First Edition (December 1994)

This edition applies to OpenEdition Version 2 DCE Base Services of Program Number 5655-068 and 5655-069 for use with the MVS/ESA SP Version 5 Release 1.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

An ITSO Technical Bulletin Evaluation Form for reader’s feedback appears facing Chapter 1. If the form has been removed, comments may be addressed to:

IBM Corporation, International Technical Support Organization
Dept. H52 Mail Station P099
522 South Road
Poughkeepsie, New York 12601-5400

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1994. All rights reserved.
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.
Abstract

The intent of the redbook is to help expedite the acceptance of MVS/ESA OpenEdition Release 2 by demonstrating the use of OpenEdition. It will help guide the MVS community into the new world of open client/server applications. The redbook is not a general purpose guide to writing DCE applications. There are several other sources for this information both commercially and internal to IBM. This is a guide for the MVS programmer, with perhaps a strong systems orientation, into the new world of OpenEdition. Its purpose is to leverage existing MVS skills into the open systems environment. The assumed business orientation is a large mainframe environment with millions of dollars or perhaps billions of dollars invested in CICS, IMS, and other applications.

(404 pages)
Contents

Abstract ................................................................. iii
Special Notices ......................................................... xvii
Preface ................................................................. xix
How This Document Is Organized ................................ xix
Related Publications .................................................. xxi
Other Publications ..................................................... xxii
Acknowledgments ....................................................... xxiii

Chapter 1. Introduction To OSF/DCE ................................. 1
1.1 DCE Design Considerations .................................... 2
1.2 Distribution Possibilities ...................................... 5
1.3 Open Blueprint .................................................... 8
1.4 Client/Server Model .............................................. 11
1.5 DCE Remote Procedure Call (RPC) ............................ 11
1.6 DCE Threads ....................................................... 13
1.7 Distributed Services ............................................. 15
  1.7.1 Directory Service .............................................. 15
  1.7.2 Security Services ............................................. 15
  1.7.3 Time Services ................................................ 17
  1.7.4 Distributed File System .................................... 18
  1.7.5 DCE Cell ...................................................... 20
1.8 MVS/ESA OpenEdition ............................................ 21

2.1 DCE Application Development Steps .......................... 25
  2.1.1 The DCE Application Development Steps ................. 26
2.2 Application Building ............................................. 27
2.3 Interface Development .......................................... 28
2.4 Server Development .............................................. 29
2.5 Client Development ............................................. 30
2.6 Generating a UUID and IDL File ............................... 31
  2.6.1 Interface Header .............................................. 33
  2.6.2 Interface Body ............................................... 33
  2.6.3 Defining the .acf File ...................................... 35

Chapter 3. Server ........................................................ 37
3.1 Implementing Remote Procedures ................................ 37
3.2 Initializing Server Code ......................................... 38
3.3 Registering the Interface ...................................... 39
  3.3.1 Entry Point Vector .......................................... 39
  3.3.2 rpc_server_register_if() ................................... 40
  3.3.3 Protocol Sequences ......................................... 40
  3.3.4 rpc_server_use_all_protseqs() ........................... 41
  3.3.5 rpc_server_use_protseq() .................................. 41
  3.3.6 rpc_server_use_protseq_ep() ................................ 42
3.4 Advertising the Server ......................................... 42
  3.4.1 Endpoints .................................................... 42
  3.4.2 Endpoint Mapper Daemon, rpcd ............................ 43
  3.4.3 Steps to Advertise the Server .............................. 43

© Copyright IBM Corp. 1994
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Threads</td>
<td>125</td>
</tr>
<tr>
<td>7.2 Data Handling</td>
<td>128</td>
</tr>
<tr>
<td>7.2.1 Conformant Arrays</td>
<td>129</td>
</tr>
<tr>
<td>7.2.2 Unions</td>
<td>137</td>
</tr>
<tr>
<td>7.3 Dynamic Storage allocation</td>
<td>139</td>
</tr>
<tr>
<td>Chapter 8. DL/I and DB2 Server Considerations</td>
<td>141</td>
</tr>
<tr>
<td>8.1.1 Ensuring the Integrity of Data Base Updates</td>
<td>142</td>
</tr>
<tr>
<td>8.2 DB2 Server Overview</td>
<td>143</td>
</tr>
<tr>
<td>8.2.1 Call Attach Facility (CAF) Overview</td>
<td>143</td>
</tr>
<tr>
<td>8.2.2 Call Attach Facility Usage</td>
<td>144</td>
</tr>
<tr>
<td>8.2.3 DB2 Manager Code</td>
<td>146</td>
</tr>
<tr>
<td>8.2.4 DB2 Functional Code</td>
<td>148</td>
</tr>
<tr>
<td>8.3 DL/I Access from the DCE Server to DL/I Databases</td>
<td>152</td>
</tr>
<tr>
<td>8.3.1 Implementation of a CCTL in a DCE Server</td>
<td>154</td>
</tr>
<tr>
<td>8.3.2 Use of the CCTL Interface in the DCE Managers</td>
<td>157</td>
</tr>
<tr>
<td>8.4 Split-up of the DCE Server</td>
<td>161</td>
</tr>
<tr>
<td>8.5 Summary on DL/I and DB2 Access from Manager Code</td>
<td>165</td>
</tr>
<tr>
<td>Chapter 9. IMS Dependant Regions as DCE Clients</td>
<td>169</td>
</tr>
<tr>
<td>9.1 BMP Main Program in C</td>
<td>170</td>
</tr>
<tr>
<td>9.1.1 C Subroutine for BMP C Main</td>
<td>171</td>
</tr>
<tr>
<td>9.2 BMP Main Program in PL/I</td>
<td>171</td>
</tr>
<tr>
<td>9.2.1 C Subroutine for BMP PL/I Main</td>
<td>172</td>
</tr>
<tr>
<td>9.3 BMP Main Program in COBOL</td>
<td>174</td>
</tr>
<tr>
<td>9.3.1 C Subroutine for BMP COBOL Main</td>
<td>175</td>
</tr>
<tr>
<td>Chapter 10. DB2 Dynamic SQL Server</td>
<td>179</td>
</tr>
<tr>
<td>10.1 Dynamic SQL Overview</td>
<td>179</td>
</tr>
<tr>
<td>10.2 Dynamic SQL IDL</td>
<td>181</td>
</tr>
<tr>
<td>Chapter 11. Configuring an Application Client and Server</td>
<td>185</td>
</tr>
<tr>
<td>11.1.1 Steps to Configure an Application Client and Server</td>
<td>185</td>
</tr>
<tr>
<td>11.2 Configuring an Application Server</td>
<td>186</td>
</tr>
<tr>
<td>11.3 OpenEdition DCE Environment Variables</td>
<td>191</td>
</tr>
<tr>
<td>Appendix A. JCL and Source Listings for MATHX and SBIND Examples</td>
<td>197</td>
</tr>
<tr>
<td>A.1 MATHX JCL and Source Listings</td>
<td>197</td>
</tr>
<tr>
<td>A.1.1 Compile MATHX Client and MATHX Client Stub</td>
<td>197</td>
</tr>
<tr>
<td>A.1.2 Compile MATHXS Server and MATHX Server Stub</td>
<td>198</td>
</tr>
<tr>
<td>A.1.3 Compile MATHXM Manager</td>
<td>198</td>
</tr>
<tr>
<td>A.1.4 LINKEDIT MATHX Client</td>
<td>199</td>
</tr>
<tr>
<td>A.1.5 LINKEDIT MATHX Server</td>
<td>199</td>
</tr>
<tr>
<td>A.1.6 RUN MATHX Client</td>
<td>200</td>
</tr>
<tr>
<td>A.1.7 RUN MATHX Server</td>
<td>200</td>
</tr>
<tr>
<td>A.2 Source for MATHX Example</td>
<td>201</td>
</tr>
<tr>
<td>A.2.1 Client Source (MATHXC)</td>
<td>201</td>
</tr>
<tr>
<td>A.2.2 Server Source (MATHXS)</td>
<td>204</td>
</tr>
<tr>
<td>A.2.3 Manager Source (MATHXM)</td>
<td>207</td>
</tr>
<tr>
<td>A.2.4 MATHX OUTPUT</td>
<td>208</td>
</tr>
<tr>
<td>A.3 SBIND JCL and Source Listings</td>
<td>209</td>
</tr>
<tr>
<td>A.3.1 Compile SBIND Client and SBIND Client Stub</td>
<td>209</td>
</tr>
<tr>
<td>A.3.2 Compile SBIND Server and SBIND Server Stub</td>
<td>209</td>
</tr>
<tr>
<td>A.3.3 Compile SBINDM Manager</td>
<td>210</td>
</tr>
<tr>
<td>A.3.4 LINKEDIT SBIND Client</td>
<td>210</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>F.3 Client Source Code</td>
<td>326</td>
</tr>
<tr>
<td>F.4 Server Source Code</td>
<td>333</td>
</tr>
<tr>
<td>F.5 Manager Source Code</td>
<td>341</td>
</tr>
<tr>
<td>Appendix G. Installation Prerequisites</td>
<td>395</td>
</tr>
<tr>
<td>G.1 Required PTFs</td>
<td>396</td>
</tr>
<tr>
<td>G.2 Prerequisites for OpenEdition DCE Application Support 1.1.0</td>
<td>396</td>
</tr>
<tr>
<td>Index</td>
<td>399</td>
</tr>
</tbody>
</table>
Figures

