFDL file.

Example A.10. SELECTION_STRING Record Groups

```
Group SELECT_LINE_GROUP
    SELECTION_STRING_1 Character(68)
    SELECTION_STRING_2 Character(79)
End Group

Group RECALL_LIST
    Occurs 20
    RECALL_ITEM_1 Character(68)
    RECALL_ITEM_2 Character(79)
End Group
```

The length values for the fields can be changed. If a change is made in one group, however, the corresponding change must be made in the other group.

Next, you must change the SELECTION_STRING field sizes in the panel definitions to correspond to the new values you entered in the record group definitions. Remember to change the field sizes in both the DEFAULT_PANEL and COMMAND_PANEL definitions. *Example A.11, "SELECTION_STRING Panel Definitions"* shows the field definitions as they appear in the panel definitions in the ACMS_MENU.IFDL file supplied by ACMS.

Example A.11. SELECTION_STRING Panel Definitions

```
Field SELECTION_STRING_1
    Line 22
    Column 12
    Autoskip
    Output Picture X(68)
End Field

Field SELECTION_STRING_2
    Line 23
    Column 1
    Output Picture X(79)
End Field
```

Remember that the values in the panel field definitions must correspond to the changes you made earlier in the record group definitions.

A.2.6. Changing the Number of Entries per Screen

The menu file supplied with ACMS causes a maximum of 16 entries per screen to be displayed when the DEFAULT_PANEL panel definition is used. The ACMS_MENU.IFDL file also contains panel definitions for a COMMAND_PANEL and for two expert panels.

As supplied by ACMS, the menu file allows up to 16 lines per menu screen in the default panel and eight lines per screen in the command panel. The expert panels display only the prompts Selection: or Command: for those who want to make selections without seeing menu entries.

When there are fewer than 16 menu entries, ACMS fills out the screen with blanks. When there are more than 16 menu entries, ACMS will break up the menu into multiple pages. Each screen will display one page of the menu. If you need to change the number of entries allowed per screen, you need to make changes in two places: in a record definition and in the DEFAULT_PANEL definition.

The first change you need to make is in the record group ENTRIES. *Example A.12, "ENTRIES Record Group"* shows how that record group appears in the ACMS_MENU.IFDL file as supplied with ACMS.

Example A.12. ENTRIES Record Group

```
Group ENTRIES
    Occurs 16
    Current MENU_ENTRIES_INDEX
    ENTRY_ENABLE Character(1)
    ENTRY_NUMBER Character(6)
    ENTRY_FILL Character(3)
    ENTRY_KEY Character(10)
    ENTRY_FLAG_TEXT Character(56)
End Group
```

You can change the "Occurs 16" line to the number of lines you wish to be the maximum per menu screen.

Caution

Do not, under any conditions, change any other field in the group ENTRIES. This is a record used by the ACMS CP and changing values other than the 16 in "Occurs 16" may cause the ACMS system to fail.

Next you must change the Displays statement of the record group ENTRIES in the DEFAULT_PANEL panel definition. *Example A.13, "ENTRIES Record in Default Panel Definition"* shows how that section of the panel definition appears in the ACMS_MENU.IFDL file.

Example A.13. ENTRIES Record in Default Panel Definition

```
Group ENTRIES
     Vertical
         Displays 16
```

You can change the number of entries allowed per screen for the Command Panel in a similar fashion.

When you change the number of entries allowed per screen, you must also change the size of the record for the entries passed from the ACMS CP to the menu form. To let the ACMS CP know about the change in the size of the record for the entries, you have to specify the new number of entries allowed per screen in the ACMS menu definition. *Example A.14, "Menu Definition Specifying Entries per Screen"* shows an example of this.

Example A.14. Menu Definition Specifying Entries per Screen

```
CREATE MENU PERSONNEL_MENU
  HEADER IS "          P E R S O N N E L     M E N U";
  CONTROL TEXT IS "DFMNU" WITH 12 ENTRIES PER SCREEN;
  DEFAULT APPLICATION IS "PERSONNEL";
  ENTRIES ARE
    ADD  : TASK IS ADD_EMPLOYEE;

           TEXT IS "Add a new employee record";
       .
       .
       .
   END ENTRIES;
END DEFINITION;
```

DFMNU is the default control text response. If you have defined other control text responses in the form, you can specify those responses. On each system, all the menus use the same form, so all menus must use the same number of entries allowed per screen.

Caution

If you change the "Occurs 16" entry in the ENTRIES Record Group, then *all* menus on your system must use this number of entries and *all* menu definitions must include the "WITH n ENTRIES PER SCREEN" phrase. So if you change the number of entries per screen *all* menu definitions on the system that use DECforms must be altered.

You can include more entries in the menu definition than the number you specify in the WITH ENTRIES PER SCREEN phrase. For example, you can include 13 or more entries in the menu definition shown in *Example A.14, "Menu Definition Specifying Entries per Screen"* even though the WITH ENTRIES PER SCREEN phrase specifies 12 entries for each screen. In this case, when ACMS displays the menu it also displays the message:

```
"....... Press <RET> for more .......".
```

If the user presses **Return**, ACMS displays the additional entries listed in the menu definition. But remember that whatever number of entries per screen you choose, that is the number that must be used by all menus on your system.

A.2.7. Changing the Size of the Command Line Recall Buffer

Another item that can be changed, if need be, is the number of lines in the command line recall buffer. This buffer stores the commands you issue at the terminal keyboard. Commands can be recalling by pressing the up-arrow key, much as commands can be recalled at the DCL level. As supplied by ACMS, the number of commands that can be recalled is 20.

To change the number of commands that can be recalled, you need to change the default value of 20 in two definitions in the ACMS_MENU.IFDL file. *Example A.15, "Command Line Recall Buffer Definitions"* shows how those definitions appear in the form file supplied with ACMS.

Example A.15. Command Line Recall Buffer Definitions

```
Group RECALL_LIST
    Occurs 20
    RECALL_ITEM_1 Character(68)
    RECALL_ITEM_2 Character(79)
End Group

RECALL_MAX Longword Integer
    Value 20
```

Remember to change the definitions so that the two values match.

A.2.8. Changing the VSI DECforms Layout Size

Another area of the ACMS menu format which you can change is the layout size.

The layout size specifies the largest rectangular area on the terminal screen that the menu form will occupy at any given time. In the ACMS_MENU.IFDL file supplied with ACMS, this size is the DECforms default value, 24 lines deep by 80 columns wide.

If you need to change this, you must alter the DEFAULT_LAYOUT definition in the ACMS_MENU.IFDL file. *Example A.16, "Definition of Default Screen Layout"* shows how that definition appears.

Example A.16. Definition of Default Screen Layout

```
Layout DEFAULT_LAYOUT
    Device
        Terminal VT_100_DEV
            Type %VT100
        Terminal VT_200_DEV
            Type %VT200
    End Device
    Size 24 Lines by 80 Columns
```

A.2.9. Using a Customized Response and Panel Definition

You can write new control text responses and panel definitions in the ACMS_MENU.IFDL file in order to create exactly the responses or panels that meet your needs. If so, you must include a CONTROL TEXT clause in your menu definition to point to the new panel definition.

When you use the CONTROL TEXT clause in a menu definition, you specify the control text response or responses that you want to execute when the ACMS CP gets input from users. By default, ACMS executes the DFMNU control text response for the default menu and the COMND control text response for the command menu.

Example A.17, "Menu Definition Using CONTROL TEXT" shows an example of a menu definition that uses the CONTROL TEXT clause.

Example A.17. Menu Definition Using CONTROL TEXT

```
CREATE MENU PERSONNEL_MENU
  HEADER IS "          P E R S O N N E L     M E N U";
  CONTROL TEXT IS "MENU1MENU2MENU3";

  DEFAULT APPLICATION IS "PERSONNEL";
  ENTRIES ARE
    ADD  : TASK IS ADD_EMPLOYEE;

           TEXT IS "Add a new employee record";

       .
       .
       .
  END ENTRIES;
END DEFINITION;
```

In the CONTROL TEXT clause, you must include the five-character string that identifies the CONTROL TEXT RESPONSE. Here, the string "MENU1MENU2MENU3" sets up a sequence of three responses.DECforms automatically reads each five characters as the CONTROL TEXT definition. You can specify up to five control text responses. At least one of these responses must have an accept phase, or input from the user. Otherwise, you will get a "control text response not found" error.

A.2.10. Building and Installing the New Menu Form

After you have customized the ACMS_MENU.IFDL file, you must build the new menu form as an executable image and make it available to the ACMS system.

You build a new menu form by creating a new ACMS_MENU.EXE file. This involves translating the form, extracting an object from the .FORM file to create a .OBJ file, linking the object file to produce a shareable image, and copying the new executable file to the common SYS$SHARE directory on the system.

If you have customized the ACMS menu form definition file, you must rebuild the form image file. Use the following DCL command to rebuild and install the new form image file on the system:

```
$ @SYS$MANAGER:ACMS_BUILD_MENU_FORM.COM
```

In order to make the new version of the menu form file available to the command process, you must stop and restart the ACMS terminal subsystem. Do this by using the **ACMS/STOP TERMINALS** and **ACMS/START TERMINALS** commands. Make sure your ACMS user definition points to the menu database containing the menu that uses the new menu format. Then sign in to ACMS and check whether or not:

- You can select the menu.
- The menu has the right number of entries.
- You can select the entries.
- The fields are in the correct places.
- The cursor moves from field to field in the correct order.

If ACMS cannot display the new menu, you get this error message when signing in to ACMS:

```
A VSI DECforms request failed in the menu form SYS$SHARE:ACMS_MENU.EXE
```

If you get this message, check the record, form, control text response, and panel definitions for unmatched names, inappropriate file protection, or differences in the number of entries.

A.3. Disabling the SELECT Command in the ACMS Command Menu

The **SELECT** command appears on the ACMS Command Menu and permits a user to select tasks by application and task name. To disable the **SELECT** command, remove it from the ACMS Command Menu by editing and replacing the command menu definitions, and rebuilding the command database as outlined in the following steps:

1. Rename the file SYS$LIBRARY:ACMS$MENU_CUST.COM to ACMSMENU.COM. This file contains the ACMS command menu and the ACMS default menu.

2. Edit the file ACMSMENU.COM and delete or comment out the **SELECT** command entry. The exclamation points (!) in the following example mark the lines you need to comment out or delete.

   ```
   HEADER IS "                          ACMS Command Menu";
   ENTRIES ARE
   ```

```
  "Continue":
    TEXT IS "Continue interrupted operation";
    COMMAND 3;
  "Exit":

    TEXT IS "Leave the ACMS menu environment";
    COMMAND 8;
  "Help":
    TEXT IS "Display ACMS help information on the terminal";
    COMMAND 17;

  "Menu":
    TEXT IS "Display selection and command menus";
    COMMAND 18;
  "Nomenu":

    TEXT IS "Do not display selection and command menus";
    COMMAND 19;
  !"Select":
  !  TEXT IS "Select task by application name and task name";
  !  COMMAND 20;
END ENTRIES;
END DEFINITION;
```

3. Change the CDD path names specified in ACMSMENU.COM to be the CDD path names of your choice.

4. Use the following ADU commands to insert the definition file ACMSMENU.COM in the CDD:

```
$ ADU
ADU> @ACMSMENU.COM
```

5. Use the following ADU commands to rebuild the ACMS command menu database:

```
$ ADU
ADU> BUILD MENU COMMAND$ROOT ACMS$DIRECTORY:ACMSCMD.MDB
ADU> EXIT
```

Appendix B. Changing the ACMS Menu Format Using TDMS

Changing the standard ACMS menu requires the full TDMS kit; do not attempt to make changes in the menu unless you are thoroughly familiar with TDMS. Errors in the TDMS requests and records that ACMS uses for menu work can produce fatal exceptions that cause other parts of the ACMS system to fail; avoid changing the ACMS menu format except when there is a serious need. This appendix also describes how to disable the **SELECT** command, which gives the user the ability to select a task by application name and task name from the ACMS Command Menu. Disabling the **SELECT** command does not require using TDMS.

B.1. Modifying the Menu Format Using ACMS

There are two ways to revise the ACMS menu format without changing the default format. First, you can include OpenVMS images that use option lists or menus in your application. For an example of a task that includes an option list, see the Employee task of the ACMS Sample Application. Second, when you set up a menu using ADU, you can make a number of choices in the menu format that ACMS supplies. For example, you can define the text that appears at the top of the menu. You define the entries that are displayed on the screen and the descriptive text for these entries.

By using the HEADER, DEFAULT APPLICATION, and ENTRIES clauses (and the subclauses under ENTRIES), you can change what is displayed on the menu. The format of the menu, however, is the same from menu to menu:

- The two lines of header text are the top two lines of the screen.
- Each page of the menu can contain up to 16 entries.
- Each entry line consists of a number, a keyword, a task/menu flag, and descriptive text.
- The Selection: prompt displays on the third line from the bottom.
- The bottom line is used for all messages except "Press <RET> for more" and "Press <RET> for first page". These messages are displayed on the line above the selection prompt.

You can change some parts of this format just by changing the form definition that ACMS uses. For example, you can change the Selection: prompt or the location of the "Press <RET> ..." message lines by changing the menu form. Other changes can require you to change the TDMS request that displays the menu form. The rest of this appendix explains how to use TDMS requests to modify the ACMS menu format.

B.2. Modifying the ACMS Menu Using TDMS Requests

To modify the default ACMS menu format, you need to change the TDMS request that displays the menu form. For example, if you want to change the number of entries that can appear on a menu, you must change the menu request.

To change the menu request, you must either modify the menu request supplied by ACMS or create a new menu request in order to add or delete fields:

- If you modify the ACMS menu request and keep the same name for it, you must also rebuild the ACMS menu request library. All menus on the ACMS system then use the new menu format. You must rebuild the ACMS menu request library if you either change the ACMS-supplied menu or add your own.
- If you are creating a new menu request, you must name the request in a REQUEST clause in all menu definitions that use the new request. Make sure the number of entries in the request is the same as the number in the ENTRIES PER SCREEN phrase in the menu definition.

  Next, add the new request to the MENU_LIBR request library definition. Then rebuild MENU_LIBR to create a new version of the ACMSREQ.RLB request library file that includes the new request.

The only way to change the number of entry lines on an ACMS menu is to change the menu request. Whenever a menu definition includes fewer entries than there are entry lines in the menu request, only the entries for that menu are displayed. The remaining entry lines are filled with spaces by TDMS.

B.2.1. Getting the ACMS Menu Request and Form

To create a new menu request, either you can create a copy of the ACMS menu request or you can create entirely new request and form definitions. To modify the ACMS menu request, load the contents of the SYS$SYSTEM:ACMSREQ.BAK file into a CDD directory:

```
$ DMU
DMU> SET DEF CDD$TOP.MENU_REQUESTS
DMU> RESTORE SYS$SYSTEM:ACMSREQ.BAK
DMU> EXIT
```

In this example, the contents of the ACMSREQ.BAK file are stored in the CDD directory MENU_REQUESTS, directly below CDD$TOP. Be sure that the directory you use is in the CDD before you use the **RESTORE** command.

If the ACMSREQ.BAK file is not in the directory pointed to by the SYS$SYSTEM logical name, you can get it from the ACMS distribution kit by reinstalling ACMS.

Table B.1, "Definitions Copied to the CDD" lists the definitions that the **RESTORE** command loads into the CDD.

Table B.1. Definitions Copied to the CDD

Definition Name	Description
BLANK_FORM <CDD$FORM>	Form used to reset the screen. *Do not change this definition.*
CLEAR_SCREEN <CDD$REQUEST>	Request used to reset the screen. *Do not change this definition.*
COMD_CONTROL_RECORD <CDD$RECORD>	Record used for control information for command menu.
COMD_ENTRY_RECORD <CDD$RECORD>	Record used for entry information for command menu.
COMD_FORM <CDD$FORM>	Form used for command menu.
COMD_HEADER_RECORD <CDD$RECORD>	Record used for header of command menu.
COMD_REQUEST <CDD$REQUEST>	Request used for command menu.

Definition Name	Description
COMD_SELECTION_RECORD <CDD$RECORD>	Record used for selection string typed after Command: prompt.
EXIT_REQUEST <CDD$REQUEST>	Request used to do a $EXIT from menu.
EXPERT_COMD_FORM <CDD$FORM>	Form used for command prompt when no command menu displayed.
EXPERT_MENU_FORM <CDD$FORM>	Form used for selection prompt when no selection menu displayed.
HCOMD_FORM <CDD$FORM>	Help form for command menu.
HMENU_FORM <CDD$FORM>	Help form for selection menu.
MENU_CONTROL_RECORD <CDD$RECORD>	Record used for control information for selection menu.
MENU_ENTRY_RECORD <CDD$RECORD>	Record used for entry information for selection menu.
MENU_FORM <CDD$FORM>	Form used for selection menu.
MENU_HEADER_RECORD <CDD$RECORD>	Record used for header of selection menu.
MENU_LIBR <CDD$REQUEST_LIBRARY>	ACMSREQ.RLB definition.
MENU_REQUEST <CDD$REQUEST>	Request used for selection menu.
MENU_SELECTION_RECORD <CDD$RECORD>	Record used for selection string typed after Selection: prompt.
RESET_SCREEN <CDD$REQUEST>	Request used to reset the screen. *Do not change this definition.*

You can then use the TDMS utilities to change the MENU_FORM or MENU_REQUEST definitions to suit your needs.

WARNING

Do not, under any conditions, change the BLANK_FORM, CLEAR_SCREEN, or RESET_SCREEN definitions.