1. OSF/DCE ................................................................. 1
2. Client/Server Design Issues ......................................... 2
3. Distribution Possibilities ........................................... 5
4. Open Blueprint ....................................................... 8
5. Client/Server Model .................................................. 11
6. Overview of DCE RPC ................................................ 12
7. MVS/ESA OpenEdition .............................................. 21
8. The Five Basic Steps ................................................ 25
9. Development Process of a Distributed Application Using RPC .... 27
10. Interface Development .............................................. 29
11. Compiling and Linking a Basic Server ............................. 30
12. Compiling and Linking a Basic Client ............................. 31
13. Sample JCL to Run the UUID Generator ........................... 32
14. Sample TSO/E Command to Create a UUID for the MATHX Application. 32
15. Sample Shell Command to Run the UUID Generator for MATHX. .... 32
16. Basic IDL File .......................................................... 34
17. Basic IDL File (mathx.idl) .......................................... 35
18. Basic Manager Code (mathxm) ....................................... 38
19. Entry Point Vector in Server Stub .................................. 39
20. Basic Error Check (ERRCHK.h) ...................................... 47
21. Basic Client Code (mathxc) .......................................... 50
22. Binding Management Methods ....................................... 52
23. Running an Application Using Explicit Binding .................... 53
24. Explicit Binding IDL File ............................................ 54
25. Running an Application Using Implicit Binding .................... 57
26. Running an Application Using String Binding ....................... 59
27. Running with a Well-Known Endpoint ................................ 63
28. User-Written Files for DCE Applications ........................... 66
29. System-Written Files for DCE Applications ......................... 67
30. HFS File Extensions .................................................. 68
31. Recommended DCE Application Data Set Attributes ................ 69
32. Sample JCL to Compile the IDL file ................................ 72
33. TSO/E Command to Compile the IDL File .......................... 72
34. Sample JCL to Link the Server, Manager, and Server Stub Code. .... 74
35. Sample JCL to Link the Client and Client Stub Code. ............... 75
36. DCE Authentication and DCE Authorization ........................ 78
37. Security Sample IDL File ............................................ 85
38. Server Calling Chain for Authenticated RPC ........................ 87
39. Mbox Data Structure .................................................. 92
40. Calling Chain for DCE Authorization ................................ 94
41. Calling Chain of the Client for Authenticated RPC .................. 99
42. Security controlled by DCE Security ................................ 106
43. Security controlled by RACF ......................................... 109
44. CICS Client and DCE Server .......................................... 113
45. Encrypted DCE Password in RACF Database ......................... 116
46. SAVEPW command .................................................... 118
47. DCEKEY Utility ........................................................ 123
48. Processor Sharing ...................................................... 126
49. Multi-Process Environment: ......................................... 127
50. Conformant Array ...................................................... 130
51. Conformant Array/IDL - Example A with Long int - Type 1 .......... 131
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>52. Conformant Array - IDL Compiled/Header File - Type 1</td>
<td>131</td>
</tr>
<tr>
<td>53. Client/Server Execution</td>
<td>132</td>
</tr>
<tr>
<td>54. Conformant Array/IDL - Example B with char Pointer</td>
<td>133</td>
</tr>
<tr>
<td>55. Conformant Array - IDL Compiled/Header File</td>
<td>133</td>
</tr>
<tr>
<td>56. Client/Server Execution</td>
<td>134</td>
</tr>
<tr>
<td>57. Conformant Array</td>
<td>135</td>
</tr>
<tr>
<td>58. Conformant Array/IDL - Example A - Type 2</td>
<td>136</td>
</tr>
<tr>
<td>59. Conformant Array - IDL Compiled/Header File - Type 2</td>
<td>137</td>
</tr>
<tr>
<td>60. Union</td>
<td>138</td>
</tr>
<tr>
<td>61. Union Structure - IDL Definition</td>
<td>138</td>
</tr>
<tr>
<td>62. Union structure - IDL Compiled</td>
<td>139</td>
</tr>
<tr>
<td>63. IMS and DB2 Overview</td>
<td>141</td>
</tr>
<tr>
<td>64. Declaration for DB2 Call Attach Facility</td>
<td>144</td>
</tr>
<tr>
<td>65. Opening a DB2 Plan</td>
<td>145</td>
</tr>
<tr>
<td>66. Closing a DB2 Plan</td>
<td>145</td>
</tr>
<tr>
<td>67. Execution JCL required for DB2SQL Server</td>
<td>146</td>
</tr>
<tr>
<td>68. DB2 Manager Code Extract</td>
<td>147</td>
</tr>
<tr>
<td>69. Interface Definition Language for CUSLST</td>
<td>149</td>
</tr>
<tr>
<td>70. DB2 Functional Code</td>
<td>150</td>
</tr>
<tr>
<td>71. DL/I FastPath Access in the DCE Environment</td>
<td>153</td>
</tr>
<tr>
<td>72. Including CCTL and DRA Within the Server.</td>
<td>154</td>
</tr>
<tr>
<td>73. DFSPZP</td>
<td>155</td>
</tr>
<tr>
<td>74. IMS Server JCL</td>
<td>155</td>
</tr>
<tr>
<td>75. Initializing CCTL and DRA</td>
<td>156</td>
</tr>
<tr>
<td>76. DRA TCB Structure Prepared by CCTL Initialization</td>
<td>157</td>
</tr>
<tr>
<td>77. Terminating the CCTL-DBCTL Connection</td>
<td>157</td>
</tr>
<tr>
<td>78. Manager Code Split-up</td>
<td>158</td>
</tr>
<tr>
<td>79. Scheduling the PSB</td>
<td>159</td>
</tr>
<tr>
<td>80. Handling the PCB</td>
<td>160</td>
</tr>
<tr>
<td>81. Fetch Routine</td>
<td>162</td>
</tr>
<tr>
<td>82. Composition of Fetched Functional Load Module</td>
<td>163</td>
</tr>
<tr>
<td>83. Fetching versus Calling the Server Functional Code</td>
<td>164</td>
</tr>
<tr>
<td>84. TCB Layout with DB2 and DL/I</td>
<td>165</td>
</tr>
<tr>
<td>85. Composition of DCE Server with DB2 and DL/I Access</td>
<td>166</td>
</tr>
<tr>
<td>86. Server Code Initialization (also via CTLINIT)</td>
<td>167</td>
</tr>
<tr>
<td>87. DCECCTL Messages</td>
<td>167</td>
</tr>
<tr>
<td>88. Server Messages</td>
<td>168</td>
</tr>
<tr>
<td>89. Manager Code</td>
<td>168</td>
</tr>
<tr>
<td>90. IMS Dependent Regions as DCE Client</td>
<td>169</td>
</tr>
<tr>
<td>91. JCL to Run</td>
<td>170</td>
</tr>
<tr>
<td>92. BMP Main Program in C</td>
<td>170</td>
</tr>
<tr>
<td>93. Skeleton for Subroutine in C</td>
<td>171</td>
</tr>
<tr>
<td>94. Main Program in PL/1</td>
<td>172</td>
</tr>
<tr>
<td>95. Skeleton Subroutine in C from PL/1</td>
<td>173</td>
</tr>
<tr>
<td>96. BMP Main Program in COBOL</td>
<td>174</td>
</tr>
<tr>
<td>97. Skeleton Subroutine in C from COBOL</td>
<td>176</td>
</tr>
<tr>
<td>98. Unions/Conformant Arrays</td>
<td>180</td>
</tr>
<tr>
<td>99. Conformant Array of Discriminated Unions</td>
<td>181</td>
</tr>
<tr>
<td>100. Parameters for the Dynamic SQL Server</td>
<td>182</td>
</tr>
<tr>
<td>101. ValueList Structure</td>
<td>182</td>
</tr>
<tr>
<td>102. ColData Discriminated Union</td>
<td>183</td>
</tr>
<tr>
<td>103. ColData Union Member Structures</td>
<td>184</td>
</tr>
<tr>
<td>104. JCL to compile the MATHXC client code.</td>
<td>197</td>
</tr>
<tr>
<td>105. JCL to Compile the MATHXS Server Code</td>
<td>198</td>
</tr>
<tr>
<td>106. JCL to Compile the MATHXM Manager Code.</td>
<td>198</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>107</td>
<td>JCL to LINKEDIT the MATHX Client Code</td>
</tr>
<tr>
<td>108</td>
<td>JCL to LINKEDIT the MATHX Server Code</td>
</tr>
<tr>
<td>109</td>
<td>JCL to RUN the MATHX Client</td>
</tr>
<tr>
<td>110</td>
<td>JCL to RUN the MATHX Server</td>
</tr>
<tr>
<td>111</td>
<td>Basic Client Code (MATHXC)</td>
</tr>
<tr>
<td>112</td>
<td>Basic Server Code (MATHXS)</td>
</tr>
<tr>
<td>113</td>
<td>Basic Manager Code (MATHXM)</td>
</tr>
<tr>
<td>114</td>
<td>PRINTF statements for MATHX Example</td>
</tr>
<tr>
<td>115</td>
<td>JCL to Compile the SBINDC Client Code</td>
</tr>
<tr>
<td>116</td>
<td>JCL to Compile the SBINDS Server Code</td>
</tr>
<tr>
<td>117</td>
<td>JCL to Compile the SBINDM Manager Code</td>
</tr>
<tr>
<td>118</td>
<td>JCL to LINKEDIT the SBIND Client Code</td>
</tr>
<tr>
<td>119</td>
<td>JCL to LINKEDIT the SBIND Server Code</td>
</tr>
<tr>
<td>120</td>
<td>JCL to RUN the SBIND Client</td>
</tr>
<tr>
<td>121</td>
<td>JCL to RUN the SBIND Server</td>
</tr>
<tr>
<td>122</td>
<td>Client Code (SBINDC)</td>
</tr>
<tr>
<td>123</td>
<td>Server Code (SBINDS)</td>
</tr>
<tr>
<td>124</td>
<td>Manager Code (SBINDM)</td>
</tr>
<tr>
<td>125</td>
<td>PRINTF statements for SBIND Example</td>
</tr>
<tr>
<td>126</td>
<td>Security Sample Server Code (mboxs.c)</td>
</tr>
<tr>
<td>127</td>
<td>Security Sample Server Manager Code (mboxm.c)</td>
</tr>
<tr>
<td>128</td>
<td>Security Sample Security Code (mboxsec.c)</td>
</tr>
<tr>
<td>129</td>
<td>Security Sample Client Code (mboxc.c)</td>
</tr>
<tr>
<td>130</td>
<td>Security Sample Header File (mboxcom.h)</td>
</tr>
<tr>
<td>131</td>
<td>Library File (mboxlib.c)</td>
</tr>
<tr>
<td>132</td>
<td>Message REXX exec</td>
</tr>
<tr>
<td>133</td>
<td>Messagebox Server JCL</td>
</tr>
<tr>
<td>134</td>
<td>Sample Source for Well-Known Endpoint</td>
</tr>
<tr>
<td>135</td>
<td>Sample Source for MATHX Server with Well-Known Endpoint</td>
</tr>
<tr>
<td>136</td>
<td>OUTPUT Sstatements for Well-Known Exexample</td>
</tr>
<tr>
<td>137</td>
<td>Sample Makefile for OE Example.</td>
</tr>
<tr>
<td>138</td>
<td>DB2 IDL</td>
</tr>
<tr>
<td>139</td>
<td>DB2 Header</td>
</tr>
<tr>
<td>140</td>
<td>DB2 Client</td>
</tr>
<tr>
<td>141</td>
<td>DB2 Server</td>
</tr>
<tr>
<td>142</td>
<td>DB2 Manager</td>
</tr>
<tr>
<td>143</td>
<td>DB2 Function Code</td>
</tr>
<tr>
<td>144</td>
<td>IDL</td>
</tr>
<tr>
<td>145</td>
<td>DL/I Header</td>
</tr>
<tr>
<td>146</td>
<td>DL/I Client</td>
</tr>
<tr>
<td>147</td>
<td>DL/I Server</td>
</tr>
<tr>
<td>148</td>
<td>DL/I Manager</td>
</tr>
<tr>
<td>149</td>
<td>DL/I Function Code</td>
</tr>
<tr>
<td>150</td>
<td>BMP Main Source</td>
</tr>
<tr>
<td>151</td>
<td>C Subroutine for BMP Source</td>
</tr>
<tr>
<td>152</td>
<td>BMP Main Source</td>
</tr>
<tr>
<td>153</td>
<td>PLI Subroutine for BMP Source</td>
</tr>
<tr>
<td>154</td>
<td>BMP Main Source</td>
</tr>
<tr>
<td>155</td>
<td>COBOL Subroutine for BMP Source</td>
</tr>
<tr>
<td>156</td>
<td>Dynamic SQL Server IDL</td>
</tr>
<tr>
<td>157</td>
<td>Dynamic SQL Server Header</td>
</tr>
<tr>
<td>158</td>
<td>Dynamic SQL Server Client</td>
</tr>
<tr>
<td>159</td>
<td>Dynamic SQL Server</td>
</tr>
<tr>
<td>160</td>
<td>Dynamic SQL Server Manager</td>
</tr>
</tbody>
</table>
## Tables

1. Files that Need to be Copied or Created in HFS ........................................ 65
2. Files That Are Compiled with Your Application Source Code .................. 71
3. Comparison of RACF and DCE Security ................................................... 105
4. DB2 Client/Server ..................................................................................... 143
5. DL/I Client/Server Modules ...................................................................... 143
6. DB2 Dynamic Server Parameters ............................................................... 181
7. OpenEdition DCE Environment Variables ................................................. 192
8. Install Before OpenEdition DCE Base Services 1.1.0 ......................... 395
9. Install PTFs Before OpenEdition DCE Base Services 1.1.0 .................. 396
10. Install Before OpenEdition DCE Application Support 1.1.0 ............... 397
This publication is intended to help IBM customers and system engineers to develop DCE applications on MVS/ESA. The information in this publication is not intended as the specification of any programming interfaces that are provided by MVS/ESA OpenEdition DCE Base Service. See the PUBLICATIONS section of the IBM Programming Announcement for OpenEdition Distributed Computing Environment for MVS/ESA (294-501) for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM’s product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM’s intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 208 Harbor Drive, Stamford, CT 06904 USA.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM (VENDOR) products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer’s ability to evaluate and integrate them into the customer’s operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The following document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AIX
CICS/ESA
COBOL/370
DB2
ES/9000
IMS/ESA
OpenEdition
Sysplex Timer

CICS
CICS/6000
DATABASE 2
Enterprise Systems Architecture/390
IBM
MVS/ESA
S/390
System/390
The following terms are trademarks of other companies:

Open Software Foundation, OSF, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc.
DEC, DIGITAL are trademarks of Digital Equipment Corporation
HP and Hewlett-Packard are trademarks of the Hewlett-Packard Company
AFS and Transarc are trademarks of the Transarc Corporation
NFS and Network File System are trademarks of Sun Microsystems, Inc.
X/Open and UNIX are trademarks of the X/Open Company Limited
POSIX is a trademark of the Institute of Electrical and Electronic Engineers, Inc.
Preface

This document is intended to be used by IBM customer and system engineers that are interested in developing DCE applications on MVS/ESA. Its purpose is to show a person with an MVS/ESA background how to get a distributed application to work in an MVS/ESA environment. It is a "cookbook" showing every step needed to get a client and a server application to work on MVS/ESA DCE.

This book borrows heavily from a previous International Technical Support Organization publication, Developing DCE Applications for AIX, OS/2 and Windows. The first six chapters are based on this publication. Most changes are minor editorial and organizational changes. Some platform-related issues were rewritten; for example, MVS/ESA provides base system thread support and, thus, does not need DCE threads. The programming examples from the original book were ported to MVS/ESA as well. This reuse of material illustrates one of the main customer values in using DCE-portability.

The later portion of the book, chapters eight through eleven, focuses on things of specific interest to the MVS/ESA platform. A major section was added to the security discussion of the previous ITSO book. This is because of the increased emphasis on security in an MVS/ESA environment and the need to integrate DCE security with MVS/ESA security (that is RACF). The later chapters of this book, chapters eight, nine, and ten, concentrate on access to DB2 and DL/1 data. These are major concerns for the MVS/ESA application development community. Usage of the call attach facility (CAF) for DB2 and database control (DBCNTL) for DL/1 are illustrated.

This book assumes the reader is familiar with both OSF DCE and MVS/ESA. There is no attempt to discuss the fundamentals of either DCE or MVS/ESA. Please check the "Related Publications" on page xxi for further information.

A diskette accompanies this book and it contains the source code for all the examples discussed. The examples in this book were developed and test using ECIP I and ECIP II systems. Early Customer Involvement Program (ECIP) was part of the IBM quality program for MVS/ESA OpenEdition. Its purpose was to get customer feedback on the product early enough in the development cycle to fix problems before they got wide distribution. ECIP I started in early 1993 and provided about a dozen customers with very early version of MVS/ESA DCE. The ECIP II expanded the customer set. It started the summer of 1994 and continued to provide early versions of the OpenEdition product to customers. Although we expect little change in the delivered product from ECIP II, readers should be warned that there may be some differences from what is described in this book.

How This Document Is Organized

The document is organized into eleven chapters and eight appendices. We give a brief overview of the sections of the book in the following section.

• Chapter 1, "Introduction To OSF/DCE"

This chapter provides a brief introduction to distributed application design issues, the client/server model, DCE, and MVS/ESA OpenEdition.
• Chapter 2, “Development Process and Interface Development”
  The Development Process chapter describes the process you should use in
  writing DCE applications.
• Chapter 3, “Server”
  This chapter describes the things the application programmer must do to
  establish and initialize an application server on MVS/ESA.
• Chapter 4, “Client”
  This chapter describes the things the application programmer must do to
  establish an application client on MVS/ESA.
• Chapter 5, “Compiling and Link Editing”
  This chapter describes the compilation and linking process for MVS/ESA
  applications.
• Chapter 6, “Security”
  This chapter describes the things programmers must do to establish a
  secure environment for their application. In addition to DCE security, RACF
  integration is covered.
• Chapter 7, “Threads and Data”
  This chapter discusses two major technologies that must be mastered to do
  effective programming in a DCE environment: thread programming and
  different data formats.
• Chapter 8, “DL/I and DB2 Server Considerations”
  This chapter discussed the techniques needed to access DL/I and DB2 data.
• Chapter 9, “IMS Dependant Regions as DCE Clients”
  This chapter discusses how to establish an application client from an IMS
  dependent region.
• Chapter 10, “DB2 Dynamic SQL Server”
  This chapter shows you how to develop a sophisticated example of a DCE
  applications.
• Chapter 11, “Configuring an Application Client and Server”
  After completing your application, it must be installed on DCE. This chapter
  shows how to configure an application in an existing DCE cell.
• Appendix A, “JCL and Source Listings for MATHX and SBIND Examples”
  This appendix contains the JCL and source listing for the examples shown is
  chapters 3 and 4.
• Appendix B, “DCE Security Example”
  This appendix contains the source code for the example from chapter 6.
• Appendix C, “MATHX Source (Well-Known Endpoint)”
  This appendix contains the well-known endpoint example from chapter 3.
• Appendix D, “IMS and DB2 Examples”
  This appendix contains the source code for the IMS and DB2 examples from
  chapter 8.
• Appendix E, “IMS Dependent Region as DCE Client Source”
This appendix contains the source code for the example in chapter 9.

- Appendix F, “Dynamic SQL Server”
  This appendix contains the source code for the dynamic SQL server described in chapter 10.

- Appendix G, “Installation Prerequisites”
  The installation prerequisites and PTFs needed to run MVS/DCE are listed in this appendix.

Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this document.

- *Developing DCE Applications for AIX, OS/2, and Windows*, GG24-4090-01
- *Distributed Computing Environment Understanding the Concepts*, GC09-1478
- *Introduction OpenEdition MVS*, GC23-3010
- *MVS/ESA Client/Server Presentation Guide*, GG24-3931
- *MVS/ESA OpenEdition DCE: Installation and Configuration*, GG24-4480
- *OpenEdition Distributed Computing Environment Base Services MVS: Administration Reference*, SC09-1486
- *OpenEdition Distributed Computing Environment Base Services MVS: Application Support Configuration and Administration*, SC09-1659
- *OpenEdition Distributed Computing Environment Base Services MVS: Configuration and Getting Started*, SC09-1490
- *OpenEdition Distributed Computing Environment Base Services MVS: Messages and Codes*, SC09-1483
- *OpenEdition Distributed Computing Environment Base Services MVS: Planning*, SC09-1489
- *MVS/ESA OpenEdition Release 2 Technical Overview*, GG24-4095
• MVS/ESA OpenEdition MVS Advanced Application Programming Tools, SC23-3017
• MVS/ESA OpenEdition Command Reference, SC23-3014
• MVS/ESA Support for IEEE POSIX Standards Technical Presentation Guide, GG24-3867
• IBM SAA AD/Cycle C/370 General Information, GC09-1358
• IBM SAA AD/Cycle C/370 Programming Guide for LE/370, SC09-1840
• IBM SAA AD/Cycle C/370 Programming Guide for C/370 Library, SC09-1841
• IBM SAA AD/Cycle C/370 User’s Guide, SC09-1763
• IBM SAA AD/Cycle Language Environment Concepts, GC26-4531
• TCP/IP Tutorial and Technical Overview, GG24-3376
• IBM DATABASE 2 Version 3 DB2 Application Programming and SQL Guide, SC26-4889
• IMS/ESA Customization Guide: Database, SC26-3064

A complete list of International Technical Support Organization publications, with a brief description of each, may be found in:


To get listings of ITSO technical bulletins (redbooks) online, VNET users may type:

TOOLS SENDTO WTSCPOK TOOLS REDBOOKS GET REDBOOKS CATALOG

How to Order ITSO Technical Bulletins (Redbooks)

IBM employees in the USA may order ITSO books and CD-ROMs using PUBORDER. Customers in the USA may order by calling 1-800-879-2755 or by faxing 1-800-284-4721. Visa and Master Cards are accepted. Outside the USA, customers should contact their IBM branch office.

Customers may order hardcopy redbooks individually or in customized sets, called GBOFs, which relate to specific functions of interest. IBM employees and customers may also order redbooks in online format on CD-ROM collections, which contain the redbooks for multiple products.

Other Publications

The following publications are available from booksellers and provide details of the distributed computing environment and related topics.

• Introduction to DCE, Prentice-Hall Inc.
• DCE Users Guide and Reference, Prentice-Hall Inc.
• DCE Application Development Reference, Prentice-Hall Inc.
• DCE Administration Guide, Prentice-Hall Inc.
• DCE Administration Reference, Prentice-Hall Inc.

Acknowledgments

The advisor for this project was:

Bob Dahle
International Technical Support Organization, Poughkeepsie Center

The authors of this document are:

Eric Finkelstein, IBM Australia
Robert Mount, IBM USA
Saul Reiser, IBM USA
Gregg Swanson, Boeing Computer Services
Egide Van Aerschot, IBM Belgium

This publication is the result of a residency conducted at the International Technical Support Organization, Poughkeepsie Center.
Chapter 1. Introduction To OSF/DCE

Open Software Foundation (OSF) Distributed Computing Environment (DCE) is composed of a set of services that support the development, use, and maintenance of distributed applications. The services, which are also called DCE technology components, fall into two generic categories:

- Programming services
- Distributed services

DCE threads and RPC are programming services that include libraries that implement application programming interfaces (APIs) and program development tools.

The remaining DCE technologies are distributed services: the directory server, the time server, the security server, and the file server. They consist in part of a daemon, or server process, that runs continuously on a machine and responds to requests that are sent over the network. The distributed services are equipped with administrative components to manage the service. They also have APIs through which programmers can access the server.

Application programmers deal mostly with the programming services. Although the distributed services are accessed through their APIs, the programmer usually uses the distributed services indirectly - through RPC, which in turn uses the distributed services APIs.

This document describes how you can develop distributed applications that makes use of the DCE RPC and Threads services.
1.1 DCE Design Considerations

Building software always seems harder than it ought to be. It takes longer than expected, the functionality and performance are not as expected and the resulting product is not easily changed. A software “crisis” has been going on for 25 years. The crisis has now reached a point where it is impacting the life of the enterprise. With a two to three-year backlog of applications to be developed and a two to three-year software development cycle, information systems are not able to provide the competitive advantages to the business that the underlying technology promises.

In addition, the applications that are needed in the 1990s are becoming much more complex. Distributed applications are now needed. These distributed applications involve multiple platforms from multiple vendors using many software offerings. One of the major objectives of OSF/DCE is to address some of these issues. Let’s briefly go through some of the issues associated with client/server design.

- Distribution Points

Perhaps one of the first questions that comes to mind when thinking of distributed applications is where will the application be partitioned? Although there appears to be an infinite number of possibilities, developers are finding several distribution models helpful in designing their applications. We will discuss these models on Figure 3 on page 5.