B.2.2. Modifying the Menu Form Only

If you are changing only the appearance of the menu without changing the number of entries, the only definition you need to modify is the MENU_FORM definition. By modifying only the form definition, you can change:

- The background text that displays on the menu. The only background text on the ACMS-supplied form is the Selection: prompt. You can change this prompt or add background text to the form if you wish.
- The location of the selection input field.
- The location of the header text output fields.
- The location of the entry output fields. However, they must still be indexed fields, and there must be 16 fields in the form's indexed array.

For an explanation of how to modify TDMS form definitions, see *VAX TDMS Forms Manual*.

Use the name MENU_FORM if you want all menus in the ACMS system to use the new form. In this case, you do not have to change the menu request. However, you must store the new MENU_FORM in the same CDD directory as the menu request and then rebuild the request library, as explained in *Section B.2.8, "Modifying and Building the ACMS Menu Request Library"*.

If you are changing only the menu form, be sure to use the same field names as MENU_FORM uses. Do not delete any input or output fields from the form. Keep their type and size the same as in the MENU_FORM definition.

If you want to change any other characteristics of the ACMS menu format, you must modify the existing ACMS menu request or create a new ACMS menu request.

B.2.3. Forms, Records, and Keypad Used by the Menu Request

The menu request has two parts. The first part identifies the forms, records, and keypad used by the request. The second part contains the instructions that TDMS performs when ACMS calls the request. *Example B.1, "MENU_REQUEST Definition"* shows the definition for MENU_REQUEST.

Example B.1. MENU_REQUEST Definition

```
CREATE REQUEST MENU_REQUEST

RECORD IS MENU_HEADER_RECORD;
RECORD IS MENU_ENTRY_RECORD;
RECORD IS MENU_CONTROL_RECORD;
RECORD IS MENU_SELECTION_RECORD;

FORM IS MENU_FORM;
FORM IS EXPERT_MENU_FORM;


KEYPAD IS NUMERIC;

CONTROL FIELD IS CTL_EXPERT_MENU
  "T":
    USE FORM EXPERT_MENU_FORM;
    DEFAULT FIELD SELECTION_STRING_1;
    DEFAULT FIELD SELECTION_STRING_2;
    INPUT SELECTION_STRING_1 TO SELECTION_STRING_1;
    INPUT SELECTION_STRING_2 TO SELECTION_STRING_2;
  NOMATCH:
    CONTROL FIELD IS CTL_NEW_ENTRIES
      "T":
        DISPLAY FORM MENU_FORM;


        OUTPUT MENU_HEADER_1 TO MENU_HEADER_1;
        OUTPUT MENU_HEADER_2 TO MENU_HEADER_2;


        CONTROL FIELD IS ENTRY_FLAG [1 TO 16]
          " ":
```

```
            WAIT;
           NOMATCH:
            OUTPUT ENTRY_NUMBER [%LINE] TO ENTRY_NUMBER [%LINE];
            OUTPUT ENTRY_KEY [%LINE] TO ENTRY_KEY [%LINE];
            OUTPUT ENTRY_FLAG [%LINE] TO ENTRY_FLAG [%LINE];
            OUTPUT ENTRY_TEXT [%LINE] TO ENTRY_TEXT [%LINE];
        END CONTROL FIELD;


        CONTROL FIELD IS CTL_MULTI_PAGE
          "T":
            CONTROL FIELD IS CTL_LAST_PAGE
              "T":
OUTPUT "              .......  Press <RET> for first page  ......."
                                  to MENU_MORE;
               NOMATCH:
OUTPUT "                  .......  Press <RET> for more  ......."
                                  to MENU_MORE;
            END CONTROL FIELD;
        END CONTROL FIELD;


        INPUT SELECTION_STRING_1 TO SELECTION_STRING_1;
        INPUT SELECTION_STRING_2 TO SELECTION_STRING_2;
      NOMATCH:
        USE FORM MENU_FORM;
        DEFAULT FIELD SELECTION_STRING_1;
        DEFAULT FIELD SELECTION_STRING_2;
        INPUT SELECTION_STRING_1 TO SELECTION_STRING_1;
        INPUT SELECTION_STRING_2 TO SELECTION_STRING_2;
  END CONTROL FIELD;
END CONTROL FIELD;

END DEFINITION;
```

The ACMS menu request uses two forms:

- MENU_FORM is the complete ACMS menu, which includes 16 entries, a 2-line menu header, the Selection: prompt, and a 2-line selection input field.

- EXPERT_MENU_FORM contains the Selection: prompt and the 2-line selection input field; this form is for users who want to select tasks without seeing menus.

There are four records in the ACMS menu request. *Example B.2, "Definition for ACMS Menu Header Record"* shows the definitions for the MENU_HEADER_RECORD.

Example B.2. Definition for ACMS Menu Header Record

```
DEFINE RECORD MENU_HEADER_RECORD
    DESCRIPTION IS /* RECORD FOR THE DEFAULT ACMS MENU HEADER INFO */.
MENU_HEADER_RECORD STRUCTURE.
    NUMBER_OF_ENTRIES    DATATYPE UNSIGNED LONGWORD.
    MENU_PATH            DATATYPE TEXT    70.
    MENU_HEADER_1        DATATYPE TEXT    80.
    MENU_HEADER_2        DATATYPE TEXT    80.
END MENU_HEADER_RECORD STRUCTURE.
END MENU_HEADER_RECORD.
```

ACMS uses the NUMBER_OF_ENTRIES field to pass to the request the number of entries in the MENU_ENTRY_RECORD record. It takes this value from the user's menu database, using the value assigned in the ENTRIES PER SCREEN clause in the definition for the menu.

ACMS uses the two menu header fields to pass the menu header, or title, to the request. It takes this text from the HEADER clause of the menu definition.

The default ACMS menu does not use the MENU_PATH field of this record. The menu path is the sequence of menus, identified by keyword, that the user followed in reaching the current menu; ACMS maintains this information. A request can use this field to display the user's current location in the menu tree.

Example B.3, "Definition for Menu Entry Record" shows the CDD definition for the second record used by the request, MENU_ENTRY_RECORD.

Example B.3. Definition for Menu Entry Record

```
DEFINE RECORD MENU_ENTRY_RECORD
    DESCRIPTION IS /* RECORD FOR THE DEFAULT ACMS MENU ENTRY INFO */.
MENU_ENTRY_RECORD STRUCTURE.
    LINE STRUCTURE OCCURS 16 TIMES.
        ENTRY_NUMBER      DATATYPE TEXT    2.
        ENTRY_KEY         DATATYPE TEXT    10.
        ENTRY_FLAG        DATATYPE TEXT    1.
        ENTRY_TEXT        DATATYPE TEXT    50.
    END LINE STRUCTURE.
END MENU_ENTRY_RECORD STRUCTURE.
END MENU_ENTRY_RECORD.
```

ACMS uses MENU_ENTRY_RECORD to pass the number, keyword, identifying flag, and descriptive text to the request. It derives all this information for each entry from the menu database. The number for an entry is derived from the sequence in which the entry occurs in the ENTRIES clause of the definition. The keyword is derived from the name used for the entry in the ENTRIES clause; the text is taken from the TEXT subclause for the entry. The flag field contains one of two values: T (for tasks) or M (for menus).

Example B.4, "Definition for Menu Control" shows MENU_CONTROL_RECORD, the third record used by the request.

Example B.4. Definition for Menu Control

```
DEFINE RECORD MENU_CONTROL_RECORD
    DESCRIPTION IS /* RECORD FOR THE DEFAULT ACMS MENU CONTROL INFO */.
MENU_CONTROL_RECORD STRUCTURE.
    CTL_EXPERT_MENU      DATATYPE TEXT 1.
    CTL_NEW_ENTRIES      DATATYPE TEXT 1.
    CTL_MULTI_PAGE       DATATYPE TEXT 1.
    CTL_LAST_PAGE        DATATYPE TEXT 1.
END MENU_CONTROL_RECORD STRUCTURE.
END MENU_CONTROL_RECORD.
```

The information in the menu control record does not come directly from the menu database. Rather, it is information that ACMS maintains for each user and for the user's current menu. The value of CTL_EXPERT_MENU determines whether the user sees the complete ACMS menu, in which case the field is set to F (False), or sees only the selection prompt, in which case the field is set to T (True).

The initial value is set from the user definition file (ACMSUDF.DAT). Each time the user types in the terminal user **MENU** or **NOMENU** command, that field is updated.

ACMS uses the MULTI_PAGE and LAST_PAGE fields in the record to tell the request whether more menu entries are available than would fit on a single screen. It uses the NEW_ENTRIES field to tell the request whether the entries to be displayed differ from the entries last displayed.

Example B.5, "Definition for Menu Selection Record" shows MENU_SELECTION_RECORD, the final record used by the request.

Example B.5. Definition for Menu Selection Record

```
DEFINE RECORD MENU_SELECTION_RECORD
    DESCRIPTION IS /* RECORD FOR THE DEFAULT ACMS MENU SELECTION */.
MENU_SELECTION_RECORD STRUCTURE.
    VARIANTS.
        VARIANT.
        SELECTION_STRING      DATATYPE TEXT 255.
        END VARIANT.

        VARIANT.
        SELECTION_STRING_1    DATATYPE TEXT 69.
        ! Must match size of selection_string_1 field in MENU_FORM
        SELECTION_STRING_2    DATATYPE TEXT 186.
        ! Should calculate as 255 - <size of selection_string_1>
        END VARIANT.
    END VARIANTS.
END MENU_SELECTION_RECORD STRUCTURE.
END MENU_SELECTION_RECORD.
```

This record consists of a single field, into which the request moves the selection string. The variant for this field lets the request move the two lines of the selection input field separately.

The second part of the menu request contains instructions that TDMS performs when ACMS calls the request. The next section explains these instructions.

B.2.4. Instructions Performed by TDMS When ACMS Calls a Request

A menu request must include the forms, records, and keypad it uses, as well as the instructions that TDMS performs when ACMS calls the request. This section explains the instructions that the request contains.

The second part of the menu request begins with the "CONTROL FIELD IS CTL_EXPERT_MENU" instruction. Although this part of the request is complex because of its control fields, the work it does is always the same:

1. The request displays either a complete menu or the selection prompt only.

2. After the user has typed in a selection keyword or number (and, optionally, a selection string) and pressed **Return**, the request moves that information to the menu selection record.

The request uses the value passed in the CTL_EXPERT_MENU field to determine whether to display the complete menu or only the expert menu. This control field is the first TDMS looks at; if the

CTL_EXPERT_MENU field is set to T, the request displays the expert menu, enters the selection in the menu selection record, and ends. These statements, taken from the menu request, perform those operations:

```
CONTROL FIELD IS CTL_EXPERT_MENU
  "T" :  USE FORM EXPERT_MENU_FORM;
         DEFAULT FIELD SELECTION_STRING_1;
         DEFAULT FIELD SELECTION_STRING_2;
         INPUT SELECTION_STRING_1 TO SELECTION_STRING_1;
         INPUT SELECTION_STRING_2 TO SELECTION_STRING_2;



END CONTROL FIELD;
```

If the CTL_EXPERT_MENU field is not set to T, the request then checks whether the entries to be displayed are the same ones as the last entries the user saw. If the value of the CTL_NEW_ENTRIES field of the control record is set to a value other than T, then the menu is the same as the last one. In this case, the request displays the last menu, accepts the selection string, and ends. The following statements, taken from the menu request, show this second way you can write the request:

```
CONTROL FIELD IS CTL_EXPERT_MENU



  NOMATCH:
    CONTROL FIELD IS CTL_NEW_ENTRIES



      NOMATCH:
               USE FORM MENU_FORM;
               DEFAULT FIELD SELECTION_STRING_1;
               DEFAULT FIELD SELECTION_STRING_2;
               INPUT SELECTION_STRING_1 TO SELECTION_STRING_1;
               INPUT SELECTION_STRING_2 TO SELECTION_STRING_2;
    END CONTROL FIELD;
END CONTROL FIELD;
END DEFINITION;
```

If the menu to be displayed does contain new entries, then ACMS has set the value of CTL_NEW_ENTRIES to T. In this case, the request displays the menu form and outputs the menu header. The request then begins displaying the entries to the indexed fields on the menu form. This work is done by the statements in the conditional instruction that begins "CONTROL FIELD IS ENTRY_FLAG". As long as there is an entry, which the request checks by looking at one of the fields to be displayed, the request continues to produce the entry information. As soon as there are no more entries, or when the request has displayed the sixteenth entry, the request stops displaying entries.

Before accepting the selection information from the user, the request checks the CTL_MULTI_PAGE field to see if there are more entries than fit on one screen. If so, ACMS sets the value of that field to T. If not, then the request does not display a message; there is an implicit NOMATCH in this control field. The request then accepts input from the user and ends.

The statements that follow show this third sequence of operations:

```
CONTROL FIELD IS CTL_EXPERT_MENU
```

```
  NOMATCH:
    CONTROL FIELD IS CTL_NEW_ENTRIES
      "T" :
            DISPLAY FORM MENU_FORM;
            OUTPUT MENU_HEADER_1 TO MENU_HEADER_1;
            OUTPUT MENU_HEADER_2 TO MENU_HEADER_2;

            CONTROL FIELD IS ENTRY_FLAG [1 TO 16]
               " " :
                    WAIT;
               NOMATCH  :
                    OUTPUT ENTRY_NUMBER [%LINE] TO ENTRY_NUMBER [%LINE];
                    OUTPUT ENTRY_KEY [%LINE] TO ENTRY_KEY [%LINE];
                    OUTPUT ENTRY_FLAG [%LINE] TO ENTRY_FLAG [%LINE];
                    OUTPUT ENTRY_TEXT [%LINE] TO ENTRY_TEXT [%LINE];
            END CONTROL FIELD;

            CONTROL FIELD IS CTL_MULTI_PAGE



            END CONTROL FIELD;
            INPUT SELECTION_STRING_1 TO SELECTION_STRING_1;
            INPUT SELECTION_STRING_2 TO SELECTION_STRING_2;



    END CONTROL FIELD;
END CONTROL FIELD;
END DEFINITION;
```

If there are more than 16 entries, which is the default number of entries for each screen, then the request tests the CTL_LAST_PAGE field to determine whether or not this is the last page of the menu displayed to the user. ACMS indicates the last page of the menu by setting the CTL_LAST_PAGE field to T. If it is the last page, the request displays the message "Press <RET> for first page". If it is not the last page of the menu, the request outputs the message "Press <RET> for more". It then accepts the selection from the user and ends.

The statements that follow show this fourth sequence of operations:

```
CONTROL FIELD IS CTL_EXPERT_MENU



  NOMATCH:
    CONTROL FIELD IS CTL_NEW_ENTRIES
      "T" :
            DISPLAY FORM MENU_FORM;
            OUTPUT MENU_HEADER_1 TO MENU_HEADER_1;
            OUTPUT MENU_HEADER_2 TO MENU_HEADER_2;

            CONTROL FIELD IS ENTRY_FLAG [1 TO 16]
               " " :
                    WAIT;
               NOMATCH  :
```

```
                    OUTPUT ENTRY_NUMBER [%LINE] TO ENTRY_NUMBER [%LINE];
                    OUTPUT ENTRY_KEY [%LINE] TO ENTRY_KEY [%LINE];
                    OUTPUT ENTRY_FLAG [%LINE] TO ENTRY_FLAG [%LINE];
                    OUTPUT ENTRY_TEXT [%LINE] TO ENTRY_TEXT [%LINE];
            END CONTROL FIELD;

            CONTROL FIELD IS CTL_MULTI_PAGE
              "T" :
                    CONTROL FIELD IS CTL_LAST_PAGE
        "T":
      CONTROL FIELD IS CTL_LAST_PAGE
          "T":
OUTPUT "              .......  Press <RET> for first page  ......."
                                  to MENU_MORE;
          NOMATCH:
OUTPUT "                  .......  Press <RET> for more  ......."
                                  to MENU_MORE;
      END CONTROL FIELD;

            END CONTROL FIELD;

            INPUT SELECTION_STRING_1 TO SELECTION_STRING_1;
            INPUT SELECTION_STRING_2 TO SELECTION_STRING_2;



    END CONTROL FIELD;
END CONTROL FIELD;
END DEFINITION;
```

The complexity of the ACMS menu request can provide several significant performance benefits. The instruction CONTROL FIELD IS ENTRY_FLAG ensures that the request performs an output mapping only when there is an entry to be displayed. Using USE FORM rather than DISPLAY FORM in the CONTROL FIELD IS CTL_NEW_ENTRIES instruction ensures that the request does not repaint the screen if a user selects an invalid selection. Removing these instructions simplifies the request but decreases ACMS performance.

B.2.5. Modifying the Menu Request

The default format for ACMS menus is set by the MENU_REQUEST request, which is included in the SYS$LIBRARY:ACMSREQ.RLB request library file. You must change the menu request if you want to:

- Change the names of fields in the form
- Add or remove input or output fields in the form
- Change the number of entries on the form
- Define program request keys available to users from the menu

Suppose you want to include a company logo on the menu and restrict that form to 12 entries. You also want to provide a program request key that would be equivalent to the terminal user **MENU** and **NOMENU** commands. First redefine the form, keeping the output fields for the entry information as an indexed array. If you kept the form field names the same, the new request, called NEW_MENU_REQUEST, would be like the one shown in *Example B.6, "Customized Menu Request"*.