- System Software Support

Key to the application design process is the base upon which application rests. That is, what type of system support is available. The type of things that have to be dealt with here are multi-tasking, scheduling, resource
allocation, locking and recovery. Although the technology associated with multi-tasking is generally thought of as an operating system function, in the client/server environments application programmers may have to deal with these technologies also. The issue becomes particularly crucial when writing servers that support many clients. Many of the complexities of distributed application development can be handled by system software. One of the main purposes of DCE is to provide a layer of system software to ease the application development process.

- **Inter-process Communication**
  
The most popular forms of inter-process communication are conversation, remote procedure call, and message queueing. Although client/server applications can be written using any of these inter-process communication technologies, the remote procedure call is probably the most natural. We will, of course, be focusing on the remote procedure call in this presentation.

- **Rendezvous**
  
  One of the obvious concerns for client/server applications is how the client is going to find the server that it needs. That is, how they are going to find each other or rendezvous. DCE provides many services for addressing these concerns with the directory services, but the application developer has to invoke the services. We will see later in this presentation how this affects the server design and in some cases the client design.

- **Security**
  
  As we have discussed before, security in a client/server environment is more complex than in a single-host environment. The risks are greater and the solutions are more involved. The application design issue is the potential risk associated with a security breach versus the overhead of the security protection. The DCE security services provide a range of security protection that an application can use.

- **Modular Design**
  
  Modular design techniques are important in any software design. Client/server design requires a stricter adherence to good modular design practices. The interface between modules must be more exact than in centralized designs. In centralized design many modularity problems can be resolved by shared global or external data; this is not possible in a client and server environment. One of the motivating factors in the movement toward client/server is the reuse of servers in application construction. The reuse of a server is dependent on how coherent its design is.

- **Parallel Processing**
  
  One of the appealing characteristics of client/server applications is that they can use many processors to accomplish a given task. Parallel processing application requirements impact the way both the client and server are designed. DCE provides multi-threading support for parallel processing. We will see later how this is accomplished.

- **Language Support**
  
  Which programming language is used is an important consideration in application design. DCE is written in C and designed for use by C programmers. The data representation in DCE is patterned after C, the client/server interface assumes a C development process; and the application programming interfaces assume a C caller. It isn’t clear how
easily another programming language such as COBOL or PL/1 will work directly with DCE.

- **Stateful/Stateless Servers**
  
  If a server can be called repeatedly by a specific client, it may make sense to have the server remember who has called it before and with what results. The DFS is an example of a stateful server that remembers that the client has opened a file when it comes to make a read request and so on. Care will have to be taken over the design of such a server where the recovery scenarios could be quite complex.

- **Multiple Instances**
  
  Servers could be replicated many times. You may wish to replicate servers for performance or availability reasons. Are these going to provide identical services or are they going to be different? How will a client choose between multiple instances? Also, servers must be able to support several clients at the same time. How will this be handled?

- **Data Conversion**
  
  There are many ways of representing data in a computer. It seems like every possible alternative has been tried by some vendors. In centralized processing, the different data types are just an idiosyncrasy of the platform. In a client/server environment, the different data types and format becomes a major design issues. The kinds of problems that must be addressed are ASCII/EBCDIC, big-endian/little-endian, floating point formats, code pages, and so on.

- **Data Placement**
  
  Where data is placed in the network can become a significant performance and availability issue. If the data and the application that uses the data are in different places, the network can bottleneck and long delays can result. Data that is widely used can shut down the complete environment when not available. Distributed design can be driven by data placement. DCE will not tell the designer where to put the data, but when the decision is made, it can help with the solution. Typical solutions involve techniques of fast data movement and replication of the data.

- **Network**
  
  There are many kinds of telecommunication networks. Many enterprises have several networks. Each network’s software provides a programming interface that is different. This means, for example, with a TCP/IP network the sockets or streams interface is used; for a SNA network, the programmer may use the LU 6.2 protocol. In dealing with these protocols, the programmer must be familiar with the network addressing schemes, recovery processes, message buffering, and so on.
1.2 Distribution Possibilities

What are the methods of splitting programs in order to distribute them? Theoretically, the application designers can divide their program any place in the execution sequence. That is what is implied by Figure 3. IBM did a study of those customers developing distributed applications to see where the splits actually take place. We saw a pattern in these applications. Basically, they fell into one of eight groups: non-distributed, remote presentation, front-ending, distributed logic, staged data, resource centric, process driven, and multi-application. The non-distributed applications are not of interest to this presentation and are included only for contrast with the distributed designs. Let’s look at the others:

- **Remote Presentation**
  Remote presentation could be represented in Figure 3 as a line to the far left of the Display Processing box. There is no application code stored on the workstation. The presentation service is distributed, part on the workstation and part on the host. This type of arrangement uses interface management systems such as X-windows. This approach is often used as the simplest way to attach a workstation to a host. Typically users run personal productivity software on the workstation and use multiple windows with terminal emulators to access several hosts.

- **Front-Ending**
  The front-ending design differs from remote presentation because the workstation has front-end application logic. The front-end application logic may be used to simply transform the user’s interface or to integrate information from other sources. In either case, the display image is
generated twice - once at the host and once at the workstation. Easel** for OS/2 is an example of this type of design.

- **Distributed Logic**

In the distributed logic design, the workstation and host application components interact with each other directly by program-to-program communication. The conversational, remote procedure call, and the message queueing models are used. This type of design is usually a new application in contrast to front-ending. An important consideration for this design is that the stored data is centralized on the back-end system. The business needs of the enterprise may dictate centralized data for sharing, integrity, or security reasons.

- **Data Staging**

Data staging typically involve a 3-tiered design approach. The master copy of the data remains on a regional back-end system, while snapshots of the data are staged to local or departmental systems. This provides performance and availability improvements for the users, performance improvements through quicker response time, and less demand on the network. Availability is provided through no single source of failure.

- **Resource Centric**

Remote resources are accessed through programming interfaces used for local resources in the resource centric design. A well-known example of this is the Network File System. A resource-centric design works best when the amount of data accessed is easily supported by the available communication bandwidth. If large amounts of data have to be moved, significant performance problems can result.

- **Process-driven**

The process-driven design may be characterized by a step-wise execution of the application. Each application may be viewed as a job step in a higher-level process that is designed to accomplish a business-oriented task. Typically, a workflow manager regulates the overall execution of the business process. In this type of design, a message-style service often seems to be preferred over conversational or RPC.

- **Multiple Application**

The multiple-application design has application logic and stored data split apart and distributed. The data is private to the distributed application and is not accessible by applications on other nodes. The various pieces of the application are active at the same time and engage in simultaneous communication with each other. This design is suggestive of an object-oriented style where each application encapsulates its data.

So where does DCE fit with these design models? Resource-centric design with NFS made RPC a common practice in the UNIX world. It is likely that system vendors will continue to exploit RPC to provide distributed services. Remote presentation and front-ending will also be supported by software vendor products that will use RPC. It is likely that process-driven applications will prefer the message queueing inter-process communication technology.

Application programmers will be using DCE for writing application that fit the distributed logic, data staging, or multiple-application design models.
The distributed-design models discussed this section are based on a paper written by Dr. John Shedletsky and John Rofrano. The complete results the customer study is in Application reference design for distributed systems, IBM Systems Journal Vol.32 No.4, 1993.
1.3 Open Blueprint

Open Blueprint

The Open Blueprint is the structure that IBM is using to allow the network of systems to function as a unit, as a network operating system. A network operating system is made up of multiple systems that are separated from each other and are connected by a communication network. In the network operating system enabled by the Open Blueprint, each individual logically contains the services described below. However, it is not necessary for each individual system to physically contain all the services included in the Open Blueprint. Just as an operating system provides the management of resources on a single system, a network operating system provides for the management across the network of the same type of resources: files, databases, printers, transactions, software packages, documents, jobs, and so on. The equivalent facilities or services in each individual system work together to provide support for distributed and client/server applications. Figure 4 represents one instance of a system in the network operating system.

The Open Blueprint addresses the challenges of the open environment by viewing a system as part of a distributed network and viewing the network as if it were a single system. It is represented in Figure 4.

The Open Blueprint serves four major roles:

- It helps users think about, discuss, and organize products and applications in an open, distributed environment.
- It describes IBM’s directions for products and solutions in the open, distributed environment.
• It guides developers as they meet users’ needs by supplying products and solutions that include the appropriate function and that can be integrated and can interoperate with other installed products.

• It provides a context for the incorporation of new technologies into a distributed environment.

A goal of the Open Blueprint is to provide consistency among IBM products and related products such that they work together to achieve a high level of systemic value. Since the wants and needs of users include openness and product/vendor heterogeneity, the Open Blueprint is based on a combination of existing and emerging industry standards. The fundamentals that allow this support are heterogeneous network support as per the IBM Networking Blueprint; the security and directory protocols and usage as per the Open Software Foundation Distributed Computing Environment; participation in systems management as per the defined set of standard interfaces and protocols; and, for object-oriented implementations, adherence to the Object Management Group Common Object Request Broker Architecture (OMG CORBA).

It is a structure that enables a network of operating systems to function as a unit, as a network operating system comprising multiple systems separated from each other and connected by a communication network.

Just as an operating system provides the management of resources on a single system, a network operating system provides for the management across the network of the same types of resources: files, databases, printers, transactions, software packages, documents and jobs.

It promotes the integration of multi-vendor systems and simplifies the more cumbersome aspects of distributed computing. This integration improves the single system image that the end user and application developer perceive for the distributed system.

The Open Blueprint describes technical attributes and characteristics of supporting software, reflects desirable functional modularity, provides software principles and guidelines, and specifies important boundaries and interfaces.

Much of the function described in the Open Blueprint exists and is being developed and used in product form. Over time, the Open Blueprint will be expanded with additional function, and additional product implementations will be provided. However, it expresses technical direction only. None of it should be construed as a commitment to deliver any of the functions described, nor should any inference to that effect be made.

In summary, the Open Blueprint is a structure that will help IBM and others deliver integrated, interoperable products and solutions. In addition to the business aspects of the application, programs written to the network services layers have to deal with all of the client/server design issues discussed in Figure 2 on page 2.

DCE solves a lot of the challenges that face the network services programmer, but programming at this level is not trivial. Having solved many of the problems facing the network services programmer, it is reasonable to ask if there is anything left to trouble the application developer. As you might have guessed, there are still many problems. A partial list of the problems are:
Does this mean that DCE is only of value for the roll-your-own pioneer? No, of course not. It is estimated that distributed applications and resource managers written without the use of distributed system services devote 40% of their code (and a larger percentage of the development time) to providing the equivalent function. This means that when you buy a distributed database product, at least 40% of the development cost for the product came from these distributed services. When you write a distributed application for your own business needs, you can write it 40% faster or with 40% fewer resources.
1.4 Client/Server Model

A useful model for implementing distributed application is the client/server model. In this model, a distributed application consists of two parts: a client program and a server program. The two programs are usually running on different systems attached to a network, and talking with each other using its unique protocols. The client’s role is to ask the server part to carry out user’s requests on behalf of its user. The server then fulfills the client’s request. Figure 5 illustrates the client/server model.

![Client/Server Model Diagram](image)

For example, the distributed file system (DFS) Service offered by DCE is a distributed application based on the client/server model. The client program of the DFS resides on every DCE system. With the underlying help of the DFS client program, you will be able to access files that reside on remote hosts. On remote hosts, the DFS server program is running, and listening for file-access requests sent by client programs on behalf of users. After the DFS server program fulfills the request and sends back a response to the client host, the DFS client program returns the response to the user.