Example B.6. Customized Menu Request

```
CREATE REQUEST NEW_MENU_REQUEST

RECORD IS MENU_HEADER_RECORD;
RECORD IS MENU_ENTRY_12_RECORD;
RECORD IS MENU_CONTROL_RECORD;
RECORD IS MENU_SELECTION_RECORD;

FORM IS NEW_MENU_FORM;
FORM IS EXPERT_MENU_FORM;


KEYPAD IS APPLICATION;

  PROGRAM KEY IS KEYPAD "0"
    NO CHECK;
    RETURN "$MENU" TO SELECTION_STRING_1;
  END PROGRAM KEY;

  PROGRAM KEY IS KEYPAD "."
    NO CHECK;
    RETURN "$NOMENU" TO SELECTION_STRING_1;
  END PROGRAM KEY;


CONTROL FIELD IS CTL_EXPERT_MENU
  "T":
    USE FORM EXPERT_MENU_FORM;
    DEFAULT FIELD SELECTION_STRING_1;
    DEFAULT FIELD SELECTION_STRING_2;
    INPUT SELECTION_STRING_1 TO SELECTION_STRING_1;
    INPUT SELECTION_STRING_2 TO SELECTION_STRING_2;


  NOMATCH:
    CONTROL FIELD IS CTL_NEW_ENTRIES
      "T":
  DISPLAY FORM NEW_MENU_FORM;

       OUTPUT MENU_HEADER_1 TO MENU_HEADER_1;
        OUTPUT MENU_HEADER_2 TO MENU_HEADER_2;

        CONTROL FIELD IS ENTRY_FLAG [1 TO 12]
          " ":
            WAIT;
          NOMATCH:
            OUTPUT ENTRY_NUMBER [%LINE] TO ENTRY_NUMBER [%LINE];
            OUTPUT ENTRY_KEY [%LINE] TO ENTRY_KEY [%LINE];
            OUTPUT ENTRY_FLAG [%LINE] TO ENTRY_FLAG [%LINE];
            OUTPUT ENTRY_TEXT [%LINE] TO ENTRY_TEXT [%LINE];
        END CONTROL FIELD;


  CONTROL FIELD IS CTL_MULTI_PAGE
    "T":
    CONTROL FIELD IS CTL_LAST_PAGE
      "T":
```

```
OUTPUT "              .......  Press <RET> for first page  ......."
                                    to MENU_MORE;
            NOMATCH:
OUTPUT "                  .......  Press <RET> for more  ......."
                                    to MENU_MORE;
    END CONTROL FIELD;

  END CONTROL FIELD;


  INPUT SELECTION_STRING_1 TO SELECTION_STRING_1;
  INPUT SELECTION_STRING_2 TO SELECTION_STRING_2;
      NOMATCH:
  USE FORM NEW_MENU_FORM;
  DEFAULT FIELD SELECTION_STRING_1;
  DEFAULT FIELD SELECTION_STRING_2;
  INPUT SELECTION_STRING_1 TO SELECTION_STRING_1;
  INPUT SELECTION_STRING_2 TO SELECTION_STRING_2;
  END CONTROL FIELD;
END CONTROL FIELD;

END DEFINITION;
```

The changes to the menu request are:

- Replacing MENU_FORM by NEW_MENU_FORM throughout the definition
- The NEW_MENU_FORM definition should include only 12 indexed fields
- Changing the indexed array output from 16 records to 12 records
- Changing the keypad to APPLICATION and adding two program request keys

If you want all menus in the ACMS system to use the revised menu format, use the name MENU_REQUEST for the revised request and MENU_FORM for the revised form. In the definition for every menu using this new request you must include the clause:

```
REQUEST IS MENU_REQUEST
  WITH 12 ENTRIES;
```

Including this clause in the menu definition tells ACMS to display the menu using the request for the customized menu. You must also rebuild the request library, as explained in *Section B.2.8, "Modifying and Building the ACMS Menu Request Library"*.

However, if you want some menus to use the default ACMS format and others use the new format, you must:

1. Create the new form, assigning a name other than MENU_FORM to the form you have created.
2. Create a copy of the menu request, assigning a name other than MENU_REQUEST.
3. Replace the FORM IS MENU_FORM instruction in the new menu request with an instruction that names your new form.
4. Include the new request in the menu library definition and rebuild the request library, as explained in *Section B.2.8, "Modifying and Building the ACMS Menu Request Library"*.

Then use the REQUEST clause in the definitions for any menus that you want to use the new format, declaring the name of the new request. Also declare the number of entries for the menu if the number is other than the default value of 16. The next section explains the REQUEST clause.

B.2.6. Using the REQUEST Clause

When you use the REQUEST clause in a menu definition, you are telling ACMS to use a request other than MENU_REQUEST for that menu.

Example B.7, "Menu Definition Using REQUEST Clause" shows an example of a menu definition that uses the REQUEST clause.

Example B.7. Menu Definition Using REQUEST Clause

```
CREATE MENU PERSONNEL_MENU
  HEADER IS "          P E R S O N N E L     M E N U";
  REQUEST IS PERSONNEL_MENU_REQUEST WITH 12 ENTRIES PER SCREEN;
  DEFAULT APPLICATION IS "PERSONNEL";
  ENTRIES ARE
    ADD  : TASK IS ADD_EMPLOYEE;

           TEXT IS "Add a new employee record";



  END ENTRIES;
END DEFINITION;
```

In the REQUEST clause, you must include the given name of the request you want ACMS to use for that menu. This is not the CDD path name of the request, but the name by which the request is listed in the request library definition.

Note

The number in the ENTRIES PER SCREEN phrase in the REQUEST clause must correspond to the number of form fields to which the request writes entry information. The default number of entries for each screen is 16.

If you want more or less than 16 entries on each screen, you must use the WITH ENTRIES PER SCREEN phrase to define the number of entries you want. You must also define a request to handle that many entries. You can change the number of entries only if you have defined a request that is different from the ACMS-supplied request. You cannot use the WITH ENTRIES PER SCREEN phrase to change the number of entries that display on the menus supplied by ACMS, unless you modify the request.

You can include more entries in the menu definition than the number you specify in the WITH ENTRIES PER SCREEN phrase. For example, you could include 13 or more entries in the menu definition shown in *Example B.7, "Menu Definition Using REQUEST Clause"* even though the WITH ENTRIES PER SCREEN phrase specifies 12 entries for each screen. In this case, when ACMS displays the menu it also displays the message:

```
"....... Press <RET> for more .......".
```

If the user presses **Return**, ACMS displays the additional entries listed in the menu definition.

You can include the REQUEST clause in any menu definition. If you want two menus with the same format, and that format is different from the ACMS default format, you must include the REQUEST clause in both definitions.

You can also use different menu requests in different menu definitions. There is no restriction on the number of menu requests for each ACMS system, but using different menu requests for different menu definitions can decrease ACMS system performance.

B.2.7. Changing the MENU_ENTRY_RECORD

If you decrease the number of entries displayed by the request, you do not have to change any of the records passed to the request. However, if you increase the number of entries, you must also change the definition for the record, MENU_ENTRY_RECORD, that ACMS passes to the request.

The number used in the LINE STRUCTURE statement in the record definition must be the same as or larger than the number of entries displayed by the request. ACMS passes only as many entries in MENU_ENTRY_RECORD as the menu definition specifies, regardless of the size of MENU_ENTRY_RECORD. However, if MENU_ENTRY_RECORD is smaller than the number of entries specified by the menu definition, the request can get access violation errors when it tries to read the record.

The MENU_ENTRY_RECORD description is included in the ACMSREQ.BAK file. When you restore that file to the CDD, the MENU_ENTRY_RECORD description is loaded into the same directory as MENU_FORM and MENU_REQUEST. Extract that description from the CDD, putting it in a text file. Then change the record description to match the number of entries in the menu request. *Example B.8, "Record Definition for MENU_ENTRY_RECORD with 12 Entries"* shows the record description for a MENU_ENTRY_RECORD containing 12 entries.

Example B.8. Record Definition for MENU_ENTRY_RECORD with 12 Entries

```
DEFINE RECORD MENU_ENTRY_RECORD
    DESCRIPTION IS /* RECORD FOR THE ACMS MENU ENTRY INFO */.
MENU_ENTRY_RECORD STRUCTURE.
    LINE STRUCTURE OCCURS 12 TIMES.
        ENTRY_NUMBER      DATATYPE TEXT    2.
        ENTRY_KEY         DATATYPE TEXT    10.
        ENTRY_FLAG        DATATYPE TEXT    1.
        ENTRY_TEXT        DATATYPE TEXT    50.
    END LINE STRUCTURE.
END MENU_ENTRY_RECORD STRUCTURE.
END MENU_ENTRY_RECORD.
```

The only item you should ever change in this record definition is the decimal number in the "LINE STRUCTURE OCCURS n TIMES" statement.

When you have edited the file, use the Common Data Dictionary Data Definition Language (CDDL) Utility to write the new record definition into the CDD. Use a different name for the new record definition and in the request definition; in that case, you must use the same name when referring to the record definition from the request. If all your menus use the same or fewer entries than the number used in the new record definition, you can delete the existing MENU_ENTRY_RECORD definition from the CDD and use the same name for the new record. You can then use the CDDL Utility to write the new MENU_ENTRY_RECORD definition into the CDD. For information on storing definitions in the CDD, refer to the CDD documentation.

Although you can change the name of MENU_ENTRY_RECORD, you cannot include additional records in a menu request. Each menu request must use only four records, in the order in which they

occur in MENU_REQUEST. Do not change the definitions for the other menu records. However, you can include additional forms in the request, such as when you use a conditional request.

If you change the name of the MENU_ENTRY_RECORD in the CDD, be sure to change the name in the request as well. Also, when you modify the menu form and request definition, or when you reload the record description into the CDD, be sure that you use the same CDD directory for all three definitions.

B.2.8. Modifying and Building the ACMS Menu Request Library

Once you have changed the menu form and, if necessary, the menu request and record, you have to rebuild the menu request library.

If you have created a new menu request, use the TDMS Request Definition Utility (RDU) to add to MENU_LIBR the request you have created. *Example B.9, "MENU_LIBR Request Library Definition"* shows a MENU_LIBR request library definition that includes all the requests that ACMS supplies and an additional menu request called NEW_MENU_REQUEST.

Example B.9. MENU_LIBR Request Library Definition

```
REQUEST IS COMD_REQUEST;
REQUEST IS MENU_REQUEST;
REQUEST IS RESET_SCREEN;
REQUEST IS CLEAR_SCREEN;
REQUEST IS NEW_MENU_REQUEST;
END DEFINITION;
```

The request name you use in the REQUEST clause of the menu definition must be the name of the request in the run-time version of the request library. By default, the run-time name of the request is the same as the CDD given name of the request as it is used in the request library definition. However, in the REQUEST IS instructions of the request library definition you can include a WITH NAME phrase to assign a unique name to the new menu request. In this case, you must use the same unique name in the REQUEST clause of the menu definition.

To edit the request library definition, either modify the definition directly in the CDD or extract the definition, revise it, and then reload it into the CDD. Once you have edited the request library definition, you can build the request library.

Before building the library, make sure that the request definition, form definition, record definition, and request library definition are all in the CDD directory in which you restored the contents of the ACMSREQ.BAK file. Then use the RDU **BUILD** command from TDMS :

```
$ RUN SYS$SYSTEM:RDU
RDU> BUILD LIBRARY MENU_LIBR ACMSREQ.RLB /LIST=ACMSREQ.LIS
RDU> EXIT
```

For explanations of the **BUILD** command and of the error messages you get, see *VAX TDMS Request and Programming Manual*.

If the **BUILD** command is successful, move the new menu request library to the SYS$LIBRARY directory. Instead of putting the request library into SYS$LIBRARY, you can redefine the logical name ACMS$REQLIB to point to the request library. This name must be defined either as a system logical name or as a group logical name, in the same group as the user name under which the ACMS Command Process (CP) is running. See *VSI ACMS for OpenVMS Getting Started* [https://docs.vmssoftware.com/vsi-

acms-get-started-guide/] for an explanation of the command process. Redefining ACMS$REQLIB makes it easy to switch back to the existing menu request library if you discover problems with the new one.

You can then build the menu definition pointing to the new request and try using the new menu. In order to make the new version of the menu request library available to the command process, you must stop and restart the ACMS terminal subsystem, as by using the **ACMS/START TERMINALS** and **ACMS/STOP TERMINALS** commands. Make sure your ACMS user definition points to the menu database containing the menu that uses the new menu format. Then sign in to ACMS and check whether or not:

- You can select the menu.
- The menu has the correct number of entries.
- You can select the entries.
- The fields are in the correct places.
- The cursor moves from field to field in the correct order.

If ACMS cannot display the new menu, you get this error message when signing in to ACMS:

```
Error while trying to display menu - bad MDB.
```

If you get this message, make sure you have put the request library in SYS$LIBRARY or that the system logical name ACMS$REQLIB is pointing to the directory that contains the menu request library. If the library is there, check the record, form, request, and library definitions for unmatched names, unmatched CDD locations, inappropriate file protection, or differences in the number of entries.

B.3. Disabling the SELECT Command in the ACMS Command Menu

The **SELECT** command appears on the ACMS Command Menu and permits a user to select tasks by application and task name. In order to disable the **SELECT** command, you need to remove it from the ACMS Command Menu by editing and replacing the command menu definitions, and rebuilding the command database, as follows:

1. Rename the file SYS$LIBRARY:ACMS$MENU_CUST.COM to ACMSMENU.COM. This file contains the ACMS command menu and the ACMS default menu.
2. Edit the file ACMSMENU.COM and delete or comment out the **SELECT** command entry. The exclamation points (!) in the following example mark the lines you need to comment out or delete.

```
REQUEST IS COMD_REQUEST WITH 8 ENTRIES;
HEADER IS "                        ACMS Command Menu";
ENTRIES ARE
  "Continue":
    TEXT IS "Continue interrupted operation";
    COMMAND 3;
  "Exit":
    TEXT IS "Leave the ACMS menu environment";
    COMMAND 8;
  "Help":
    TEXT IS "Display ACMS help information on the terminal";
    COMMAND 17;
```

```
  "Menu":
    TEXT IS "Display selection and command menus";
    COMMAND 18;
  "Nomenu":
    TEXT IS "Do not display selection and command menus";
    COMMAND 19;
  !"Select":
  !  TEXT IS "Select task by application name and task name";
  !  COMMAND 20;
END ENTRIES;
END DEFINITION;
```

3. Change the CDD path names specified in ACMSMENU.COM to be the CDD path name of your choice.

4. Use the following ADU commands to insert the definition file ACMSMENU.COM in the CDD:

```
$ ADU
ADU> @ACMSMENU.COM
```

5. Use the following ADU commands to rebuild the ACMS command menu database:

```
$ ADU
ADU> BUILD MENU COMMAND$ROOT ACMS$DIRECTORY:ACMSCMD.MDB
ADU> EXIT
```

Appendix C. Using CDO for Pieces Tracking

Chapter 1, "Writing Definitions with ADU" describes using the ADU utility to place ACMS definitions in the CDD. In some ACMS applications, you may want to track relationships among ACMS entities in the CDD.

C.1. Overview of Dictionary Object Types

The CDD dictionary allows you to see the relationships among entities. You can only create relationships for ACMS entities that are in CDO format. You cannot track DMU format objects. Some relationships are established automatically; for example, if you create an Rdb record entity, the relationship between the record and each of its fields is defined for you. For ACMS entities, you must define the relationships using the CDO Utility.

To define relationships you need to know more about the three distinct types of dictionary objects in CDD:

- Entity

  An entity is the dictionary object that contains the definition of, for example, an ACMS task definition or an Rdb record. You use ADU commands to create entities for ACMS applications, menus, tasks, and task groups. Use the CDO Utility to create any other ACMS dictionary entities needed to create relationships. You must create dictionary entities for your ACMS definitions and for the database objects required by your ACMS applications.

- Relationship

  A relationship is the dictionary object that links two entities, modeling a functional link between the application objects those entities describe. For example, you can create a relationship that links a task group to an application, and one that links the task group to a task. You can create relationships only with the CDD CDO Utility. If you do not want to do pieces tracking, you do not need to create relationships.

- Attribute

  An attribute describes a specific characteristic of a CDD entity or relationship. Attributes are the basic units of information contained in a CDD dictionary. You do not need to create attributes for ACMS dictionary entities or relationships.

Within the dictionary, each particular entity is identified by the name you give it and by a predefined protocol name for each type of entity.

Protocols govern the use of the dictionary objects. A protocol is a set of rules controlling the entities, relationships, and attributes of an item in the dictionary (for example, an ACMS task definition or an Rdb record). If you plan to use the dictionary without pieces tracking, you need not concern yourself with protocols; the dictionary automatically associates your ACMS definition with the appropriate protocol. However, if you plan to do pieces tracking, you must use protocols as you:

- Create ACMS entities needed for pieces tracking that cannot be created from ADU

- Define relationship between entities for which relationship protocols exist

Section C.2, "Creating Relationships Between Entities" describes how to use CDO to create relationships for pieces tracking.

The CDO **DIRECTORY** command lists the entities within the current dictionary directory. The left column lists the specific entities by name. The right column lists the protocol for each type of entity listed to the left. For example:

1. Use the ADU **REPLACE TASK** command to create a dictionary entity for your TASK_ADD definition, and one for your TASK_UPDATE definition.

2. Use the CDO **DIRECTORY** command at the dictionary directory where you placed TASK_ADD. CDO lists the entity name on the left and the protocol for that type of entity on the right:

```
      .                .
      .                .
      .                .
TASK_ADD         ACMS$TASK
TASK_UPDATE      ACMS$TASK
      .                .
      .                .
      .                .
```

C.2. Creating Relationships Between Entities

CDD contains protocols for all ACMS entity types and ACMS relationship types. When you use an ADU command to place an ACMS definition in CDD, the dictionary automatically associates the entity with one of the following four protocols:

- ACMS$APPLICATION – for an ACMS application definition
- CDO$MENU – for an ACMS menu definition
- ACMS$TASK – for an ACMS task definition
- ACMS$TASK_GROUP – for an ACMS task group definition

Although ADU commands create application, menu, task, and task group entities, you may need other ACMS dictionary entities to create relationships. If you want to do pieces tracking, you must use the CDO Utility to:

1. Define all other entities needed to establish a relationship, associating each entity with a predefined protocol for that entity type.