1.5 DCE Remote Procedure Call (RPC)

A distributed application based on the client/server model consists of two parts: the client side of the application, which runs on one machine and makes a request for service on behalf of a user, and the server side of the application, which runs on another machine on the network and fulfills the service request. The two pieces of code on two different machines need to be able to communicate across the network. One model for implementing communications between the client and server of an application is the remote procedure call (RPC). In this model, a client makes what appears to be a normal procedure call. The RPC mechanism translates the procedure call into network communications. The server receives the RPC and runs the requested procedure. Any return value is transmitted back to the client. Figure 6 on page 12 illustrates how a distributed application using DCE RPC runs on a network environment.
An RPC server or client contains application code, one or more RPC stubs, and a copy of the RPC run-time. An RPC stub is an interface specific code module that uses an RPC interface to pass and receive arguments. A server and a client contain complementary stubs for each RPC interface they share. RPC application code implements and calls remote procedures and also calls any RPC run-time routines the application needs. The DCE RPC run-time manages communications for RPC applications. In addition, a library of run-time routines enables RPC applications to set up their communications, manipulate information about servers, and perform optional tasks such as remotely managing servers and accessing security information.

1.5.1.1 RPC Interface
Traditionally, calling code and called procedures share the same address space and are linked together. In an RPC application, the calling code and the called remote procedures are not linked together. Rather, they communicate indirectly through an RPC interface. An RPC interface is a logical grouping of operations, data types, and constants that serves as a unique definition for a set of remote procedures. DCE RPC interfaces are compiled from formal interface definitions written by application developers using the DCE Interface Definition Language (IDL).

1.5.1.2 Stubs
A stub uses its RPC interface to pass call arguments. The stub performs basic support functions for remote procedure calls. For instance, a stub prepares input and output arguments for transmission between systems with different forms of data representation. The stubs use the RPC run-time to send and receive remote procedure calls. The client stub can also use the run-time to find servers for the client.

When a client application calls a remote procedure, the client stub first prepares the input arguments for transmission. The process for preparing arguments for transmission is known as marshalling. Marshalling converts call arguments into a byte-stream format and packages them for transmission. Upon receiving call arguments, a stub unmarshalls them. Unmarshalling is the process by which a stub disassembles incoming network data and converts this data into application...
data using a format that the local system understands. Marshalling and unmarshalling both occur twice for each remote procedure call:

1. The client stub marshalls input arguments and unmarshalls output arguments.
2. The server stub unmarshalls input arguments and marshalls output arguments.

The DCE IDL compiler is a tool with which you compile interface definitions. It produces stubs that invoke marshalling and unmarshalling routines. For data types that are defined in the IDL file, a library of marshalling and unmarshalling routines are built into the header files as macro calls.

To build the client for an RPC application, a developer links client application code to a client stub for each RPC interface of the application. To build the server, the developer links the server application code to the corresponding server stubs.

1.5.1.3 RPC run-time
In addition to one or more RPC stubs, every RPC server and RPC client is linked with a copy of the RPC run-time. run-time operations perform tasks such as controlling communications between clients and servers and finding servers for clients on request. The client stub and server stub exchange arguments through the RPC run-time. The client stub transmits the call to the server stub using the run-time. The server run-time receives the calls and dispatches it to the appropriate server stub. After the call has been processed by the server, the results are passed back through the run-time to the client stub. The stubs and RPC application code use the DCE RPC run-time application programming interface (API) supported by the run-time.

1.6 DCE Threads

DCE Threads is a user-level threads library based on the pthreads interface specified by POSIX in their 1003.4a standard (Draft 4). It consists of an API that gives programmers the ability to create and manipulate threads. The other technology components of DCE depend on the availability of threads. DCE Threads is provided for use on operating systems that do not provide threads already; if a threads package is already available, then DCE Threads may not be needed. The threads package that ships with OpenEdition and LE/370 is based on POSIX 1003.4a Draft 6 standard.

A thread is a division of a program into multiple parts (threads) that execute concurrently. It is a single sequential flow of control within a program. It is the active execution of a designated routine, including any nested routine invocations. Within a single thread, there is a single point of execution. Most traditional programs consist of a single thread.

Each thread has its own thread identifier, scheduling policy and priority, errno value, thread-specific data bindings, and the required system resources to support a flow of control.

DCE Threads can be used as is, as a user-level threading facility, or it can be mapped to an existing threads facility provided by the host operating system.
DCE Threads is designed for compatibility with existing operating systems that deal with processes rather than with threads and libraries that are not reentrant (that is, not written to handle multiple threads executing within them at the same time).

The distributed application programmers can use threads to help structure a program. Having multiple threads of control can introduce a higher level of complexity than programming with a single thread of control. Threads must be managed, scheduled, and allowed to communicate with one another in a controlled manner.

1.6.1.1 Threads Management
Programming with threads is similar to programming with multiple TCB in the same address space. In fact the MVS implementation of threads is based on TCBs. As with programming with multiple TCB, when there is more than one thread of control, the threads must be created and destroyed explicitly. OpenEdition Threads provides the facilities for these tasks.

1.6.1.2 Thread Scheduling
Thread scheduling is not supported by OpenEdition. The default thread scheduling policy is first in first out.

1.6.1.3 Threads Communication and Synchronization
Threads communicate through shared variables: one thread sets a variable that another thread later reads. If multiple threads are accessing the same variable, incorrect results can occur because of the scheduling of threads and the race conditions. Therefore, access to shared variables must be synchronized. DCE Threads provides three facilities for synchronizing threads within a process:
- Mutual exclusion objects (mutexes)
- Condition variable
- The join routine

1.6.1.4 DCE Threads Exceptions
DCE Threads provides two ways to obtain information about the results of a threads call. One way is specified by the POSIX 1003.4a (pthreads) draft standard: status values are returned to the thread. DCE Threads also gives the programmer an alternative to status values: an exception-returning interface, which is an extension to the basic POSIX functionality. Exceptions enable routines ignore status returns when other parts of the program are handling errors.
1.7 Distributed Services

1.7.1 Directory Service

Distributed systems may contain many resources, each of them described by a number of attributes. Resources and their attributes are constantly changing. Trying to keep track of these changes may be very difficult. DCE provides a solution to this problem by its directory service component.

DCE Directory Services consist of:

- DCE Cell Directory Services (CDS)
- DCE Global Directory Services (GDS)
- DCE Global Directory Agent (GDA)

CDS stores names and attributes of resources located in a DCE cell. GDS controls names outside of the local cell. GDA is the intermediary used for inter-cell communication between different CDSs. (note that GDS and GDA are not supported on MVS at this time).

How are resources identified within a DCE? Each resource (file, printer and so on) has a name or identifier that is unique. The name stays constant as things change throughout the distributed environment. A resource’s network location can change as the environment grows and changes with the business needs. The purpose of the directory service is to dynamically find the network address of objects.

The name that makes a resource unique within a cell is called its “local name.” Local names refer to entries located within a certain cell. Each cell has a unique name within the global network. The name that makes a resource unique within a distributed computing environment is called its “global name.” The global name is the concatenation of the cell name and the local name.

The most common use of the directory is through the use of the RPC. That is, when a programmer issues an RPC, DCE in many cases, automatically calls directory services to find the location of the requested procedure. In other cases, the programmer has to call directory services directly to find the needed procedure. In any case, the main purpose of the directory service is to find the network address of a server.

1.7.2 Security Services

The DCE security server provides a means whereby both the client and server can mutually identify each other and thus address the threat of impersonation. Access control facilities provided by DCE address concern of users snooping around shared systems. It also provides a means of encrypting part or all of the messages passed through the network. By encrypting control information associated with messages, DCE can assure the accuracy or integrity of data. Any changes to the data can be detected. This addresses the forgery issue. By encrypting the entire message, DCE addresses the eavesdropping problem. The implementation of DCE security allows most of these services to be activated with very minimal effort by the end user, programmer, and administrator. Good security is invisible to the authorized and impenetrable to the unauthorized.

Some new terms have to be introduced before the general concepts of DCE security can be discussed. A “principal” is any entity in a cell that can enter
into authentication negotiations to prove its identity to another. Examples of principals are humans, servers, or computers. Principals in a cell are represented by entries in the registry database.

A set of principals identified by a name is called a group. Groups are entries in the registry database. A group object in the registry database contains a list of principal identifiers, all of which are called “members” of that group. Groups may be associated with access permissions on objects.

Another domain within the registry database contains entries for “organizations.” Entries for organizations consist of a list of principals that are “members” of that organization. Unlike principals or groups, organizations are not used to express access permissions on objects but they are used for holding a set of policies and defaults. Examples may be password policies or default life span for accounts created in that organization.

The main services of DCE security services are

- Authentication
- Authorization
- Data integrity
- Data privacy

The authentication service enables two principals (that is a user, computer, or server) on different machines to be certain of one another’s identity. Unidentified users may eventually get access to some services, but DCE provides the functionality to block unidentified users from certain accesses. Principals have to prove their identity, to make sure they really are who they claims to be (this is done using a shared secret, his password). Authentication services are provided by the third-party authentication server that is part of the security server. Integral to making a third-party authentication secure is the distribution of secret keys through an insecure network. Massachusetts Institute of Technology (MIT) project Athena (sponsored by IBM and DEC) developed a product called Kerberos that solves the key distribution problem. It is Kerberos that makes up the authentication part of DCE security.

Authorization services are provided by the privilege server. The access to resources and functions may be restricted or allowed. The privilege service was developed by Hewlett-Packard and provides the privileges and group associations for a authenticated user.

For data privacy and data integrity, there are mechanisms in place to allow detection of tampered messages, or messages not originating from the expected caller. It may also be possible to make messages unreadable for everybody except the addressee.

The registry database, managed by the registry service, contains information associated with a principal, including encryption keys that can be used for the encryption of messages.

Control of the security services - authentication, authorization, data integrity, and data privacy - is through the RPC. When the client first calls the server, a dialogue takes place that determines the level of security that will be used.

Beside the features mentioned above, DCE security services include the following facilities:
DCE Access Control Lists (ACLs) contain information about authorization for access to a given resource such as files, entries in the directory service, and entries in the security service. When a user logs in to DCE, the Login Facility initializes a user’s DCE security environment after authenticating the user to the security service.

### 1.7.3 Time Services

In a single-system environment, there is only one clock providing time services for the computing system. Although time provided by this clock may not be completely accurate, there is only one notion of time. In a distributed system, consisting of a number of computers, each node has its own clock. Because clocks cannot be continuously synchronized, there will always be some discrepancy in their ideas of what the “correct time” is. This usually will cause problems, because in a distributed environment there maybe events that have to run on different machines in a strictly predetermined sequence.

In addition to having multiple clocks synchronized, it would also be nice if they provided the correct time. Distributed time service (DTS) maximizes the probability that a local system’s time closely matches the correct time. Closely associated with correctness is monotonicity. Clocks should run in only one direction. DTS provides unidirectional clock adjustment so that clocks never have to be run backwards to adjust for drift.

If these machines do not have a common understanding of time, the results of work may be unpredictable. DCE distributed time services provide a solution to these problems.

The following components make up the DTS:

- Time clerk
- Local time server
- Global time server
- Courier time server

Time clerks run on client machines. They request times from time servers in order to keep their “local time” synchronized. For this reason time clerks query various time servers for their time of day, calculate the “correct time” and its inaccuracy based on the answers they receive, and update their own machine’s system time.

The number of time servers in a DCE cell may vary, but three is the practical minimum. As time clerks query time servers to get the actual idea of time, time servers query one another in order to adjust their own clocks. There are different types of time servers: local time servers are servers located on a LAN (in this context, LAN means a collection of computers that are relatively close to one another in terms of communication delays). Global time servers are servers in other LANs (but within the same cell) that maybe contacted either by time clerks or by local time servers if additional time information is needed. Usually there are external time providers connected to global time servers. It is the task of external time providers to deliver the “correct time” to the cell. DTS uses the universal coordinated time (UTC) as a standard for time reference. The external time provider might be an atomic clock or some other hardware devices. In the case of MVS the sysplex timer may be used.
A courier time server is a special (local) time server that maintains cell-wide synchronization by gathering time information from global time servers from other LANs as well as from local servers on its own LAN. This is how time gets propagated from one cell to another.

The value of the time server is less obvious than that of the security and directory services but it is essential. The following is a brief list of its uses:

- The security server uses time in several ways. The conversation key used between client and server have a finite life time, typically several hours. Authenticators, a type of message used in the authentication process, have a time stamp that protects messages from being replayed.
- Universally Unique IDentifier (UUID) used among other purposes to uniquely identify the interface between client and server, is partially based on time.
- make is a UNIX utility for building executable programs. If make detects a source file newer than the time in the current binding, it will automatically recompile the source for inclusion in the program.
- cron is a UNIX utility that allows the scheduling of programs for periodic execution. Heavily used by system administrators, it depends on a reliable time source.
- Some recovery analysis routines depend on the sequence of events, which is know only through time stamps.
- In the CDS namespace, a timestamp accompanies each entry. The timestamp is used for propagation to replicas and for terminating temporary entries. CDS relies on DTS to maintain clock synchronization in the network so that the timestamps are accurate.

1.7.4 Distributed File System

Distributed File Service (DFS) is a part of DCE that provides an integrated file system, that may be shared among all DCE users and hosts. It is “integrated” because it joins the local file systems of several DFS server machines and it will be “shared” because all DFS clients within a DCE cell may access data stored in this system.

Let us suppose that a user on system A needs to access a file that resides on system B and that these computers are connected by a network. In an environment without DFS, there are two possibilities:

- The user on system A logs on system B (using a remote login feature such as Telnet in TCP/IP) and works on the file on system B.
- The user on system A transfers the requested file from system B to system A (using a file transfer application such as FTP in TCP/IP) and works on the file on system A.

In an environment with DFS this looks different.

The user specifies only the name of the requested file and continues working on system A as though the file were local. The result is that DFS provides transparency of data access to users - remote files appear like local files.

Now, let’s have a look at DFS data organization. Files and directories are stored in a unit called a “fileset.” A fileset is a (logical) grouping of files and corresponding directories for the purpose of common administration. Files are organized by directories into a hierarchical tree structure. “Aggregate” is a
term for physical data organization. It describes a certain unit of disk space that is used for storing one or more filesets.

The process of accessing a file managed by DFS is as follows:

1. After being invoked by another DCE component (for example, an application server), the DFS client asks the CDS server for the address of the fileset location server. The fileset location server is a special directory service that keeps track of the filesets and where they can be found on file server machines. The address is stored in the DFS client’s cache for later use.

2. The DFS client calls the fileset location server (using RPC) in order to get the address of the file server that keeps the requested file.

3. By invoking another RPC, the DFS client sends a request to open the file to the file server. After verifying authorization, the requested file can be accessed by the DFS Client. The DFS client caches the data locally.

DFS provides several features that provide good performance and high availability. Fast response is achieved by:

- Keeping local copies of files on DFS client machines. These copies can be used for further data accesses instead of the original files located on DFS file servers. That means less network traffic and reduced server loads.
- Caching addresses of file location server machines for later use.
- Using multi-threading on DFS server machines in order to handle multiple requests efficiently.
- Transporting large amount of data by using RPC pipes.
- Replicating files on different DFS server machines leads to efficient load-balancing.

High availability is achieved by:

- Backing up files on a special server called DFS backup server, which schedules file system backups and makes incremental backups based on filesets.
- Replication of files on more than one File Server machine. Only the original copy of a file can be updated; replicas of a file are read-only.
- Fast recovery after failures. The physical file system of DFS, called the DCE local file system (LFS), keeps a record of actions taken that affect the file system structure.
- Having local copies of DFS files on DFS client machines. If a client is disconnected from the network, a user may be able to access a file since a copy may reside in the local cache.
1.7.5 DCE Cell

A group of DCE machines that work together and are administered as a unit is called a cell. A cell is the fundamental unit for configuration and administration in DCE. A cell consists of a network connecting three kinds of nodes:

- DCE user machines
- DCE administrator machines
- DCE server machines

DCE user machines are DCE general-purpose machines. They contain software that enables them to act as clients to all other DCE services. Software running on a DCE user machine is basic for the other kinds of machines also. These services are CDS clerk, DTS clerk, security clerk, and DCE run-time library.

DCE administrator machines contain software that enables a DCE administrator to manage DCE services. A DCE administrator has a job function similar to that of an MVS system programmer. The administrator is responsible for configuring and maintaining the distributed computing environment.

DCE server machines are equipped with special software enabling them to provide one or more of the DCE services as CDS, DTS, or Security services.

Every cell must have at least the following servers in order to function:

- One (or more) cell directory service (CDS) server(s)
- One (or more) security server(s)
- Three (or more) distributed time service (DTS) servers

Note that there can be more than one DCE server running on a DCE server machine and that the servers necessary to run a DCE cell can be distributed over several machines within the cell. Other DCE servers may be present in a given DCE cell to provide additional functionality. DCE assumes that all nodes (hosts) participating in the DCE environment are physically connected by a highly available network (LAN, WAN, or a combination of both). The DCE architecture supports different types of network protocol families. The OSF DCE 1.0.2 implementation runs over the internet protocol (IP) family but OpenEdition Release 2 supports only UDP.

A DCE environment consists of one or more DCE cells that can communicate with each other. A cell becomes part of a DCE environment when it obtains access to one or more global directory services in which the other cells in the environment are registered.
OpenEdition is the MVS facility for providing open and client/server functionality. It is being delivered to customers in several releases of MVS starting in 1994 and extending into the future. It implements a number of the open standards that are developing throughout the IT industry.

MVS/ESA OpenEdition Release 1, available in the first quarter of 1994, implements some of the POSIX standards. OpenEdition Release 2 became available in June 1994. DCE Base Services are part of OpenEdition Release 2 but were delivered at the end of 1994. DCE Base Services provide the application development features, the run-time features, and the client portion of DCE services. It is very likely that IBM’s will provide CDS, security, and DFS servers at a later time.

DCE is built from technology gathered from the UNIX world. POSIX (“Portable Operating System Interface based on UNIX”) is a group of standards being developed by the IEEE. These standards define a vendor-neutral, open interface to an operating system. Many of the interfaces are UNIX based and many are still being developed and approved. DCE is making a concerted effort to comply with the POSIX standards as they are approved.

The bottom boxes on Figure 7 outline the POSIX functions provided by OpenEdition. POSIX 1003.1 provides the base operating systems services such
as process, file system, and communications. POSIX 1003.2 defines over 180
commands and utilities. POSIX 1003.2a is an extension to 1003.2. OE Release 1
implements only a subset of POSIX 1003.2a. POSIX 1003.2 and 2a commands
and utilities are very important to the DCE environment. They are primarily used
by administrators and application developers.

POSIX 1003.4a is the standard on which DCE threads are based. Recently POSIX
1003.4a was renamed POSIX 1003.1c and approved. There is no need to include
DCE threads on MVS since they have already been integrated with OE Release
2. Notice the DCE logo in Figure 7 on page 21. We have dropped the bottom
two boxes. This is because OpenEdition Release 1 and Release 2 have
implemented these functions. This is, of course, part of the OpenEdition
strategy. By providing a platform that implements international standard
interfaces, applications and products can be ported to MVS easily. DCE is one of
the first products to take advantage of this capability.

DCE is currently based on the usage of the C programming language. Therefore,
an appropriate C compiler is required. IBM AD/Cycle C/370 implements the
ANSI C standard and ISO C standard and is compatible with the DCE
requirements. Other commercially available C compliers are expected to
function in the OE DCE environment as well. As we have indicated in Figure 7
on page 21, IBM AD/Cycle Language Environment/370 has been extended to
include the POSIX functions. OpenEdition Release 2 will include IBM AD/Cycle
Language Environment/370 (C programming language only) and DCE base
services as non-priced features.

OpenEdition MVS provides a POSIX 1003.1 defined file system called the
hierarchical file system (HFS). The HFS is a set of files contained within a
hierarchy of directories, much like the DOS and OS/2 file system. A file within a
hierarchical file system directory is called an HFS file. All HFS files belong to a
directory. Each directory belongs to a directory above it in the file system
hierarchy. The highest level of the hierarchy is the root directory. HFS files can
belong directly to the root directory.

HFS file data is byte oriented, unlike most MVS data sets, which are record
oriented. Input and output for HFS files is typically performed through use of a
data stream. Despite the differences between them, you can copy HFS files into
MVS data sets and MVS data sets into HFS files using special TSO/E COPY
commands. You can access HFS files and manipulate file data from application
programs using standard POSIX 1003.1 defined C/370 library functions.

MVS treats the file system as a new type of data set, a hierarchical file system
data set (HFS data set). DFSMS/MVS facilities are used to manage HFS data
sets. MVS views an entire hierarchical file system as a collection of HFS data
sets. Each HFS data set is a mountable file system. The root file system, of
which there can be only one, is the first file system mounted when OpenEdition
MVS is started. Subsequent file systems can be mounted on a directory within
the root file system or on a directory within any mounted file system.

DCE uses HFS for its own internal use. Configuration files are stored in the HFS.
The CDS clerk uses HFS to back up its cache. The registry and directory service
and the application server use HFS as well.

The POSIX shell is a command interpreter that accepts all commands defined in
the POSIX 1003.2 standard. For some command requests, the shell calls other
programs called utilities. After logging on to MVS TSO/E, a user requests the
shell by entering the OMVS TSO/E command. The user then works within the shell environment until exiting or temporarily switching over to the TSO/E interface.

Application development on MVS/ESA can be done in three environments: batch, TSO/E, and OpenEdition. The batch and TSO/E are probably familiar to most readers. OpenEdition’s POSIX shell allow C compilations (using C89 command) and links (using the make command).
Chapter 2. Development Process and Interface Development

The purpose of this chapter is to provide a general understanding of the development process of distributed applications using DCE RPCs and the particulars related to the MVS/ESA environment. A brief description on what is necessary to write the code of the various parts of a distributed application will be shown. This chapter will not detail design aspects of distributed applications. Such information can be obtained from several sources found in the “Related Publications” on page xxi.

2.1 DCE Application Development Steps

OSF has developed a step by step approach to developing client/server applications.

Figure 8 divides the process into five basic phases, identified by letters A through E along the right side of the illustration. Each high-level phase consists

Figure 8. The Five Basic Steps

This section consists of a step-by-step checklist of every decision a programmer makes to develop a typical DCE application. Each set of decisions or choices is combined into one step. The combination of these steps takes you from the initial coding stages through the normal course of running the application itself. The intent is to give you a useful mental model of the overall code development process.
of a series of steps or decisions that normally occur in the development of a DCE application. The individual steps that make up a phase, are indicated by the numbers in the reference key. Each step is described in the following text.

The first phase A represents a series of events that occur before anything else in the development process occurs. This phase is the IDL file definition and compilation phase.