   For example, if you use the command ADU **REPLACE GROUP MYGROUP**, **MYGROUP** automatically enters CDD as the MYGROUP entity, based on the ACMS$TASK_GROUP protocol. In this example, you establish relationships between the MYGROUP entity and:

   - Server entities, based on the ACMS$SERVER protocol
   - Record entities, based on the CDD$DATA_AGGREGATE protocol
   - A .TDB or .RLB file entity, based on the CDD$FILE protocol

- Task entities, based on the ACMS$TASK protocol

In this example, only the task and group entities can be created through the ACMS ADU commands. You must create the other entities through the CDO Utility.

Use CDO **DEFINE GENERIC** to create an ACMS dictionary entity other than the four that are created automatically (that is, other than application, menu, task, and task group).

```
CDO> DEFINE GENERIC ACMS$SERVER MYSERVER
CDO> CDD$PROCESSING_NAME 'MYSERVER'.
CDO> END ACMS$SERVER MYSERVER.
```

Refer to the CDD documentation for complete syntax for this command.

2. Define relationships between entities for which relationship protocols exist.

   You must define all relationships through the CDO Utility. Each relationship consists of an owner and a member. For example, a task entity is the owner and a task procedure item is the member in a relationship between a task and a task procedure item. You can create relationships only between entities for which CDD supplies a relationship protocol. See *Table C.1, "CDD Protocols for ACMS Entity and Relationship Objects"* for a list of all the protocols used to create relationships between ACMS entities, and the entities which can be connected by these relationships.

   You use the CDO **DEFINE GENERIC** or CDO **CHANGE GENERIC** commands to create relationships between specific instances of ACMS dictionary entities. **DEFINE GENERIC** allows you to create both the entity and the relationship to another entity. **CHANGE GENERIC** creates a relationship between already existing entities, or changes the relationship to connect to a different member entity. The CDO **CHANGE GENERIC** syntax is:

   ```
   CDO> CHANGE GENERIC protocol-name entity-name. DEFINE relationship-name
   cont> relationship-mbr END relationship-name DEFINE.
   ```

   For example:

   ```
   CDO> CHANGE GENERIC ACMS$TASK_GROUP MYGROUP. DEFINE
   cont> ACMS$TASK_GROUP_SERVER MYSERVER. END
   cont> ACMS$TASK_GROUP_SERVER DEFINE.
   cont> END MYGROUP ACMS$TASK_GROUP
   ```

   See the CDD documentation set for complete syntax and for the CDO **CHANGE GENERIC** command.

   A description of the parts of this command follows:

   - ACMS$TASK_GROUP is the protocol name of the owner entity in the relationship.
   - MYGROUP is the name you gave the task group entity. This entity was created and associated with ACMS$TASK_GROUP when you placed the MYGROUP task group definition in the dictionary.
   - ACMS$TASK_GROUP_SERVER is the CDD protocol for a relationship between a task group entity and a server entity in that task group.
   - MYSERVER is the name of an ACMS server entity you created with the CDO **DEFINE GENERIC** command. MYSERVER is the member entity in the relationship.

MYSERVER must be based on the ACMS$SERVER protocol. Only an entity based on the ACMS$SERVER protocol can be used to create a relationship between the task group and server entities.

Relationships are always one-to-one, but you can establish multiple relationships for any entity. For example, you can create a relationship between an ACMS menu entity and an ACMS task entity, or between the same task entity and a DECforms form entity. You can then use CDO commands such as **SHOW USES** and **SHOW USED_BY** to do pieces tracking.

If you create a relationship and subsequently change the member entity, CDD assigns a message to the owner entity. At the next use of the owner entity, ADU informs you that there are messages pending for the owner entity. This situation can occur after you create a relationship between a task and a task group. For example, if you change a task in task group RMS_GROUP and then build the group, ADU issues the message:

```
Object RMS_GROUP has CDD messages
```

See the CDD documentation set for complete information about these commands.

The left column of *Table C.1, "CDD Protocols for ACMS Entity and Relationship Objects"* lists all the dictionary protocols for ACMS entities. The table also lists the protocols for entities eligible for relating to any entities of each protocol in the left column. Finally, the table lists the relationship protocols you can use to relate the eligible entities.

Table C.1. CDD Protocols for ACMS Entity and Relationship Objects

Protocol for Entity of This Type		
ACMS$APPLICATION	*Entities of this protocol defined by ADU commands in ACMS application definition.*	
	Owns Entities with Protocol:	**By Relationship:**
	ACMS$APPL_TASK_ITEM	ACMS$APPLICATION_ TASK
	ACMS$TASK_GROUP	ACMS$APPLICATION_ TASK_GROUP
	ACMS$APPL_SERVER_ITEM	ACMS$APPLICATION_ SERVER
	CDDFILE	ACMS$APPLICATION_ ADB_FILE
	Member of Entity with Protocol:	**By Relationship:**
	ACMS$MENU_TASK_ITEM	ACMS$MENU_TASK_APPL
ACMS$APPL_SERVER_ITEM	*Entities of this protocol defined through CDD CDO Utility. Defined by server and group that server is in.*	
	Owns Entities with Protocol:	**By Relationship:**
	ACMS$SERVER	ACMS$APPLICATION_ SRV_SRV
	ACMS$TASK_GROUP	ACMS$APPLICATION_ SRV_GRP
	Member of Entity with Protocol:	**By Relationship:**

Protocol for Entity of This Type		
	ACMS$APPLICATION	ACMS$APPLICATION_ SERVER
ACMS$APPL_TASK_ITEM	*Entities of this protocol defined through CDD CDO Utility. Defined by task name and group in which task appears.*	
	Owns Entities with Protocol:	**By Relationship:**
	ACMS$TASK	ACMS$APPLICATION_ TASK_TSK
	ACMS$TASK_GROUP	ACMS$APPLICATION_ TASK_GROUP
	Member of Entity with Protocol:	**By Relationship:**
	ACMS$APPLICATION	ACMS$APPLICATION_ TASK
	ACMS$MENU_TASK_ITEM	ACMS$MENU_TASK_TASK
CDD$MENU	*Entities of this protocol defined by ADU commands in ACMS menu definition.*	
	Owns Entities with Protocol:	**By Relationship:**
	ACMS$MENU_TASK_ITEM	ACMS$MENU_TASK
	CDD$FILE	ACMS$MENU_MDB_FILE
	CDD$MENU	ACMS$MENU_CONTAINS
	Member of Entity with Protocol:	**By Relationship:**
	CDD$MENU	ACMS$MENU_CONTAINS
ACMS$PROCEDURE	*Entities of this protocol defined through CDD CDO Utility.*	
	Owns Entities with Protocol:	**By Relationship:**
	CDD$EXECUTABLE_ IMAGE	ACMS$PROCEDURE_ ENTRY_PT
	CDD$PROCEDURE	ACMS$PROCEDURE_ ENTRY_PT
	Member of Entity with Protocol:	**By Relationship:**
	ACMS$SERVER	ACMS$SERVER_ ABORT_PROCEDURE
	ACMS$SERVER	ACMS$SERVER_ ACTION_PROCEDURE
	ACMS$SERVER	ACMS$SERVER_ EXIT_PROCEDURE
	ACMS$SERVER	ACMS$SERVER_ INIT_PROCEDURE
	ACMS$TASK_ PROCEDURE_ITEM	ACMS$PROCEDURE_ PROCEDURE
ACMS$SERVER	*Entities of this protocol defined through CDD CDO Utility.*	

Protocol for Entity of This Type		
	Owns Entities with Protocol:	**By Relationship:**
	ACMS$PROCEDURE	ACMS$SERVER_ ABORT_PROCEDURE
	ACMS$PROCEDURE	ACMS$SERVER_ ACTION_PROCEDURE
	ACMS$PROCEDURE	ACMS$SERVER_ EXIT_PROCEDURE
	ACMS$PROCEDURE	ACMS$SERVER_ INIT_PROCEDURE
	ACMS$SERVER	ACMS$SERVER_ BASED_ON
	CDD$COMPILED_MODULE	ACMS$SERVER_ MODULE
	Member of Entity with Protocol:	**By Relationship:**
	ACMS$APPL_SERVER_ITEM	ACMS$APPLICATION_ SRV_SRV
	ACMS$SERVER	ACMS$SERVER_ BASED_ON
	ACMS$TASK_GROUP	ACMS$TASK_ GROUP_SERVER
	ACMS$TASK_ PROCEDURE_ITEM	ACMS$_PROCEDURE_ SERVER
ACMS$TASK	*Entities of this protocol defined by ADU commands in ACMS task definition.*	
	Owns Entities with Protocol:	**By Relationship:**
	ACMS$TASK	ACMS$TASK_ BASED_ON
	ACMS$TASK_TASK_ITEM	ACMS$TASK_TASK
	ACMS$TASK_ PROCEDURE_ITEM	ACMS$TASK_ PROCEDURE
	CDD$VIDEO_DISPLAY	ACMS$TASK_ VIDEO_DISPLAY
	CDD$DATA_AGGREGATE	ACMS$TASK_ DATA_AGGREGATE
	Member of Entity with Protocol:	**By Relationship:**
	ACMS$APPL_TASK_ITEM	ACMS$APPLICATION_ TASK_TSK
	ACMS$TASK	ACMS$TASK_ BASED_ON
	ACMS$TASK_GROUP	ACMS$TASK_GROUP_TASK
	ACMS$TASK_TASK_ITEM	ACMS$TASK_ITEM_TASK

Protocol for Entity of This Type		
ACMS$TASK_GROUP	*Entities of this protocol defined by ADU commands in ACMS task group definition.*	
	Owns Entities with Protocol:	**By Relationship:**
	ACMS$TASK	ACMS$TASK_GROUP_ TASK
	ACMS$TASK_GROUP	ACMS$TASK_GROUP_ BASED_ON
	ACMS$SERVER	ACMS$TASK_GROUP_ SERVER
	CDD$FILE	ACMS$TASK_GROUP_ MSG_FILE
	CDD$FILE	ACMS$TASK_GROUP_ TDB_FILE
	CDD$VIDEO_DISPLAY	ACMS$TASK_GROUP_ VIDEO_DISPLAY
	CDD$DATA_AGGREGATE	ACMS$TASK_GROUP_ DATA_AGG
	Member of Entity with Protocol:	**By Relationship:**
	ACMS$APPLICATION	ACMS$APPLICATION_ TASK_GROUP
	ACMS$APPL_SRV_ITEM	ACMS$APPL_SRV_GRP
	ACMS$APPL_TASK_ITEM	ACMS$APPLICATION_ TASK_GRP
	ACMS$TASK_GROUP	ACMS$TASK_GROUP_ BASED_ON
ACMS$TASK_ PROCEDURE_ITEM	*Entities of this protocol defined through CDD CDO Utility. Defined by procedure and server for that procedure.*	
	Owns Entities with Protocol:	**By Relationship:**
	ACMS$PROCEDURE	ACMS$PROCEDURE_ PROCEDURE
	ACMS$SERVER	ACMS$PROCEDURE_ SERVER
	CDD$DATA_AGGREGATE	ACMS$PROCEDURE_ DATA_AGGREGATE
	Member of Entity with Protocol:	**By Relationship:**
	ACMS$TASK	ACMS$TASK_ PROCEDURE
ACMS$TASK_TASK_ITEM	*Entities of this protocol defined through CDD CDO Utility. Defined by task renamed in group, and task arguments.*	
	Owns Entities with Protocol:	**By Relationship:**
	ACMS$TASK	ACMS$TASK_

Protocol for Entity of This Type		
		ITEM_TASK
	CDD$DATA_AGGREGATE	ACMS$TASK_ITEM_ DATA_AGGREGATE
	Member of Entity with Protocol:	**By Relationship:**
	ACMS$TASK	ACMS$TASK_TASK
CDD$COMPILED MODULE	*Entities of this protocol defined through CDD CDO Utility.*	
	Owns Entities with Protocol:	**By Relationship:**
	CDD$EXECUTABLE_ IMAGE	CDD$IMAGE_ DERIVED_FROM
	Member of Entity with Protocol:	**By Relationship:**
	ACMS$SERVER	ACMS$SERVER_ MODULE
CDD$EXECUTABLE_IMAGE	*Entities of this protocol defined through CDD CDO Utility.*	
	Owns Entities with Protocol:	**By Relationship:**
	none	
	Member of Entity with Protocol:	**By Relationship:**
	ACMS$PROCEDURE	ACMS$PROCEDURE_ ENTRY_PT
	CDD$COMPILED_MODULE	CDD$IMAGE_ DERIVED_FROM
CDD$FILE	*Entities of this protocol defined through CDD CDO Utility.*	
	Owns Entities with Protocol:	**By Relationship:**
	none	
	Member of Entity with Protocol:	**By Relationship:**
	ACMS$APPLICATION	ACMS$APPLICATION_ ADB_FILE
	CDD$MENU	ACMS$MENU_MDB_FILE
	ACMS$TASK_GROUP	ACMS$TASK_GROUP_ TDB_FILE
	ACMS$TASK_GROUP	ACMS$TASK_GROUP_ MSG_FILE
CDD$PROCEDURE	*Entities of this protocol defined through CDD CDO Utility.*	
	Owns Entities with Protocol:	**By Relationship:**
	none	
	Member of Entity with Protocol:	**By Relationship:**
	ACMS$PROCEDURE	ACMS$PROCEDURE_ ENTRY_PT

Protocol for Entity of This Type		
CDD$VIDEO_DISPLAY	*Defined by*	
	Owns Entities with Protocol:	**By Relationship:**
	CDD$VIDEO_DISPLAY	ACMS$VIDEO_DISPLAY_ BASED_ON
	Member of Entity with Protocol:	**By Relationship:**
	ACMS$TASK	ACMS$TASK_ VIDEO_DISPLAY
	ACMS$TASK_GROUP	ACMS$TASK_GROUP_ VIDEO_DISPLAY
CDD$DATA_AGGREGATE	*Defined by*	
	Owns Entities with Protocol:	**By Relationship:**
	CDD$DATA_AGGREGATE	CDD$DATA_ AGGREGATE_BASED_ON
	Member of Entity with Protocol:	**By Relationship:**
	ACMS$TASK	ACMS$TASK_DATA_AGG
	ACMS$TASK_GROUP	ACMS$TASK_GROUP_ DATA_AGG
	ACMS$TASK_ PROCEDURE_ITEM	ACMS$TASK_PROCEDURE_ ITEM_DATA_AGG
	ACMS$TASK_TASK_ITEM	ACMS$TASK_ITEM_ DATA_AGG
	CDD$DATA_AGGREGATE	CDD$DATA_AGGREGATE_ BASED_ON

Appendix D. Using LSE with ACMS

This appendix provides an overview of the optional Language-Sensitive Editor (LSE) productivity tool. ACMS software does not include this tool; you must purchase it separately. For information on how to purchase this tool, contact your VSI sales representative.

D.1. Using LSE with ACMS

LSE is a text editor designed specifically for software development with features that help you produce syntactically correct ACMS code. With LSE, you can:

- Edit programs or text files using standard text editor commands with either the EDT or EVE keypad
- Use formatted language constructs to quickly develop syntactically correct programs
- Use two windows and multiple buffers
- Review and correct compilation errors from the editing session
- Customize your editing environment
- Integrate with other VAXset tools and products

In ACMS, you can use LSE for creating application, menu, task, and task group definitions for processing by the Application Definition Utility (ADU). Some of the key features of LSE when used with ACMS include:

- Creating source code
- Using placeholders and tokens
- Reviewing diagnostic files
- Getting ACMS-specific help for placeholders and tokens created by LSE

D.2. Creating ACMS Source Files with LSE

There are five LSE templates, one each for syntax for ACMS application definitions, task group definitions, task definitions, and menu definitions. The fifth template can be used for any of the four ACMS definition types.

There are two ways to invoke an LSE template for an ACMS definition. At the DCL prompt, do one of the following:

- Issue the **LSEDIT** command followed by a file name with one of these file extensions:
  - .ADF – For the ACMS application definition template.
  - .GDF – For the ACMS task group definition template.

  - .TDF – For the ACMS task definition template.
  - .MDF – For the ACMS menu definition template.
  - .ADU – Invokes LSE with a template that can be used for any of the four preceding ACMS definitions. You can include more than one type of ACMS definition in a file with a .ADU extension.

- Issue the **LSEDIT** command with the **/LANGUAGE** qualifier and one of the following languages, followed by a file name with any file extension:
  - ACMS$APPLICATION – For the ACMS application definition template.
  - ACMS$TASK_GROUP – For the ACMS task group definition template.
  - ACMS$TASK – For the ACMS task definition template.
  - ACMS$MENU – For the ACMS menu definition template.

For example:

```
$ LSEDIT ADDCAR.TDF
```

or:

```
$ LSEDIT/LANGUAGE = ACMS$TASK ADDCAR.TDF
```

The ADU file type tells LSE to use the ADU template for a task group definition.

You can also use LSE as the editor of choice with ADU **EDIT** and **MODIFY** commands. To do this, you must define the ADU$EDIT logical name to point to a command procedure that invokes LSE. For example, in the following command line, the ADU$EDIT process logical name points to the command procedure LSEDIT.COM:

```
$ DEFINE ADU$EDIT "LSEDIT.COM"
```

The command procedure LSEDIT.COM actually invokes LSE. You can devise your own command procedure to perform this task or you can use the one shown in *Example D.1, "LSEDIT.COM File"*.

Example D.1. LSEDIT.COM File

```
$ ! LSEDIT.COM - Invoke VAXLSE for ADU
$ !
$ ! Inputs:
$ !
$ ! P1 = Input file name
$ ! P2 = Output file name
$ !
$ ! Note that this procedure is run in the context of a spawned
$ ! subprocess.  Though LOGIN.COM is not executed when this
$ ! subprocess starts, the spawning procedure copies the symbols
$ ! and logicals from the spawning process.
$ ! The default directory for the subprocess is the same as
$ ! that for the spawning process.
$ !
$ ASSIGN/USER 'F$LOGICAL("SYS$OUTPUT")' SYS$INPUT
```

```
$ IF P1 .EQS. "" THEN GOTO NOINPUT
$ LSEDIT /LANGUAGE=ACMSADU /OUTPUT='P2' 'P1'
$ EXIT
$ NOINPUT:
$ LSEDIT /LANGUAGE=ACMSADU 'P2'
```

This command procedure accepts a definition file or command file for input. It then invokes LSE, edits the definition file or command file using ACMSADU language type, and produces a listing file if specified.

When you invoke LSE to create a new file with the .TDF extension, your terminal screen displays the text:

```
{TASK_definition}
```

Similarly, when you invoke LSE with the .ADF, .GDF, or .MDF extensions, your terminal screen displays initial template information for application, task group, and menu definitions.

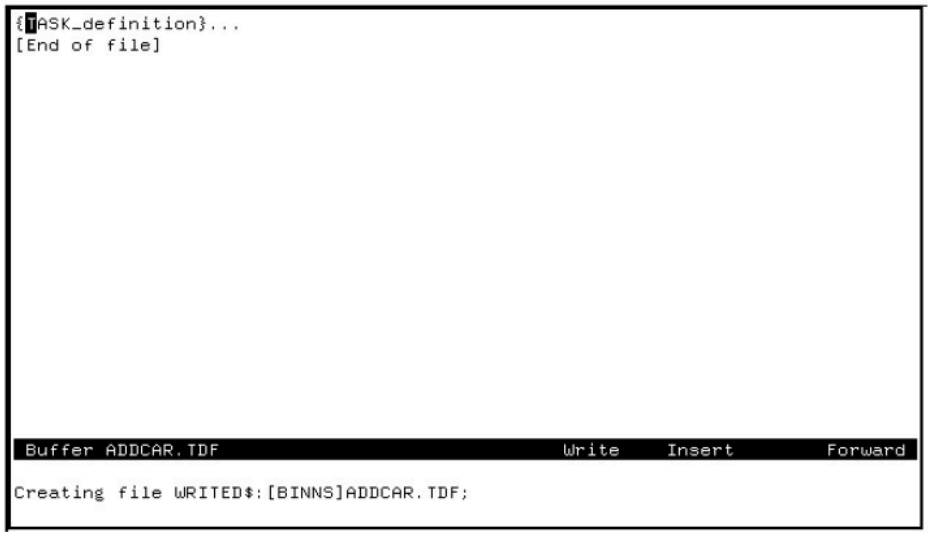
D.2.1. Using Placeholders and Tokens in LSE

At the bottom of the screen shown in *Figure D.1, "Creating a New File with LSE"*, LSE displays a bar containing TPU editing information. The phrase TASK_definition appears at the top of the screen. LSE provides this phrase, called a **placeholder**, as a starting template to aid you in coding your application. LSE surrounds a placeholder with:

- A pair of curly braces ({}) to represent a place in the source code where the user must provide additional program text
- A pair of square brackets ([]) to represent optional syntax which the user can delete or modify

The curly braces surrounding the phrase TASK_definition indicate that the phrase represents a required language construct.

Figure D.1. Creating a New File with LSE



In order to get template entries for a placeholder, you must move your cursor to the placeholder and expand it by using special LSE key sequences or **key bindings**. The most commonly used LSE commands and their default key bindings are:

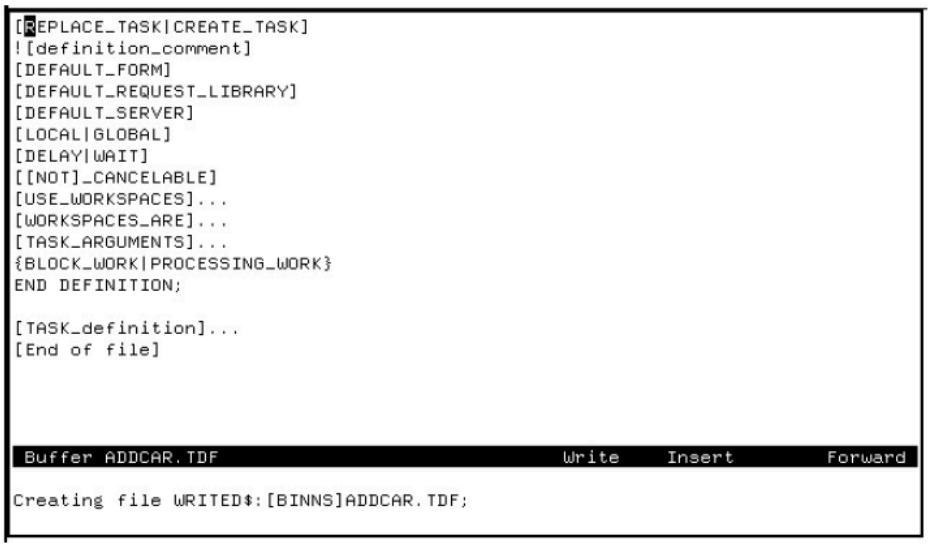
- **EXPAND** (**CTRL/E**)
- **GOTO PLACEHOLDER/FORWARD** (**CTRL/N**)
- **ERASE PLACEHOLDER** (**CTRL/K**)
- **GET LSE COMMAND LINE** (**CTRL/Z**)
- **HELP/INDICATED** (**PF1PF2**)

The **HELP/INDICATED** command summons ADU help for the token or placeholder on which you place your cursor. See *Section D.5, "Using HELP in LSE"* for a description of help within the LSE template.

Suppose that you want to duplicate the source definition file ADDCAR.TDF shown in *Example 2.2, "Contents of a Source Definition File"*.

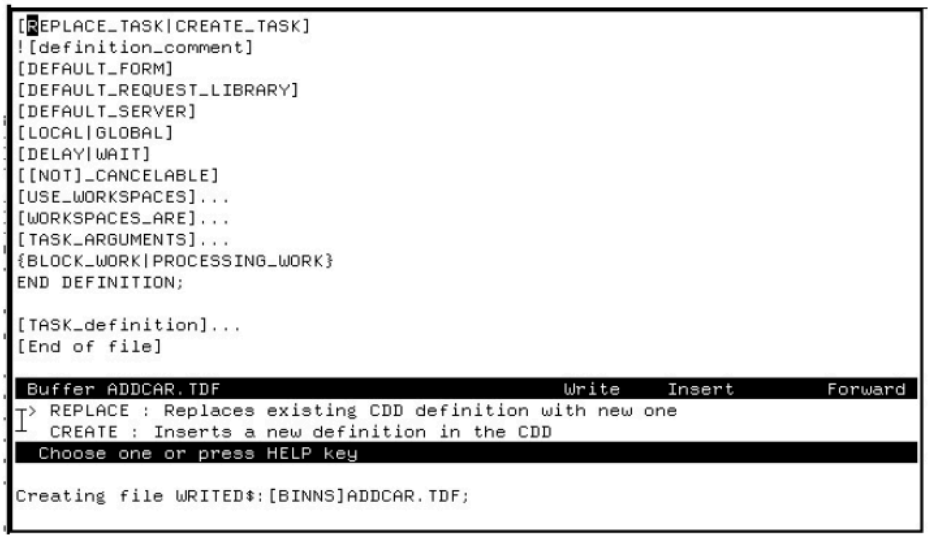
Use the **EXPAND** command to display in the editing buffer the clauses associated with the placeholder on which your cursor rests. *Figure D.2, "Expanding a Placeholder"* shows the result when you use the **EXPAND** command on the LSE phrase TASK_definition. You can also expand these clauses until you display the options necessary to create the source code for your particular task.

Figure D.2. Expanding a Placeholder



The cursor now rests on the [REPLACE_TASK|CREATE_TASK] placeholder. Use the **EXPAND** command again to list the additional options for that placeholder. *Figure D.3, "Expanding the REPLACE TASK|CREATE TASK Placeholder"* displays these options with the an arrow on the first choice.

Figure D.3. Expanding the REPLACE TASK|CREATE TASK Placeholder



To select an option from this menu, use the up and down arrow keys to position the pointer at one of the options and press **Return**. *Figure D.4, "Choosing REPLACE_TASK"* shows the display you receive when you select the option REPLACE_TASK.

Figure D.4. Choosing REPLACE_TASK

LSE inserts the clause REPLACE_TASK into the buffer and displays a new string of arguments associated with the REPLACE clause. When you expand the placeholder path name, LSE displays the following message:

```
Enter the CDD path name of the object in the CDD
```

In *Example 2.2, "Contents of a Source Definition File"*, the object name is ADD_CAR_RESERVATION_TASK. When you type this object name, LSE automatically removes the placeholder path name from the display.

Move the cursor to the `[replace_qualifiers]` placeholder with the command **`GOTO PLACEHOLDER/FORWARD`** **Ctrl/N**. Since the REPLACE TASK clause in *Example 2.2,*

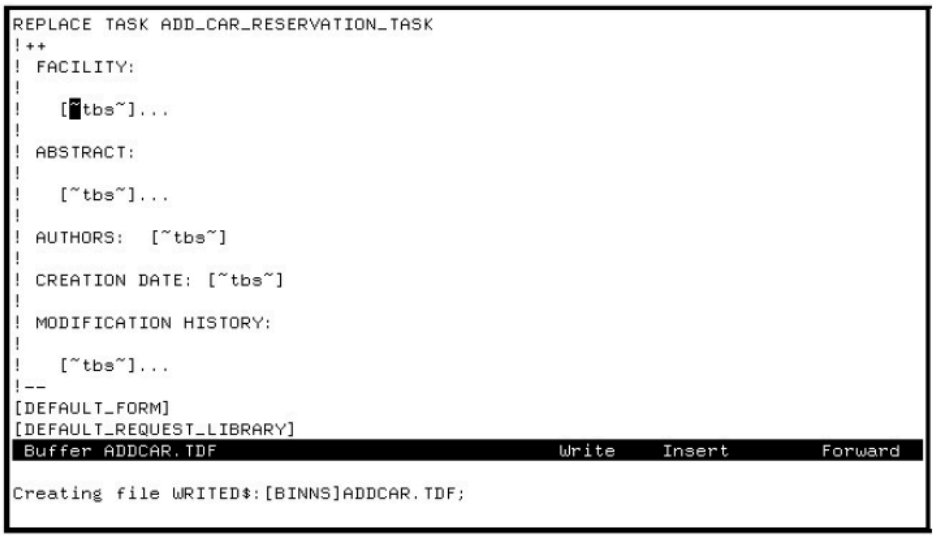
"Contents of a Source Definition File" does not contain any qualifiers, you can delete the placeholder with the **ERASE PLACEHOLDER** command **Ctrl/K**. When you erase a placeholder, LSE automatically positions the cursor at the next placeholder.

Note that when you attempt to delete a required placeholder (signified by curly braces), LSE returns the following message:

```
This is a required placeholder.  Continue erase operation [Y or N]?
```

You can also use LSE to enter comments. *Figure D.5, "Expanded Comment Placeholders"* demonstrates the expansion of the **![definition_comment]** placeholder.

Figure D.5. Expanded Comment Placeholders



The string **~tbs~** stands for *to be supplied* and indicates where you need to provide additional information.

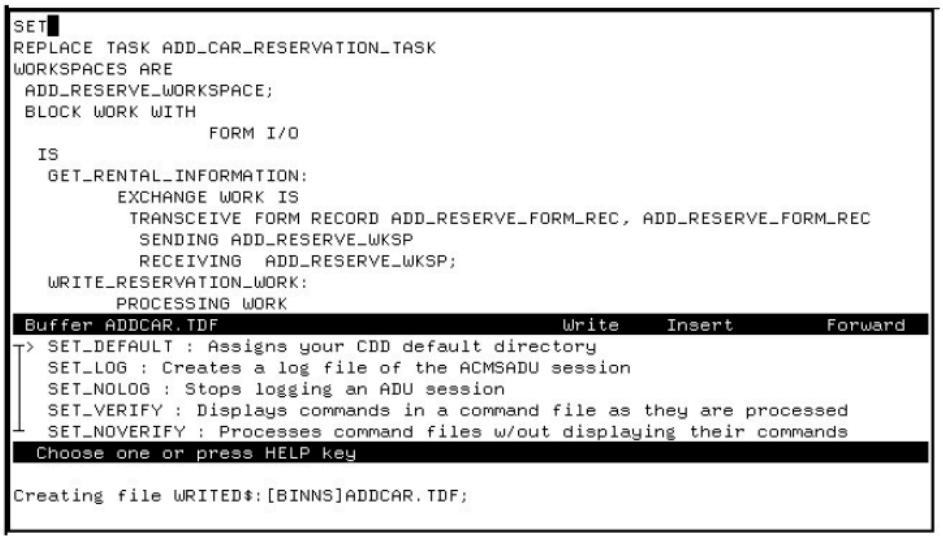
Example 2.2, "Contents of a Source Definition File" contains a line at the top of the file for which LSE did not give the user a prompt. That line is:

```
SET VERIFY
```

You can request that LSE prompt you for this line by expanding a token. A **token** is a keyword specific to ADU syntax. LSE does not provide tokens in the same way it provides placeholders. Instead, you must type a token directly into the buffer to add an ADU clause when there are no placeholders in the existing program. When you expand the token, LSE displays the available options in the same way it displays options for placeholders.

Figure D.6, "Expanding Tokens" demonstrates token expansion. The user has typed a portion of the token for the ADU clause SET VERIFY and expanded the token. LSE displays the option menu from which the user can select the SET_VERIFY clause.

Figure D.6. Expanding Tokens



You can also use tokens to bypass menus in cases where expanding a placeholder would result in a lengthy menu.

LSE also provides a COMMENT token that you can expand to include inline code. To use the COMMENT token, type the word **COMMENT** and expand the token. LSE inserts the following line at the position indicated by the cursor:

```
![inline_comment]
```

When you expand this new placeholder, LSE inserts the following lines into your program:

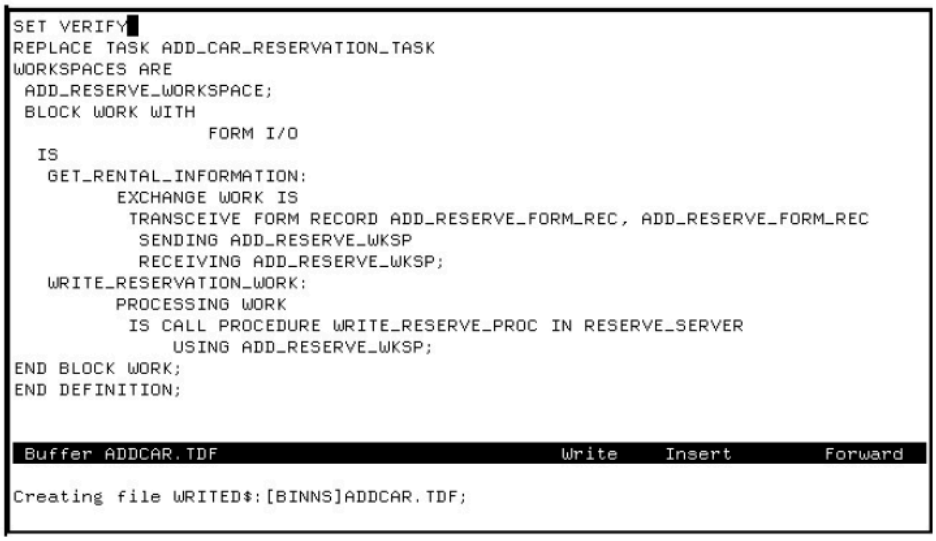
```
!++
![~tbs~]...
!--
```

LSE positions the cursor on the placeholder **~tbs~**, which stands for *to be supplied*. You can then enter comment text.

D.2.2. Creating the Final Source File

By using this process of expanding, selecting, moving to, and deleting placeholders and tokens and adding text where necessary, you can use LSE to create a source file that looks something like *Figure D.7, "Final Source File Created with LSE"*.

Figure D.7. Final Source File Created with LSE



D.2.2.1. Syntax Differences

The syntax generated by LSE may differ slightly from syntax that you use. For example, the source file produced by LSE contains expanded versions of the ADU clauses, such as PROCESSING WORK IS, where you may simply use PROCESSING. Both versions are correct. Whether or not you use the expanded syntax and different spacing depends on the programming conventions used at your site.

D.2.2.2. Exiting Editing Mode

Once you finish editing text, you can save the file and process it. To exit editing mode, press **Ctrl/Z**. LSE opens a command line preceded by the prompt LSE> at the bottom of the screen. At this command prompt you can:

- **EXIT** or **QUIT** the LSE session
- Access **HELP**
- Enter LSE and SCA commands
- Continue editing

For example, to continue editing, use the **CONTINUE** command:

```
LSE> CONTINUE
```

LSE then positions your cursor in the editing buffer.

To leave the LSE session and save your editing changes, use the **EXIT** command:

```
LSE> EXIT
```

LSE then writes the buffer to a file and exits to the DCL level. You can then process your file with ACMS.

D.3. Compiling Definitions with LSE COMPILE

ACMS lets you use the LSE **COMPILE** command from within an LSE editing session to create, modify, or replace a task, task group, application, or menu definition. To use the **COMPILE** command, you must

include the ADU **CREATE**, **REPLACE**, or **MODIFY** command in the definition, and enter **COMPILE** at the the LSE prompt.

```
LSE> COMPILE ADDCAR.TDF
```

LSE then displays the following message, indicating the start of the compilation:

```
Starting compilation: @SYS$LIBRARY:ACMS$LSE_COMPILE.COM ADDCAR.TDF
```

When the compilation is finished, LSE displays the following message:

```
Compilation of buffer ADDCAR.TDF completed
```

The ADU language templates define the LSE COMPILE string as "@SYS$LIBRARY:ACMS$LSE_COMPILE.COM". LSE passes the definition filename as a parameter to the ACMS$LSE_COMPILE.COM command procedure. The command procedure then enters ADU and invokes the definition file, using the @ command.