Almost all the steps in B through E consist of specific choices on how, or whether, various DCE library routines are called. You can consider steps B2 through E2 (phase B through E) as a walk through of the client and server code development of a DCE application.

The five basic phases of DCE application development are as follows:

A Define RPC/IDL interface: Steps A1 to A5 (client and server)
B Set-up and listen: Steps B1 to B12 (server)
C Bind to and call the server: Steps C1 to C3 (client)
D Service request(s): Steps D1 to D6 (server)
E Receive results: Steps E1 to E2 (client)

2.1.1 The DCE Application Development Steps

Following is a list of all the steps in the DCE application development process. The steps are separated into the five main phases previously described. For each phase, the location where the steps apply is indicated by a client, or server, or both within parentheses.

A Define the RPC/IDL interface (client and server)
   A1 Generate the interface UUID
   A2 Determine the interface version number
   A3 Write the IDL file
   A4 Write the ACF file (optional)
   A5 Process files with IDL compiler

B Set up and listen (server)
   B1 Define the manager entry point vectors (EPVs)
   B2 Register the object/type UUID associations with RPC run-time
   B3 Register the interface, type UUID, EPV with RPC run-time
   B4 Specify multithreadedness
   B5 Tell RPC run-time what protocol sequences to use
   B6 Request the bindings from RPC run-time
   B7 Register the authentication information with RPC run-time
   B8 Establish the server principal identity
   B9 Plan what to do when the server stops
   B10 Register the binding information with the endpoint mapper
   B11 Export the binding information to the namespace (CDS)
   B12 Listen for incoming service requests

C Bind to and call the server (client)
   C1 Import binding information from the namespace (CDS)
   C2 Annotate the binding handle for security
   C3 Start an RPC interface operation

D Service the request (server)
   D1 Wake up in the manager routine
   D2 Get the client’s privilege attribute certificate (PAC)
2.2 Application Building

In order to develop DCE applications using RPC, a series of steps must be followed. These steps involve the usage of DCE facilities that will considerably simplify writing distributed applications.

Figure 9 shows a good pictorial view of the development process steps.

Interface development involves defining an interface with one or two files and compiling these definitions with a special compiler. Compiling the interface file or files produces a header file and two C source files - client stub and server stub. A C compiler may be invoked by the interface compiler to produce the necessary client and server stub object files which are used in the client and in the server development, respectively.
Server development comprises the steps of developing subroutines, as if they were used in a local application, with the addition of setup code. The header file produced by the interface compiler must be included. Once compiled, the resulting objects are linked with the RPC run-time library and the server stub object to create the server executable code.

Client development consists of developing a program, as if it were a local application, with the eventual addition of RPC setup code. The header file produced by the interface compiler must be included. Compiling these source files generates the objects which are linked with the RPC run-time library and the client stub object to produce the client executable code.

Next we will take a closer look at the decisions that need to be made.

### 2.3 Interface Development

The interface is the core of the client/server relationship when using DCE RPC's. Client and server must adhere to a well defined interface to enable the client to pass the right parameters in the right order to the appropriate server. That is the reason why the client and the server source files must contain the header file produced by the interface compiler. This header file will contain typedefs, constant definitions, function prototypes of the remote procedures, and eventually a global variable.

Interfaces are defined with an interface definition file. This file contains statements that uniquely identify the interface, including its version. It also contains statements defining an interface name, constants data types and the remote procedures. These statements are written using **IDL - interface definition language**. This file has the extension `.idl`.

An optional extension to the interface definition is the attribute configuration file (ACF). It contains information for the interface compiler which is not included in the `.idl` file. The acf file contains the desired method to connect with the server, statements to exclude the object code for some procedures from the client stub and so on. This file uses **ACL - attribute configuration language** and has the extension `.acf`. The file name must match the name used with the `.idl` file for a given interface.

The interface compiler is known as **IDL compiler**.

The basic steps in developing an interface are:

1. Write the `.idl` file.
   - Provide the interface identification and version.
   - Provide the interface name.
   - Declare the necessary constants and data types.
   - Describe the operations based on the remote procedures prototypes.
2. If needed, code the `.acf` file.
   - Describe attributes that are interface-wide.
   - Describe attributes of the data types and operations.
3. Run the IDL compiler providing the complete `.idl` filename as input.
• Client and server stub files will be generated.
• Common header file will be generated.
• C compiler will be invoked to create the client and the server stub object files.

These steps are shown schematically in the Figure 10.

![Figure 10. Interface Development](image)

2.4 Server Development

A basic server consists of two major portions of code: the server initialization portion and the implementation of the RPC procedures, or the manager portion. The initialization portion registers the interfaces supported by the server to the RPC run-time and, among other things, starts the server listening for remote procedure calls. The manager portion contains the procedures advertised to clients. Both parts should include the header file for the interface created by the IDL compiler to provide the data structures and types needed for the connection to the client by RPC run-time.

The principal steps in developing a basic server are:

1. Write the implementation of the RPC procedures advertised by the server.
2. Write the initialization portion of the server.
   • Register the interface to the RPC run-time.
   • Select a protocol sequence for the RPC calls.
   • Advertise the server.
   • Listen for remote procedure calls.
3. Implement signal handling for a graceful exit, if necessary.
The two portions of the server code will be linked together with the server stub created by the IDL compiler and the DCE libraries to create the server executable code. This is shown schematically in Figure 11.

2.5 Client Development

This section describes the steps to develop basic client code.

A basic client should be able to retrieve and manage binding information which is used to contact the server over the network. If necessary the client should be able to identify itself to the server and to choose one server if more than one is available.

When the client selects a server it receives a binding handle which it uses for subsequent remote procedure calls. The binding contains the following:

- Interface UUID
- Interface version number
- Protocol sequence
- Network address information
- Endpoint information

This is referred to as binding a client to the server.

The phases of client development are as follows:

- Implement a binding management method and obtain a binding handle.
- Make the remote procedure calls according to your application.

The IDL compiler creates a C language header file when you compile the interface definition. This header file should always be included in the
application’s client code. The header file contains the data types and structures that are needed by the RPC run-time. It is the same header file that is included by the server part of the application.

Finally the client code is linked together with the client stub and the DCE library as shown in Figure 12.

2.6 Generating a UUID and IDL File

The first step in creating your application’s interface is creating a universal unique identifier (or Interface UUID). A UUID is a 128-bit number that is guaranteed to be unique throughout DCE. This UUID makes it possible for the DCE naming service to identify and locate your application server throughout DCE. The UUID is generated along with a template for your application’s IDL file using a utility called UUIDGen.

**In Batch**: Use the example JCL in Figure 13 on page 32 to generate a UUID for your application. In addition to generating a UUID, running this JCL will also create a template for your application’s IDL file.
Figure 13. Sample JCL to Run the UUID Generator

In TSO/E: Figure 14 shows how you would generate a UUID from the TSO/E command line.

```sh
uuidgen -i -o "//USERPREFIX.DCEMVS.IDL(MATHX)"
```

Figure 14. Sample TSO/E Command to Create a UUID for the MATHX Application.

In the Shell: Figure 15 shows how you would generate a UUID and create a template for the MATHX IDL file.

```sh
uuidgen -i -o mathx.idl
```

Figure 15. Sample Shell Command to Run the UUID Generator for MATHX.

IDL, the language used in .idl files, has a syntax very similar to C. It allows the developer to describe fundamental aspects of the interface such as:

- Names of the remote procedures
- Order and types of parameters passed to the remote procedures
- Data types of any values returned by the remote procedures
- Any special data types needed to declare the procedure parameters
- Any constants that could be used by both the program and the remote procedures
- Any context information that must be saved between procedure calls

The .idl file can be broken down in two components:

- **Interface header**
  Contains information that applies to the entire interface such as interface identifier, version number and interface name.

- **Interface body**
  Contains information pertinent to the remote procedures such as constants, data types and the set of operations (remote procedures) available through this interface.
2.6.1 Interface Header

The required information for the interface header is the UUID and the interface name.

A UUID is a 128-bit number that contains a timestamp and adapter physical address, making a UUID unique throughout DCE. Client and servers using DCE RPC only communicate if they have the same interface UUID. A UUID in an .idl file is enclosed in an interface UUID declaration:

```
uuid(0082C594-282B-1BF1-9799-10005A4F5444)
```

The interface name can have up to 17 characters and is used by the IDL compiler in identifiers it constructs for various data structures and data types. The interface name is placed after the keyword interface in an .idl file:

```
interface mathx
```

Optionally, an interface header may contain information such as version number, pointer defaults, well-known endpoints and other attributes.

**Version number** allows the developer to maintain multiple versions of the interface. This number is composed of a major number and a minor number. Clients and servers can only communicate if they have the same major number. Changes in the major number reflect a change that does not provide upward compatibility such as changing the purpose of the remote procedures and the parameters passed to these procedures. Clients and servers can only communicate if the client minor number is less than or equal to the server minor number. Changes in the minor number reflect changes that do not compromise upward compatibility such as adding data types and constants to the .idl file and adding new operations placing them at the end of the .idl file. Version number is enclosed within a version statement in an .idl file:

```
version(1.0)
```

The UUID and version number uniquely define an interface. The combination of both is called **interface spec**. The UUID declaration and the version declaration are separated by a comma (,) and contained within brackets ([ ]):

```
[ uuid(0082C594-282B-1BF1-9799-10005A4F5444),
  version(1.0)
]
```

2.6.2 Interface Body

The declarations that are part of the interface body have a C like syntax. They must be terminated with a semicolon (;). These declarations specify the constants, data types and operations to be used through the interface. The .idl file interface body must be contained within a pair of curly braces ({ }).

**Constants** are defined by using the keyword const, giving it a name, specifying the data type and assigning a value to it:

```
const short MAX_VALUES = 12;
```

**Data types** used or returned by the remote procedures must be declared in the body of the .idl file. These type definitions are used by the IDL compiler to generate the stub code necessary for marshalling and unmarshalling data of that type. The declaration of a data type consist of the keyword typedef followed by
an optional attribute enclosed in brackets, the data type specifier and the data type name:

typedef long value_array_t[ MAX_VALUES ];

Operations are the remote procedures provided by the interface. The operation declaration names the remote procedure, defines the parameters of the remote procedures in terms of their data types, directional attributes and call order and eventually informs the data type of the value returned by the remote procedure. Operation declarations are placed after constant and data type declarations and have the following elements:

- Type specifier
  - Describes the data type returned by the operation
- Operation identifier
  - Name of the remote procedure
- Parameter declaration
  - Provides the directional attributes which inform whether parameters are being passed from the client to the server [ in ], from the server to the client [ out ], or both ways [ in, out ], being that parameters with the attribute of [ in ] are values and with the attributes [ out ] and [ in, out ] are pointers
  - Defines the data types of the parameters
  - Names the parameters
  - Parameters are separated by comma (,)

Below we have an example of an operation declaration:

```c
void add (
    [ in ] value_array_t value_a,
    [ in ] long num_values,
    [ out ] long *sum
);
```

After you have run the UUIDGEN in batch, as a TSO/E command, or through the shell, you will generate a file like Figure 16.

```
[ uuid(008FB3A8-0FD7-1BFC-AB6C-10005A4F5444),
  version(1.0)
 ]
interface INTERFACENAME
{
}
```

Figure 16. Basic IDL File

Substituting INTERFACENAME with mathx and inserting the declarations above between the opening brace and the closing one, we have a basic .idl file. In this case for our MATHX example (See Figure 17 on page 35).
2.6.3 Defining the .acf File