If ADU finds an error in the **CREATE**, **REPLACE**, or **MODIFY** command, ACMS does not create the LSE diagnostics file, and LSE does not display any messages referring to the error.

D.4. Examining Diagnostic Messages with LSE REVIEW

The **REVIEW** command issued within an LSE session enables you to display a diagnostics file at the same time you are examining the source code that generated the diagnostics file. ACMS creates the diagnostics file when you attempt to create, modify, or replace a definition with the **/DIAGNOSTICS** qualifier.

When you process the definition, ACMS creates a separate diagnostics file using the entity name and a .DIA extension. You can then use LSE to review the source code and the diagnostics file with a split screen.

D.4.1. Generating the Diagnostics File

Suppose you have created a definition file called ADDCAR.TDF and want to generate a diagnostics file to aid you in fixing coding errors. The **REPLACE** command in the definition file contains the task name with **/DIAGNOSTICS** qualifier required to produce the diagnostics file:

```
REPLACE TASK ADD_CAR_RESERVATION_TASK/DIAGNOSTICS
```

When you compile the definition file ADDCAR.TDF in ADU, ACMS produces a diagnostic file called ADD_CAR_RESERVATION_TASK.DIA.

ACMS names the diagnostic file after the entity, not after the source file. For example, if your source file is CLOCKS.GDF and you have defined a task group called GRANDFATHER with the **/DIAGNOSTICS** qualifier in the file, ACMS names the resulting diagnostics file GRANDFATHER.DIA.

If you want the name of the diagnostic file to be different than the entity name, specify the filename with the **/DIAGNOSTICS** qualifier. For example:

```
REPLACE GROUP CLOCKS/DIAGNOSTICS=CLOCKS.DIA
```

D.4.2. Examining the Diagnostics File with LSE REVIEW

To examine the diagnostics file, invoke LSEDIT and supply the name of the source file:

```
$ LSEDIT ADDCAR.TDF
```

Obtain the LSE> prompt by pressing **Ctrl/Z**. Then issue the **REVIEW** command, supplying the name of the diagnostics file:

```
LSE> REVIEW/FILE=ADD_CAR_RESERVATION_TASK.DIA
```

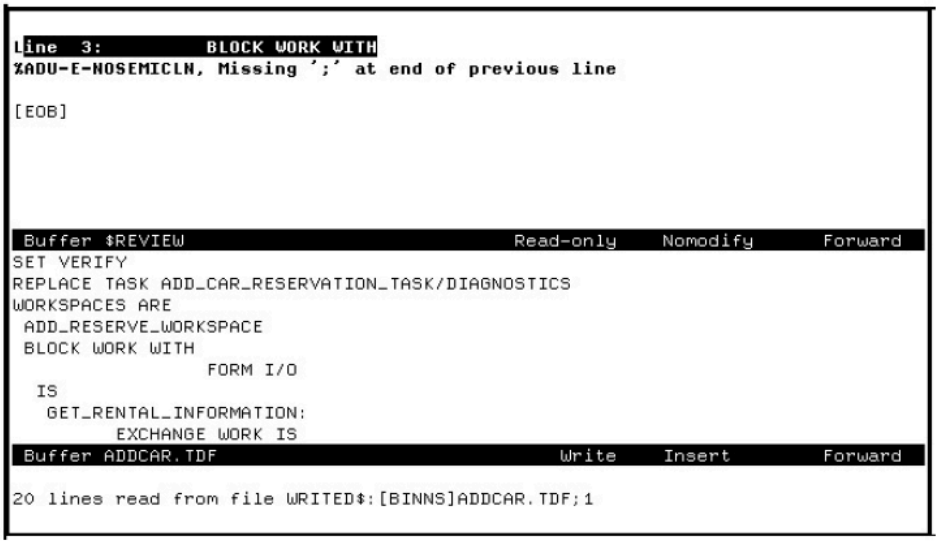
If your compilation was successful and the diagnostic file contains no errors, LSE displays the following message:

```
Compilation produced no errors
```

Because the compilation was error free, LSE does not display the .DIA file. If you make an error in the ADU command line of the task definition, ADU does not compile the task definition, and a success status is returned.

If the compilation was unsuccessful and the diagnostic file contains error messages, the screen displays two buffers – one at the top half of the screen and one at the bottom half – as shown in *Figure D.8, "Examining Diagnostic Files with LSE REVIEW"*.

Figure D.8. Examining Diagnostic Files with LSE REVIEW



The window at the top of the screen contains the $REVIEW buffer that displays diagnostic messages. The window at the bottom of the screen contains the source file text. By moving between these two windows, you can review the compilation errors and correct the corresponding code.

Table D.1, "LSE REVIEW Window Commands" lists some commands you can use at the LSE> prompt to move between the $REVIEW buffer and the source code buffer.

Table D.1. LSE REVIEW Window Commands

Command	Default Key Binding	Description
NEXT ERROR	**Ctrl/N**	Highlights next error message in the $REVIEW buffer

Command	Default Key Binding	Description
GOTO SOURCE	**Ctrl/G**	Highlights source line corresponding with error message currently highlighted in the $REVIEW buffer
CONTINUE	**Ctrl/Z**	When used after **GOTO SOURCE**, places cursor in source code buffer for editing. To return to REVIEW mode, press **Ctrl/Z**.
PREVIOUS ERROR	**Ctrl/B**	Highlights previous error line in $REVIEW buffer
END REVIEW	None	Terminates review. Removes $REVIEW buffer from screen and displays source code buffer only.

When you start the REVIEW session, LSE highlights the first error message in the $REVIEW buffer. To find the source code corresponding with this error message, use the **GOTO SOURCE** command:

```
LSE> GOTO SOURCE
```

In the lower part of the screen, LSE then displays and highlights the line of source code that generated the error. To edit the source code, use the **CONTINUE** command:

```
LSE> CONTINUE
```

You can now enter your changes in the source code buffer. Once you have made your changes, you can return to the LSE command by typing **Ctrl/Z**.

When you have completed your review of the diagnostics file, you can terminate the review session by entering the **END REVIEW** command at the LSE> prompt:

```
LSE> END REVIEW
```

LSE then removes the diagnostics file from your screen and displays the source file.

When you use the **REVIEW** command, you must call the correct version of the diagnostics file generated by the source file you are currently editing. For example, suppose you compiled ADDCAR.TDF;4 to generate the diagnostics file ADD_CAR_RESERVATION_TASK.DIA;4. If you create another version called ADDCAR.TDF;5 but do not recompile it to generate a matching .DIA file, you will receive the following message when you attempt to use the **GOTO SOURCE** command during LSE REVIEW mode:

```
File in buffer is not the version compiled
```

When you receive this message, you can perform either of the following actions to resolve the problem:

- Enter the LSE **COMPILE** command to recompile the source file to generate a new diagnostics file. Then, enter the **REVIEW** command. When you enter the **REVIEW** command and the file specification, LSE reads the most recent version of the .DIA file into the buffer.
- End the LSE session. Invoke LSE again and specify the source file name and the version number. For example, although ADDCAR.TDF;5 is the most recent version, you want to debug ADDCAR.TDF;4 with its corresponding diagnostics file, ADDCAR.DIA;4. Therefore, at the DCL level, specify the file name and version number:

  ```
  LSE> ADDCAR.TDF;4
  ```

  Then invoke **REVIEW** as you normally would. To ensure that you are requesting the correct version of the .DIA file, you can specify the version number:

```
$ LSE> REVIEW/FILE=ADDCAR.DIA;4
```

If you try to use **REVIEW** with a diagnostics file that was not generated from the source file you are displaying, LSE **REVIEW** attempts to match errors in the diagnostics file with lines in the source file. To avoid such confusion, always delete outmoded .DIA files.

D.5. Using HELP in LSE

LSE includes a help system for navigating and editing within the buffer being edited. Press **Help** within an LSE editing buffer to get a description of the LSE editing commands and key definitions.

LSE also provides access to context-senstive ADU help for ACMS. Within the LSE editing sesion, place your cursor on any placeholder or token and use the **HELP/INDICATED** command (**PF1 PF2**). LSE summons an appropriate ADU help message from the ADU help message file. Just as when you type a series of help topics to reach a particular help message, the help message indicates the topic titles that led to the message for the placeholder or token for which you summoned help. For example, the help message for the placeholder GROUP in a task definition file is:

```
TASK

  WORKSPACES

    Keywords

      GROUP

         Identifies the workspace as a GROUP type workspace.  The
         contents of a GROUP workspace can be used by many instances
         of the same or different tasks.  ACMS maintains these contents
         from application startup to application shutdown.
```

You cannot move back up the topic path within the LSE editing session. However, if there are additionial subtopics for the current topic, you can reach them as you would for subtopics through the ADU **HELP** command.

If you type a portion of a token and press **PF1 PF2**, you get a message for each ADU keyword that could be expanded from the portion you typed. The messages correspond to the choices on the option menu that appears if you expand the partial token. For example, if you type the partial token SET as shown in *Figure D.6, "Expanding Tokens"*, and then press **PF1 PF2**, you get help for each ADU keyword listed in the options menu.

For more comprehensive information on other LSE features, consult *VSI DECset for OpenVMS Language-Sensitive Editor/Source Code Analyzer Reference Manual* [https://docs.vmssoftware.com/vsi-decset-for-openvms-language-sensitive-editor-source-code-analyzer-reference-manual/].

Appendix E. Request Interface Kit Components

When you install the Request Interface, the installation procedure creates the system logical ACMS$RI_EXAMPLES, which is a subdirectory of the ACMS$EXAMPLES directory. It also creates a new CDD dictionary, ACMS$RI_CDD, under the ACMS$DIR dictionary.

This appendix lists the files included with the Request Interface software. *Table E.1, "Request Interface Kit Components"* gives the name and location of the components that are on the kit.

Table E.1. Request Interface Kit Components

Component	Name	Location
ACMS Request Interface agent source modules	ACMS$RI_AGENT.B32	ACMS$RI_EXAMPLES
ACMS Request Interface AGENT and DEBUG object module library	ACMS$RI.OLB	SYS$LIBRARY
ACMS-supplied RI agent image (ACMS$RI_AGENT)	ACMS$RI_AGENT.EXE	SYS$SYSTEM
ACMS Request Interface MENU (FMS menu) source modules used by ACMS$RI_AGENT	ACMS$RI_FMS_MENU.BAS	ACMS$RI_EXAMPLES
ACMS Request Interface MENU object module library	ACMS$RI_FMS_MENU.OLB	SYS$LIBRARY
ACMS Request Interface menu form definitions for the menu interface of ACMS$RI_AGENT	MENU_MAIN_FORM.FRM	ACMS$RI_EXAMPLES
ACMS Request Interface FMS menu form	MENU_HELP_FORM.FRM	ACMS$RI_EXAMPLES
ACMS Request Interface FMS menu form library	FMS_MENU_LIBRARY.FLB	ACMS$RI_EXAMPLES

For more information on these components, continue on to the next sections of this appendix.

E.1. Application Independent Modules

The application independent modules are as follows:

- ACMS$RI_AGENT.B32

  The ACMS$RI_AGENT is an example of how an agent program can be developed to utilize the RI interface. The ACMS$RI_AGENT uses the Systems Interface (SI) to sign users in to ACMS and call tasks in an ACMS application, just like any other ACMS agent.

  The ACMS$RI_AGENT allows a choice of two different user interfaces to enter the task and application selections. The first interface is a user-written menu interface. This type of interface requires that a programmer write two special purpose procedures: an initialization procedure names

ACMS$RI_MENU_INIT, and a menu interface procedure names ACMS$RI_MENU_ROUTINE. These two menu routines can be included into the ACMS$RI_AGENT code in one of two ways:

- The ACMS$RI_MENU_INIT procedure and the ACMS$RI_MENU_ROUTINE procedure can be linked into a shared image and then, at run time, dynamically activated. The shareable image is activated into the RI agent process, using the LIB$FIND_IMAGE_SYMBOL RTL routine. In order to activate these routines in the ACMS$RI_AGENT, the user must define the logical ACMS$RI_MENU that points to the shared image file that contains the two procedures.

- The ACMS$RI_MENU_INIT procedure and the ACMS$RI_MENU_ROUTINE procedure can be linked directly into the RI agent code and called as required.

The second interface is the default interface. If the user chooses not to write a menu interface, ACMS$RI_AGENT prompts the user for the task and application desired.

With either interface, the ACMS$RI_AGENT signs the user in to the ACMS system. The agent then calls the ACMS$INIT_EXCHANGE_IO service and specifies the ACMS$M_IO_ENABLE_SYNC_RI flag. This flag indicates to the ACMS system, EXC specifically, that all task I/O will be executed in the agent process synchronously.

- ACMS$RI_DEBUG.B32

  This is the source code for the Request Interface (RI) supplied debugger module. The debugger module signals with an SS$_DEBUG status, which acts like pressing **Ctrl/Y** and then typing DEBUG at the DCL prompt. If you include this module in the Request Interface request library shared image file, the first time this library file is accessed the OpenVMS debugger prompt (DBG>) appears. This allows you to set breakpoints at User Request Procedures (URPs) so you can debug the request procedure code.

- ACMS Request Interface AGENT and DEBUG object module library, ACMS$RI.OLB

  This is the object library that contains the RI agent object and the RI debugger module object. It contains the following modules:

  - ACMS$RI_AGENT

    RI agent object module is provided so you are not required to write an agent to make use of RI facility. In addition, users may want to code their own menu interfaces and link them directly into the RI agent code.

  - ACMS$RI_DEBUG

    Debug object module which, if included in the RI request library shared image file, allows users to test and debug their user request procedures (URPs) using the OpenVMS debugger. This must be linked into the shared image file.

- ACMS Request Interface Agent executable image(s), ACMS$RI_AGENT.EXE

  This is the supplied RI agent executable image, which can be used to activate the menu interface shared image or to debug URPs. By default, this agent prompts you for task name and application name. In addition, if the default prompt mode is selected, it displays the error (or success) message of a selected task.

- ACMS Request Interface MENU (FMS menu) source modules, ACMS$RI_FMS_MENU.BAS

This is the source code for the supplied FMS menu interface. It contains the two special-purpose procedures, an initialization procedure called ACMS$RI_MENU_INIT and a user interface (menu) procedure named ACMS$RI_MENU_ROUTINE.

- ACMS$RI_MENU_INIT

  This module is BASIC FUNCTION source code for the initialization procedure for the FMS Menu interface. This procedure creates the FMS workspaces, attaches the terminal, and opens the FMS Menu form library pointed to by the logical ACMS$RI_FMS_MENU_LIB.

- ACMS$RI_MENU_ROUTINE

  This module is BASIC FUNCTION source code for the FMS menu interface. This procedure displays a menu and prompts the user for a selection. The FMS menu interface duplicates some of the ACMS menu functionality (*, -, $EXIT, $HELP, keyword, number). This procedure uses the named data functionality of FMS. To adapt this FMS interface to your specific requirements, you need to change only the supplied FMS forms; you do not need to change the FMS interface procedures.

- ACMS Request Interface MENU object module library, ACMS$RI_FMS_MENU.OLB

  This is the Request Interface FMS menu interface object module library. It contains the ACMS$RI_MENU_INIT and the ACMS$RI_MENU_ROUTINE object modules. With it, you can link the FMS menu interface directly into the RI agent code.

- ACMS Request Interface FMS menu form definitions:

  - MENU_MAIN_FORM.FRM

    This is the FMS form used as the top-level (main) menu form. It is similar to the layout of the ACMS default menu format, which includes a number, a keyword, task or menu indicator (T or M) and some descriptive text. At the bottom of the menu is the selection prompt. The menu layout is static text in the FMS form. The only field on the menu form is the selection field. Numbers and keywords are used as named data indexes and named data keywords. To change the form for a specific application, modify the static text on the main menu, and change the named data field information. The default main menu is set up for the FMS sample application.

  - MENU_HELP_FORM.FRM

    This is the FMS form used when the user types $HELP or presses **PF2** twice at the selection prompt. It displays a form that explains what functionality is available under the FMS menu interface such as *, -, $EXIT, number, keyword, and so on. Include this form in all FMS menu interface form libraries.

- ACMS Request Interface FMS menu form library, FMS_MENU_LIBRARY.FLB

  This is the FMS form library that contains the MENU_MAIN_FORM and the MENU_HELP_FORM. The name of the form library is not important since the FMS menu interface uses a logical (ACMS$RI_FMS_MENU_LIB) to point to the proper FMS form library.

E.2. RI Sample Application

This section describes the components of the ACMS Request Interface SMG and QIO sample application source modules. In general, the RI sample application uses SMG and QIO user request

procedures (URPs) to do terminal I/O. There are two task group definitions: one uses .RLB request libraries, and the other uses an EXE shareable image file in the REQUEST LIBRARY IS clause. All these components are located in the ACMS$RI_EXAMPLES directory.

E.2.1. RI Sample Application Source Modules

This section lists the sample applicaton source modules for the RI sample application.

- RI_EMPLOYEE_RECORD.CDO record definition

  This is the CDD record definition for the ACMS Request Interface (RI) SMG and QIO sample application. The record contains three fields: the EMPLOYEE ID NUMBER, EMPLOYEE FIRST NAME, and EMPLOYEE LAST NAME.

- RI_APPL_FORMS.BAK – CDD backup of FMS form definitions

  This is the CDD backup of the FMS forms used in the SMG and QIO sample application. It contains the following forms:

  - RI_ADD_FORM

    TDMS form used in the add request that is used in the add task example.

  - RI_INQ_FORM

    TDMS form used in the inquire request that is used in the inquiry task.

  - RI_INQ_DISP_FORM

    TDMS form used in the inquire display request that is used in the inquiry task.

- FMS request definitions

  - RI_ADD_REQUEST.RDF

    This is the TDMS request definition used in the RI_ADD_TASK task in the SMG and QIO sample application. It prompts the user to enter employee ID, employee first name, and employee last name.

  - RI_INQ_REQUEST.RDF

    This is the TDMS request definition used in the RI_INQ_TASK task in the SMG and QIO sample application. It prompts the user to enter employee ID.

  - RI_INQ_DISP_REQUEST.RDF

    This is the TDMS request definition used in the RI_INQ_TASK task in the SMG and QIO sample application. It displays the employee ID, employee first name, and employee last name.

- TDMS request library definition (includes **BUILD** command), RI_REQUEST_LIBRARY.LDF

  This is the request library definition for the two TDMS request libraries. It also includes the **BUILD** command to build the request libraries RI_REQUEST_LIBRARY1.RLB and RI_REQUEST_LIBRARY2.RLB.

- Task definitions

- RI_ADD_TASK.TDF

  This is the ACMS Task Definition for the add task in the sample application. The task has one exchange step and one processing step. The exchange step prompts the user to enter employee ID, employee first name, and employee last name and then the processing step writes the data out to an RMS datafile using employee ID as the key. The task contains no control action or program request keys.

- RI_INQ_TASK.TDF

  This is the ACMS Task Definition for the inquire task in the sample application. The task has two exchange steps and one processing step. The first exchange step prompts the user to enter employee ID. The processing step retrieves the employee information from an RMS datafile using employee ID as the key. Then the second exchange step displays the employee information retrieved from the previous processing step. The task contains no control action or program request keys.

- Task group definition (includes **BUILD** command)

  - RI_LOGICAL_GROUP.GDF

    This is the ACMS task group definition for the RI SMG and QIO sample application. This group does not define a request library shared image (.EXE) file in the REQUEST LIBRARY IS clause (only TDMS request library files (RLB)). If you build the ACMS application using this task group definition, you must define a logical name (ACMS$RI_LIB_libraryname) to use the SMG or QIO user request procedures for this group. By default, the sample SMG/QIO application was built using the logical name task group definition. In order to make use of the SMG or QIO procedures, you must define the ACMS$RI_LIB_libraryname logical. The definition also includes the **BUILD** command to build the task group (.TDB).

  - RI_PASSED_GROUP.GDF

    This is the ACMS task group definition for the RI SMG and QIO sample application. This task group defines a request library shared image (.EXE) file in the REQUEST LIBRARY IS clause. If you build the ACMS application using this task group definition, you do not need a logical to use the SMG or QIO user request procedures for this group. This definition also includes the **BUILD** command to build the task group (.TDB).

- Procedure server source modules

  - RI_INIT_PROCEDURE.COB

    This is the COBOL source code for the initialization routine for the SMG/QIO sample application server procedure. The routine opens an RMS datafile called RI_SIMPLE_APPL.DAT.

  - RI_ADD_PROCEDURE.COB

    This is the COBOL source code for the add routine for the SMG/QIO sample application server procedure. This procedure is used in the RI_ADD_TASK to store a record containing employee ID, employee first name, and employee last name.

  - RI_GET_PROCEDURE.COB

This is the COBOL source code for the get routine for the SMG/QIO sample application server procedure. This procedure is used in the RI_INQ_TASK to retrieve a record containing employee ID, employee first name, and employee last name.

- RI_TERM_PROCEDURE.COB

  This is the COBOL source code for the termination routine for the SMG/QIO sample application server procedure. The routine closes the RMS datafile called RI_SIMPLE_APPL.DAT.

- Command procedures to COMPILE and LINK the procedure server
  - RI_SERVER_COMPILE.COM

    This is a command procedure to recompile all the procedure server modules. It also creates a new object library (RI_SERVER.OLB) and inserts all the server modules.

  - RI_SERVER_LINK.COM

    This is a command procedure to relink the procedure server modules into SMG/QIO sample application server image. It uses the server object library created in the recompile command procedure.

- Application definition (includes **BUILD** command), RI_SAMPLE_APPL.ADF

  This is the ACMS application definition for the RI SMG and QIO sample application. It uses the RI_LOGICAL_GROUP.TDB task group. The definition includes the **BUILD** command to build the application database (.ADB).

- Menu definition (includes **BUILD** command), RI_APPL_MENU.MDB

  This is the ACMS menu definition for the RI SMG and QIO sample application. It includes the **BUILD** command to build the menu database (.MDB).

- Request Interface user request procedure (URP) source modules
  - RI_FORTRAN_QIO.FOR

    This is the FORTRAN source code for the RI request library file that contains all the user request procedures (URPs) for the RI QIO sample application. The URPs use the OpenVMS QIO system service to read and write to the terminal. This file contains the following URP source modules:

    - ACMS$RI_LIB_INIT

      This is the FORTRAN FUNCTION source code for the initialization procedure for the RI request library shared image file. This assigns a channel to the current terminal using the OpenVMS SYS$ASSIGN system service.

    - ACMS$RI_LIB_CANCEL

      This is the FORTRAN FUNCTION source code for the cancellation procedure for the RI request library shared image file. This cancels all outstanding terminal I/O, using the OpenVMS SYS$CANCEL system service, and signals an ACMS cancel error.

    - RI_ADD_REQUEST

This is the FORTRAN FUNCTION source code to duplicate the TDMS add request by using the OpenVMS SYS$QIOW system service to prompt the user for employee ID, employee first name, and employee last name. The employee record workspace (RI_EMPLOYEE_RECORD) is passed to it as a parameter.

- RI_INQ_REQUEST

  This is the FORTRAN FUNCTION source code to duplicate the TDMS inquire request by using the OpenVMS SYS$QIOW system service to prompt the user for employee ID number. The employee record (RI_EMPLOYEE_RECORD) workspace is passed to it as a parameter.

- RI_INQ_DISP_REQUEST

  This is the FORTRAN FUNCTION source code to duplicate the TDMS inquire display request by using the OpenVMS SYS$QIOW system service to display the employee first name and the employee last name. The employee record workspace (RI_EMPLOYEE_RECORD) is passed to it as a parameter.

- RI_BASIC_SMG.BAS

  This is the BASIC source code for RI request library file that contains all the user request procedures (URPs) for the RI SMG sample application. The URPs use the OpenVMS SMG Run-Time Library (RTL) service to read and write to the terminal. This file contains the following URP source modules (note that this file does not contain an optional cancel URP):

  - ACMS$RI_LIB_INIT

    This is the BASIC FUNCTION source code for the initialization procedure for the RI request library shared image file. This creates a pasteboard and a keyboard for the terminal using the OpenVMS SMG RTL services.

  - RI_ADD_REQUEST

    This is the BASIC FUNCTION source code to duplicate the TDMS add request by using the OpenVMS SMG RTL services to prompt the user for employee ID, employee first name, and employee last name. The employee record workspace (RI_EMPLOYEE_RECORD) is passed to it as a parameter.

  - RI_INQ_REQUEST

    This is the BASIC FUNCTION source code to duplicate the TDMS inquire request by using the OpenVMS SMG RTL services to prompt the user for employee ID number. The employee record workspace (RI_EMPLOYEE_RECORD) is passed to it as a parameter.

  - RI_INQ_DISP_REQUEST

    This is the BASIC FUNCTION source code to duplicate the TDMS inquire display request by using the OpenVMS SMG RTL services to display the employee first name and the employee last name. The employee record workspace (RI_EMPLOYEE_RECORD) is passed to it as a parameter.

- Command procedures to COMPILE and LINK the request library shared image file:

- RI_REQ_LIB_COMPILE.COM

  This is a command procedure to recompile both the user request procedure modules for the SMG and the QIO request library shared image file. It also creates two new object libraries (RI_FORTRAN_QIO.OLB and RI_BASIC_SMG.OLB) and inserts all URP modules in the appropriate object library.

- RI_REQ_LIB_LINK.COM

  This is a command procedure to relink the user request procedure modules into the request library shared image file used in the RI_SAMPLE_APPLICATION. It uses the object library created in the recompile command procedure.

E.2.2. RI Sample Application Run-Time Files

This section lists the run-time files for the RI sample application.

- TDMS request library file (RLB)

  - RI_REQUEST_LIBRARY1.RLB

    This is the TDMS request library for the RI SMG/QIO sample application. It contains the RI_ADD_REQUEST request and the RI_INQ_REQUEST request. The reason for splitting the TDMS request between two TDMS request libraries is to demonstrate the use of a URP and a TDMS request in a single task (RI_INQ_TASK).

  - RI_REQUEST_LIBRARY2.RLB

    This is the TDMS request library for the RI SMG/QIO sample application. It contains the RI_INQ_DISP_REQUEST request.

- Task group database (TDB)

  - RI_LOGICAL_GROUP.TDB

    This is the ACMS task group database for the RI SMG and QIO sample application. This group does not define a request library shared image (.EXE) file in the REQUEST LIBRARY IS clause (only TDMS request library files (RLB)). If you build the ACMS application using this task group database, you must define a logical name (ACMS$RI_LIB_libraryname) to use the SMG or QIO user request procedures for this group. By default, the sample SMG/QIO application was built using the logical name task group database. To make use of the SMG or QIO procedures, you must define the ACMS$RI_LIB_libraryname logical.

  - RI_PASSED_GROUP.TDB

    This is the ACMS task group database for the RI SMG and QIO sample application. This task group defines a request library shared image (.EXE) file in the REQUEST LIBRARY IS clause. If you build the ACMS application using this task group database, no logical is needed to use the SMG or QIO user request procedures for this group.

- RI_SERVER.OLB – Procedure server object module library

  This is the object library containing the four object modules that make up the RI server used in the SMG/QIO sample application. It includes the following four modules:

- RI_INIT_PROCEDURE – Object module for the initialization procedure
- RI_TERM_PROCEDURE – Object module for the termination procedure
- RI_ADD_PROCEDURE – Object module for the add record procedure
- RI_GET_PROCEDURE – Object module for the get record procedure

- RI_SERVER.EXE – Procedure Server image file

  This is the procedure server image for the RI SMG/QIO sample application.

- RI_SAMPLE_APPL.ADB – Application Database (ADB)

  This is the ACMS application database for the RI sample application. It uses the RI_LOGICAL_GROUP.TDB task group.

- RI_APPL_MENU.MDB – Menu Database (MDB)

  This is the ACMS menu database for the RI sample application.

- Request Interface User Request Procedure object module library
  - RI_BASIC_SMG.OLB

    This is the object library that contains the object modules for the SMG URPs contained in the RI request library shared image file. It contains the following object modules:

    - ACMS$RI_LIB_INIT

      This is the object module for the RI request library shared image file initialization user request procedure.

    - RI_ADD_REQUEST

      This is the object module for the RI request library shared image file add request user request procedure.

    - RI_INQ_REQUEST

      This is the object module for the RI request library shared image file inquiry request user request procedure.

    - RI_INQ_DISP_REQUEST

      This is the object module for the RI request library shared image file inquiry display user request procedure.

  - RI_FORTRAN_QIO.OLB

    This is the object library which contains the object modules for the QIO URPs contained in the RI request library shared image file. It contains the following object modules:

    - ACMS$RI_LIB_INIT

This is the object module for the RI request library shared image file initialization user request procedure.

- ACMS$RI_LIB_CANCEL

  This is the object module for the RI request library shared image file cancellation user request procedure.

- RI_ADD_REQUEST

  This is the object module for the RI request library shared image file add request user request procedure.

- RI_INQ_REQUEST

  This is the object module for the RI request library shared image file inquiry request user request procedure.

- RI_INQ_DISP_REQUEST

  This is the object module for the RI request library shared image file inquiry display user request procedure.

- Request Interface request library shared image file

  - RI_BASIC_SMG.EXE

    This is the SMG RI request library shared image file that includes all SMG URP modules.

  - RI_FORTRAN_QIO.EXE

    This is the QIO RI request library shared image file that includes all QIO URP modules.

E.3. FMS Sample Application

This section describes the components of the FMS sample application. In general, the FMS sample application uses FMS user request procedures written in FORTRAN to replace the TDMS requests included in the request libraries that are defined in the task group definition. This requires the use of the ACMS$RI_LIB_libraryname logical. All the components listed are in the ACMS$RI_EXAMPLES directory.

E.3.1. FMS Sample Application Source Modules

This section lists the source modules for the FMS sample application.

- Record definitions

  - FMS_EMPLOYEE_RECORD.DDL

    This is the CDD record definition for the ACMS Request Interface (RI) FMS sample application. The record contains employee information.

  - FMS_WORKSPACE_RECORD.CDO

This is the CDD record definition for the ACMS RI FMS sample application. The record contains two fields: the program request key field and the error status field.

- FMS_SCROLL_RECORD.CDO

  This is the CDD record definition for the ACMS RI FMS sample application. The record contains scroll area information.

- FMS_SALARY_RECORD.CDO

  This is the CDD record definition for the ACMS RI FMS sample application. The record contains the salary range information used by the FMS user action routines to demonstrate cross-field validation.

- FMS_FORM.BAK – CDD backup of TDMS form definitions

  This is the CDD backup of the TDMS forms used in the FMS sample application. It contains the following forms:

  - FMS_ADD_FORM

    TDMS form used in the add request that is used in the add task example.

  - FMS_INQ_FORM

    TDMS form used in the inquiry request and the inquiry display request that is used in the inquiry task.

  - FMS_UPD_FORM

    TDMS form used in the inquiry update request and the display/modify request that is used in the update task.

  - FMS_SCROLL_FORM

    TDMS form used in the inquiry scroll request that is used in the scroll task.

  - FMS_SCROLL_DISP_FORM

    TDMS form used in the display scroll request that is used in the scroll task.

- TDMS request definitions

  - FMS_ADD_REQUEST.RDF

    This is the TDMS request definition used in the FMS_ADD_TASK task in the FMS_SAMPLE application. It prompts the user to enter all employee information.

  - FMS_INQ_REQUEST.RDF

    This is the TDMS request definition used in the FMS_INQ_TASK task in the FMS_SAMPLE application. It prompts the user to enter the employee badge number.

  - FMS_INQ_DISP_REQUEST.RDF

This is the TDMS request definition used in the FMS_INQ_TASK task in the FMS_SAMPLE application. It displays employee information that was requested in the inquire request.

- FMS_UPD_REQUEST.RDF

  This is the TDMS request definition used in the FMS_UPD_TASK task in the FMS_SAMPLE application. It prompts the user to enter the employee badge number. This is the same as the FMS_INQ_REQUEST.

- FMS_UPD_DISP_REQUEST.RDF

  This is the TDMS request definition used in the FMS_UPD_TASK task in the FMS_SAMPLE application. It displays the employee information requested in the update request; it then allows users to modify the employee information.

- FMS_SCROLL_REQUEST.RDF

  This is the TDMS request definition used in the FMS_SCROLL_TASK task in the FMS_SAMPLE application. It prompts the user to enter an employee last name.

- FMS_SCROLL_DISP_REQUEST.RDF

  This is the TDMS request definition used in the FMS_SCROLL_TASK task in the FMS_SAMPLE application. It displays employee last name, employee first name, and employee badge number for all employees with the specified last name from the FMS_SCROLL_REQUEST.

- TDMS request library definition (includes **BUILD** command), FMS_REQUEST_LIBRARY.LDF

  This is the request library definition for the TDMS request libraries. This also includes the **BUILD** command to build the request library FMS_REQUEST_LIBRARY.RLB.

- Task definitions

  - FMS_ADD_TASK.TDF

    This is the ACMS task definition for the add task in the FMS_SAMPLE application. The task has one exchange step and one processing step. The exchange step prompts the user to enter the employee information, and then the processing step writes the data out to an RMS datafile using employee ID as the key. The task contains control action for duplicate records and a program request key for **PF1/E**.

  - FMS_INQ_TASK.TDF

    This is the ACMS task definition for the inquire task in the FMS_SAMPLE application. The task has two exchange steps and one processing step. The first exchange step prompts the user to enter employee ID. The processing step retrieves the employee information from an RMS datafile using employee ID as the key; then the second exchange step displays the employee information retrieved from the previous processing step. The task contains control action for record not found and a program request key for **PF1/E**.

  - FMS_UPD_TASK.TDF

    This is the ACMS Task Definition for the update task in the FMS_SAMPLE application. The task has two exchange steps and two processing step. The first exchange step prompts the user to

enter employee ID. The processing step retrieves the employee information from an RMS data file using employee ID as the key. The second exchange step displays the employee information retrieved from the previous processing step and lets user modify the employee data. Then the second processing step rewrites the data record to the file. The task contains control action for record not found and a program request key for **PF1/E**.

- FMS_SCROLL_TASK.TDF

  This is the ACMS task definition for the scroll task in the FMS_SAMPLE application. The task has two exchange steps and one processing steps. The first exchange step prompts the user to enter the last name. The processing step retrieves employee information on all employees with that last name from an RMS datafile, using employee last name as the key. Then the second exchange step displays the employee information retrieved from the previous processing step in a scroll region. The task contains control action for record not found and a program request key for **PF1/E**.

- Task group definition (includes **`BUILD`** command), FMS_GROUP.GDF

  This is the ACMS task group definition for the RI FMS sample application. This group does not define a shared image (.EXE) file in the REQUEST LIBRARY IS clause. You must define a logical name to use the RI for this group. This also includes the **`BUILD`** command to build the task group database FMS_GROUP.TDB.

- Procedure server source modules

  - FMS_INIT_PROCEDURE.COB

    This is the COBOL source code for the initialization routine for the FMS sample application server procedure. The routine opens an RMS datafile called FMS_APPL.DAT.

  - FMS_ADD_PROCEDURE.COB

    This is the COBOL source code for the add routine for the FMS sample application server procedure. This procedure is used in the FMS_ADD_TASK to store a record containing the employee information.

  - FMS_GET_PROCEDURE.COB

    This is the COBOL source code for the get routine for the FMS sample application server procedure. This procedure is used in the FMS_INQ_TASK and FMS_UPD_TASK to retrieve a record containing all employee information using the EMPLOYEE_ID as the key.

  - FMS_UPDATE_PROCEDURE.COB

    This is the COBOL source code for the update routine for the FMS sample application server procedure. This procedure is used in the FMS_UPD_TASK to rewrite the record after the user modifies it.

  - FMS_SCROLL_PROCEDURE.COB

    This is the COBOL source code for the get scroll data routine for the FMS sample application server procedure. This procedure is used in the FMS_SCROLL_TASK to retrieve employee information for all employees with a specified last name.

  - FMS_TERM_PROCEDURE.COB

This is the COBOL source code for the termination routine for the FMS sample application server procedure. The routine closes the RMS datafile called FMS_APPL.DAT.

- Message source file, FMS_MESSAGES.MSG

  This is the message file source that contains the error messages for the FMS sample application. These messages are displayed using the ACMS GET MESSAGE clause.

- Command Procedure to build message file, FMS_MESSAGES.COM

  This is a command file to build the FMS messages file.

- Command Procedures to compile and link the procedure server

  - FMS_SERVER_COMPILE.COM

    This is a command procedure to recompile all the procedure server modules. It also creates a new object library (FMS_SERVER.OLB) and inserts all server modules.

  - FMS_SERVER_LINK.COM

    This is a command procedure to relink the procedure server modules into FMS sample application server image. It uses the server object library created in the recompile command procedure.

- Application Definition (includes **BUILD** command), FMS_APPLICATION.ADF

  This is the ACMS application definition for the RI FMS sample application. It uses the FMS_GROUP.TDB task group and includes the **BUILD** command to build the application database (.ADB).

- Menu Definition (includes **BUILD** command), FMS_APPLICATION_MENU.MDB

  This is the ACMS menu definition for the RI FMS sample application. It includes the **BUILD** command to build the menu database (.MDB).

- FMS form definitions

  - FMS_ADD_FORM.FRM

    This is the FMS form used in the add request URP that the add task example uses.

  - FMS_INQ_FORM.FRM

    This is the FMS form used in the inquiry request URP that is used in the inquiry task.

  - FMS_DISP_FORM.FRM

    This is the FMS form used in the inquiry display request URP that is used in the inquiry task.

  - FMS_DEP_FORM.FRM

    This is the FMS form used in the inquiry display request URP and the update display request URP that is used in the inquiry task/update task.

  - FMS_UPD_FORM.FRM

This is the FMS form used in the update inquiry request URP that is used in the update task.

- FMS_SCROLL_FORM.FRM

  This is the FMS form used in the inquiry scroll request that is used in the scroll task.

- FMS_SCROLL_DISP_FORM.FRM

  This is the FMS form used in the display scroll request that is used in the scroll task.

- FMS_HELP_FORM.FRM

  This is the FMS form used as a help form to the FMS_ADD_FORM.FRM.

- Command Procedure to build the FMS form library, FMS_BUILD_FORM_LIB.COM

  This is a command procedure that builds the FMS form library for the FMS sample application.

- Request Interface URP source modules

  - FMS_INIT_LIBRARY.COB

    This is COBOL PROCEDURE source code for the initialization procedure for the FMS RI request library. This procedure creates the FMS workspaces, attaches the terminal, and opens the FMS forms library defined by the logical ACMS$RI_FMS_MENU_LIB.

  - FMS_ADD_REQUEST.COB

    This is COBOL PROCEDURE source code to duplicate the TDMS add request by using FMS service calls to prompt the user employee information. The employee record and workspace record are passed to it as parameters.

  - FMS_INQ_REQUEST.COB

    This is COBOL PROCEDURE source code to duplicate the TDMS inquiry request by using the FMS service calls to prompt the user for badge number. The employee record, the ACMS processing status record, and the workspace record are passed to it as parameters.

  - FMS_INQ_DISP_REQUEST.COB

    This is COBOL PROCEDURE source code to duplicate the TDMS inquiry display request by using the FMS service calls to display the employee information. The employee record is passed to it as a parameter.

  - FMS_UPD_REQUEST.COB

    This is COBOL PROCEDURE source code to duplicate the TDMS update request by using the FMS service calls to prompt the user for badge number. The employee record, the ACMS processing status record, and the workspace record are passed to it as parameters.

  - FMS_UPD_DISP_REQUEST.COB

    This is COBOL PROCEDURE source code to duplicate the TDMS inquiry display/modify request by using the FMS service calls to display the employee information and request user updates. The employee record and workspace record are passed to it as parameters.

- FMS_SCROLL_REQUEST.COB

  This is COBOL PROCEDURE source code to duplicate the TDMS inquiry scroll request by using the FMS service calls to prompt the user for an employee last name. The scroll record, the ACMS processing status record, and the workspace record are passed to it as parameters.

- FMS_SCROLL_DISP_REQUEST.COB

  This is COBOL PROCEDURE source code to duplicate the TDMS scroll display request by using the FMS service calls to display all employees with the requested last name. The scroll record is passed to it as a parameter.

- FMS user action routines (UAR) source modules

  - FMS_TITLE_CHECK_UAR.COB

    This is COBOL PROCEDURE source code to execute as a UAR in the FMS add task. It checks the title field entered on the form with a data file named TITLE_SALARY.DAT.

  - FMS_SALARY_CHECK_UAR.COB

    This is COBOL PROCEDURE source code to execute as a UAR in the FMS add task. It checks the salary field entered on the form with a salary range for the entered title field.

- Command Procedures to COMPILE and LINK Request Library shared image file

  - FMS_REQ_LIB_COMPILE.COM

    This is a command procedure used to recompile the FMS RI request library modules. It also creates a new object library and inserts all the modules.

  - FMS_REQ_LIB_LINK.COM

    This is command procedure used to relink the FMS RI request library shared image. It uses the object library.

E.3.2. FMS Sample Application Run-Time Files

This section contains the run-time files for the FMS sample application.

- TDMS request library file (RLB), FMS_REQUEST_LIBRARY.RLB

  This is the TDMS request library for the RI FMS sample application. It contains all the TDMS requests for the FMS sample application.

- Task group database (TDB), FMS_GROUP.TDB

  This is the ACMS task group database for the RI FMS sample application. This group does not define a request library shared image (.EXE) file in the REQUEST LIBRARY IS clause (only TDMS request library files (RLB)). Because the FMS sample application is built using this task group database, you must define a logical name (ACMS$RI_LIB_libraryname) to use the FMS user request procedures for this group.

- Procedure server object module library, FMS_SERVER.OLB

This is the object library containing the six object modules that make up the ACMS server used in the FMS sample application. It includes the following modules:

- FMS_INIT_PROCEDURE

  Object module for the initialization procedure

- FMS_TERM_PROCEDURE

  Object module for the termination procedure

- FMS_ADD_PROCEDURE

  Object module for the add record procedure

- FMS_GET_PROCEDURE

  Object module for the get record procedure

- FMS_UPDATE_PROCEDURE

  Object module for the update record procedure

- FMS_SCROLL_PROCEDURE

  Object module for the get scroll data procedure

- Procedure server image file, FMS_SERVER.EXE

  This is the procedure server image for the RI FMS sample application.

- Application database (ADB), FMS_APPLICATION.ADB

  This is the ACMS application database for the RI FMS sample application. It uses the FMS_GROUP.TDB task group.

- Menu database (MDB), FMS_APPLICATION_MENU.MDB

  This is the ACMS menu database for the RI FMS sample application.

- Application message file, FMS_MESSAGES.EXE

  This is the message file that contains the error messages for the FMS sample application. These messages are displayed using the ACMS GET MESSAGE clause.

- Request Interface FMS form library, FMS_FORM_LIBRARY.FLB

  This is the FMS form library that contains all the FMS forms used in the FMS sample application.

- Request Interface request library shared image file, FMS_REQUEST_LIBRARY.EXE

  This is the FMS RI request library shared image file that includes all FMS URP modules.

- Request Interface URP object module library, FMS_REQ_LIB.OLB

  This is the object library that contains the object modules for the FMS RI request library shared image file.

- ACMSRI$INIT_LIBRARY

  Object module for the FMS RI request library initialization procedure

- FMS_ADD_REQUEST

  Object module for the FMS RI request library add request procedure

- FMS_INQ_REQUEST

  Object module for the FMS RI request library inquiry request procedure

- FMS_INQ_DISP_REQUEST

  Object module for the FMS RI request library inquiry display request procedure

- FMS_UPD_REQUEST

  Object module for the FMS RI request library inquiry update request procedure

- FMS_UPD_DISP_REQUEST

  Object module for the FMS RI request library update display/modify request procedure

- FMS_SCROLL_REQUEST

  Object module for the FMS RI request library inquiry scroll request procedure

- FMS_SCROLL_DISP_REQUEST

  Object module for the FMS RI request library display scroll request procedure

- FMS_TITLE_CHECK_UAR

  Object module for the FMS UAR

- FMS_SALARY_CHECK_UAR

  Object module for the FMS UAR

- FMS User Action Routine RMS data file, TITLE_SALARY.DAT

  This is the data file that contains title and salary range information. This matches the FMS_HELP_FORM information.

Appendix F. Modifying the FMS Menu Interface

ACMS software includes an FMS-based menu interface to use with the ACMS-supplied RI agent, ACMS$RI_AGENT. The sample FMS form library for the menu interface is in the ACMS$RI_EXAMPLES directory.

This appendix describes how to modify this menu interface to suit the needs of a particular application. To incorporate the modified menu interface into the ACMS$RI_AGENT, see *Chapter 14, "Using the ACMS Request Interface"*.

There are two FMS menu form definitions:

- MENU_MAIN_FORM.FRM

  This is the FMS form used as the top-level (main) menu form. It is similar to the layout of the ACMS default menu format, which includes a number, a keyword, a task or menu indicator (T or M), and descriptive text. At the bottom of the menu is the SELECTION: prompt.

  The menu layout is static text in the FMS form. The only field on the menu form is the selection field. The numbers and keywords are used as named-data indexes and named-data keywords. To change the form for a specific application, modify the static text on the main menu and change the named-data field information. The default main menu is set up for the FMS sample application.

- MENU_HELP_FORM.FRM

  This is the FMS form used when the user types $HELP or presses **PF2** twice at the SELECTION: prompt. It displays a form explaining the functionality available under the FMS menu interface, such as *, -, $EXIT, number, keyword, and so on. Include this form in all FMS menu interface form libraries.

To modify the supplied FMS menu form, MENU_MAIN_FORM.FRM:

1. Make a copy of the form. Do not modify the original copy of MENU_MAIN_FORM.FRM; you must modify a copy of the form. The form MENU_MAIN_FORM must stay the same because the menu interface routine uses that name. For example:

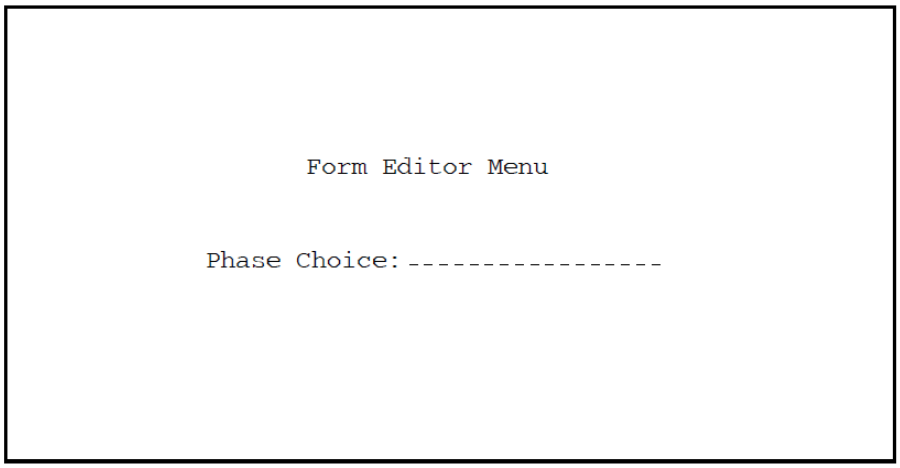
   ```
   $ COPY MENU_MAIN_FORM.FRM newmenu.frm
   ```

2. Edit the form to make the necessary changes.

   ```
   $ FMS/EDIT newmenu.frm
   ```

   FMS responds by clearing the screen and displaying the Form Editor Menu, shown in *Figure F.1, "Form Editor Menu"*. You use this menu to select an action to perform. Type the name of the phase you wish to enter.

Figure F.1. Form Editor Menu



First change the layout of the form. To enter the layout phase, type LAYOUT and press **Return**.

The layout of the ACMS-supplied menu form is similar to the ACMS menu format. The FMS menu form contains a menu header, a selection list, a prompt line, and a message line. The menu header is the name of the menu. The selection list shows the tasks you can run and the menus available.

- The first column in the list is the number of the entry on the menu.
- The second column is a keyword for the item.
- The third column is a label identifying the item as a task (T) or a menu (M).
- The fourth column is text describing the item.
- The prompt line includes both the SELECTION: prompt and blank spaces after the prompt.

Type the number or keyword for the task or menu in the blank space after the prompt. Press **Return** after making your selection.

The FMS menu form contains only one field, the selection field. The remaining data on the screen is background text. In the layout phase, you can change, delete, or add any background text that is needed for a given application; the selection field name and size, however, must remain the same.

The selection list is also used by the menu interface routines. The FMS menu interface routine uses the selection list numbers as indexes or the selection list keywords as keywords for FMS-named data calls to retrieve the task and application names.

3. After modifying the FMS menu form layout, change the Named Data Area of the form. Named Data provides a convenient way for the FMS menu routine to store program parameters with the form instead of coding them into the menu routines. Use Named Data to add additional menu forms and change the existing menu form without changing the menu interface routine.

   Named Data is data that is associated with a form but does not appear on the operator's screen. Named Data exists in the FMS workspace with the form. Define the Named Data to the menu form during the data phase. Enter the data phase by typing PHASE at the form editor menu.

The data can be any string up to 80 characters and is referenced by either name or index. The FMS menu routine subdivides the 80-character data area into ACMS task name and application name. The task name must be first; the application name follows, with at least one space in between them.

Table F.1, "Defining the Named Data Associated with the Form" shows how to define the named data associated with a form.

Table F.1. Defining the Named Data Associated with the Form

Index	Name	Data Area Task Name/Application
1	ADD	FMS_ADD_TASK / FMS_APPLICATION
2	DISPLAY	FMS_INQ_TASK / FMS_APPLICATION

The FMS menu interface routine uses the Named Data area to determine the ACMS task name and the ACMS application name by providing the user with two ways to select a task or menu: by number or by keyword. The user can type either the number from the menu (that is used as a index into the Named Data area) or the keyword from the menu (that is used as a keyword into the Named Data area). You can use the short form of a keyword. Use enough letters to make the selection unique.

The select list numbers and keywords must exactly match the indexes and names in the Named Data area of the menu form. If a user enters a selection that is not contained in the Named Data area, an error message is displayed indicating that it was an invalid selection and to enter another one.

In addition, the FMS menu interface routines define three keywords that implement some of the ACMS menu functionality. The interface routines keywords must be placed in the task-name field of the Named Data area. The keywords are as follows:

- MENU

  If the menu interface routine finds the keyword MENU in the task-name field, FMS assumes that the application-name field contains the name of another menu form in the same FMS menu form library. This keyword enables you to develop menu trees.

  The FMS menu interface routine provides two special characters for moving through the menu tree. This is similar to the ACMS menu facility. The special characters are the asterisk (*) for moving up to the main level menu and the hyphen (-) for moving up to the previous level. The FMS menu interface supports five levels of menus.

- $EXIT

  If the menu interface routine finds the keyword $EXIT in the task-name field, it exits from the RI agent code.

- $HELP

  If the menu interface routine finds the keyword $HELP in the task-name field, it displays the help form (MENU_HELP_FORM) that is provided in the ACMS$RI_EXAMPLES directory. The help form simply describes the user's options at the selection prompt.

4. Create a new FMS menu library.

After modifying the supplied menu form, create a new FMS form library. Use the FMS Form Librarian Utility. Putting forms in form libraries makes the forms available to the Form Driver and the FMS menu interface routine. The create operation makes a new library file and puts one or more binary forms in it, as follows:

```
$ FMS/LIBRARY/CREATE

LIBRARY: newmenulibrary.flb

FORMS:   newmenu.frm,menuhelpform.frm
```

5. Link the FMS menu interface routine.

   You can link the FMS menu interface routine directly into the ACMS$RI_AGENT or into a shared image that is activated at run time. See *Section 14.5.1, "Providing a Menu Interface"* for more information on linking and running the menu interface.

Appendix G. Checklist of References to Platform-Specific Files

An ACMS application executing on OpenVMS must reference images that have been built (either natively or by using the VEST utility) on OpenVMS. This appendix identifies potential areas that may cause platform compatibility errors.

G.1. Task Group Definition

Check that the following objects or image files referenced in the task group definition are compatible with the platform on which the application is executing:

- Procedure server object module
- Procedure server image
- Message object module
- Message executable image
- Form file image

G.2. ADU BUILD GROUP Command

The ADU **BUILD GROUP** command has several qualifiers that reference platform-specific object modules and libraries. If you use the qualifiers **/OBJECT**, **/SYSLIB**, **/SYSSHR**, or **/USERLIBRARY** when building a task group, check that the object modules or libraries you reference are compatible with the platform on which the application is executing.