The .acf file allows developers to specify how an RPC interface appears to local application code and how the local application interacts with the RPC interface. Its components are:

- **Interface header**

  This interface header contains information that applies to the entire interface being defined such as binding method, marshalling choices, code generation options and an interface name matching the one defined in the .idl file. An example of an .acf file interface header would be:

  ```
  [ explicit_handle
  ]
  interface mathx
  {
  }
  ```
  
  or for implicit binding it would be:

  ```
  [ implicit_handle ( handle_t global_var_h )
  ]
  interface mathx
  {
  }
  ```

- **Interface body**

  The interface body is composed of type declarations, operations and parameter lists, allowing attributes to be attached to the matching .idl file. These attributes define aspects of data conversion during marshalling and unmarshalling of data, whether marshalling is done by inline code or by calls, if the heap is to be used to store the serve parameters in place of the stack, use of explicit binding in a specific procedure, whether procedures or parameters return error status and if an individual procedure stub code is to be generated in the client stub. These declarations are terminated with a semicolon (;). The .acf file interface body must also be contained within a pair of curly braces ({ }). For example:
typedef [ in_line ] value_array_t;
[ explicit_handle ] add ();
[ nocode ] subtract ();
}

An example of a complete .acf file:
[
explicit_handle
]
interface mathx
{
    typedef [ out_of_line ] value_array_t;
    [ comm_status ] multiply ();
    [ nocode ] divide ();
}

As mentioned in Chapter 2, “Development Process and Interface Development” on page 25 the statements of an .acf file are written using attribute configuration language. The reader should refer to the bibliography for details on using attribute configuration language.
Chapter 3. Server

In this section we show how a basic server can be developed. As we mentioned before there are two main portions of server code, which we prefer to put in separate modules of code:

- Manager code
- Server Initialization code

These two main portions implement two very different tasks of the server code. The manager code implements the remote procedures which are exported to clients. The server initialization module implements the setup with the RPC run-time. This part looks pretty much the same for every application.

For this document we decided to show the different parts of the development of the initialization module in several steps. Our examples of setting up a server will not be sophisticated but they show the important steps.

We start this chapter with an example of manager code which implements a small remote procedure. This remote procedure will be used with our examples called MATHX, SBIND and later in Chapter 6, “Security” on page 77 with our security example SECUR.

3.1 Implementing Remote Procedures

Designing and implementing remote procedures is very much like designing and implementing local procedures. However there are some things which require a closer look:

- Client and server don’t share memory
- Heterogeneous systems in different locations with different operating systems
- Threads and concurrent calls

If you design remote procedures you have to look at the data you are passing to the client. We will discuss this topic more fully in Chapter 7, “Threads and Data” on page 125.

If you decide to run client and server in a heterogeneous environment you must be careful about calls to the underlaying operating system.

Be sure that your manager code is thread safe. When you plan to allow more than one concurrent call, you should insure the threads which answer the client call don’t interfere with one another.

The following example for manager code shows the implementation of a remote procedure. It adds the integer entries of a vector with numv ≤ MAX_VALUES entries and returns the result in the reference parameter total. The constant MAX_VALUES is declared in the basic.idl file.

The file MATHXM includes the header files for stdio, stdlib and the header file generated by the IDL compiler from the interface definition.
Figure 18. Basic Manager Code (mathxm)

3.2 Initializing Server Code

Before we look into an example of the server initialization portion of the server code, we take a look at the steps necessary to set up the server part. Initialization of the server splits into the following main pieces:

- Registering the interface to RPC run-time
- Selection of a protocol sequence
- Advertising the server
- Listening for RPC calls

These steps are done by calling routines from the DCE RPC library. Each step will be explained on the next few pages, but we will not talk about all options that are available.
3.3 Registering the Interface

Registering the interface informs RPC run-time it should accept and dispatch calls for the registered interface. RPC run-time can only dispatch client RPC calls to the correct server interface if it knows the interface is available on the server. Therefore registering the interface makes the remote procedures offered by the server available to requesting clients.

You can register multiple interfaces and multiple versions of the same interface from a single server. Use the `rpc_server_register_if()` call to register an interface.

Before we look in detail at the `rpc_server_register_if()` call we will talk about the parameters and the parts from RPC run-time which are involved in this call.

3.3.1 Entry Point Vector

An entry point is an address of a call entry point for a procedure. In DCE an entry point vector (EPV) is a data structure used by the server stub code to dispatch incoming remote procedure calls to the requested remote procedures provided by the manager code. The entry point vector contains a list of operations offered by the interface and their entry points.

A default manager entry point vector is generated automatically by the IDL compiler in the server stub and declared in the application header file. A server can also supply an entry point vector other than the default, to overwrite the default entry point vector and provide a list of call entry point addresses other than those generated by the IDL compiler.

![Figure 19. Entry Point Vector in Server Stub](image-url)
3.3.2 rpc_server_register_if()

The call rpc_server_register_if() is used to register the server interface to RPC run-time. The prototype for the rpc_server_register_if() call is declared in the header file <dce/rpc.h>. Below we show the definition of the call with its arguments.

```c
void rpc_server_register_if (
    rpc_if_handle_t if_handle,
    uuid_t *mgr_type_uuid,
    rpc_mgr_evp_t *mgr_epv,
    unsigned32 *status
);
```

The call arguments of rpc_server_register_if() are:

- **if_handle**
  An IDL generated data structure specifying the interface to register. It is declared in the interface header file. Its name includes the interface name and version number. Our MATHX example looks like this: `mathx_v1_0_s_ifspec`.

- **mgr_type_uuid**
  A type UUID to associate with the mgr_epv argument. Specifying a NULL argument value registers the if_handle with a NULL type UUID. A none-NULL manager type UUID indicates that this interface is dedicated to work on a specific type of object.

- **mgr_epv**
  The manager entry point vector. To use the IDL generated default entry point vector, specify a NULL value to this argument. To select a server application supplied entry point vector, pass a none-NULL address of the entry point vector whose value has been declared by the server as the value to the mgr_epv argument.

- **status**
  It returns the status code from the interface register operation. This status code is a value that indicates whether the routine completed successfully and, if not, why.

In the example for a basic server we use rpc_server_register_if() with the default entry point vector. We issued the following call to register our interface:

```c
rpc_server_register_if ( mathx_v1_0_s_ifspec, NULL, NULL, &status );
```

3.3.3 Protocol Sequences

A protocol sequence is a particular combination of RPC, transport, and network protocols, represented by a character string. It is used to indicate the RPC run-time that it should accept calls from clients that arrive using those protocols.

Each protocol sequence contains three options that together identify a single type of communication protocol. The options are:

1. An RPC protocol for communications
   Currently choices are the NCA (Network Computing Architecture) connection oriented protocol (`ncacn`) and the NCA datagram protocol (`ncadg`).

2. A network address family
Currently the choice is (ip).

3. A network transport protocol for communications

Currently only udp is available on MVS.

To receive remote procedure calls, a server must register at least one protocol sequence with RPC run-time. The call `rpc_server_use_all_protseqs()` can be used on a server to register all available protocol sequences and the call `rpc_server_use_protseq()` can be used on a server that wants to specify a protocol sequence to be used.

### 3.3.4 `rpc_server_use_all_protseqs()`

The prototype for the `rpc_server_use_all_protseqs()` call is declared in the header file `<dce/rpc.h>`. Below we show the definition of the call with its arguments:

```c
void rpc_server_use_all_protseqs ( 
    unsigned32 max_call_requests, 
    unsigned32 *status 
);
```

The call arguments for `rpc_server_use_all_protseqs()` are:

- **max_call_requests**
  - This specifies the number of concurrent remote procedure call requests the server can accept over each protocol sequence. This parameter is only meaningful for connection oriented protocol sequences (`ncacn`) in this version.

- **status**
  - This returns the status code from the `rpc_server_use_all_protseqs()` call. This status code is a value that indicates whether the routine completed successfully and, if not, why.

### 3.3.5 `rpc_server_use_protseq()`

The prototype for the `rpc_server_use_protseq()` call is declared in the header file `<dce/rpc.h>`. Below we show the definition of the call with its arguments:

```c
void rpc_server_use_protseq ( 
    unsigned_char_t *protseq, 
    unsigned32 max_call_requests, 
    unsigned32 *status 
);
```

The call arguments `max_call_requests` and `status` are the same used in the `rpc_server_use_all_protseqs()` function.

The call argument `protseq` specifies a string identifier for the protocol sequence to register with RPC run-time. Current valid protocol sequences is:

- `ncadg_ip_udp` for datagram protocol using udp.

In the example for our MATHX server we use either `rpc_server_use_protseq()` with the constant `MAX_CONC_CALLS_PROTSEQ` declared in the `.idl` file.

or we issued the following call to select udp as a protocol sequence:

```c
rpc_server_use_protseq ("ncadg_ip_udp", MAX_CONC_CALLS_PROTSEQ, &status );
```
3.3.6 rpc_server_use_protseq_ep()

The prototype for the rpc_server_use_protseq_ep() call is declared in the header file <dce/rpc.h>. Below we show the definition of the call with its arguments:

```c
void rpc_server_use_all_protseqs_ep(
    unsigned_char_t *protseq,
    unsigned32 max_call_requests,
    unsigned_char_t *endpoint,
    unsigned32 *status);
```

The call arguments max_call_requests and status are the same used in the rpc_server_use_all_protseqs() function.

The call argument protseq() specifies a string identifier for the protocol sequence to register with RPC run-time. Current valid protocol sequence is:

- ncadg_ip_udp for datagram protocol using udp.

3.4 Advertising the Server

A server application advertises itself to make binding information available to clients. Client applications need the binding information to be able to connect the server application. There are two basic methods available to advertise a server application:

- Server can use the DCE cell directory service.

  This is the usual way to advertise a server.

  The MATHX and SECUR examples use explicit binding. Servers use the cell directory service to advertise themselves to clients; clients use it to find servers. The cell directory service provides the necessary binding information and prevents the client from having to deal with the details of locating binding information.

- Server can use string binding.

  A string binding is a string of text that provides all necessary binding information.

  The SBIND example is a string binding example. The server can generate a string binding and write it to a well-known file, or simply print the string binding on the screen.

  String binding is frequently used during application development to create a prototype distributed application without using the cell directory service.

3.4.1 Endpoints

An endpoint is a transport layer address assigned to the server process. You can consider an endpoint similar to a socket port in TCP/IP or a LU in SNA. The endpoint address is specific to the transport protocol the server will use.

DCE RPC deals with two types of endpoints:

- **Well-known endpoints**

  When the application uses a specific endpoint you can specify the endpoint you want to use in the IDL code that defines the client/server interface.
Well-known endpoints are not usually used because they must be generally allocated by a central network authority and they are relatively scarce.

- **Dynamically assigned endpoints**

  RPC run-time usually assigns endpoints to the server dynamically, as protocol sequences are registered. If you are not using a well-known endpoint, the RPC run-time assigns an available endpoint to your interface at the time of protocol selection. Because of its dynamic nature the endpoint assigned to the server will neither be stored in the interface definition nor in the CDS name space database. The information about the endpoint assigned by the RPC run-time is stored in the endpoint map database on the RPC server system.

### 3.4.2 Endpoint Mapper Daemon, rpcd

The endpoint map associates endpoints with other binding information. Clients who import binding information from the CDS namespace start out with partial bindings when they try to contact their server for the first time. The partial binding will get them only as far as the server endpoint mapper daemon, rpcd.

To find the valid endpoint, the client interface definition and binding information are compared to information in the endpoint map. When the endpoint of an appropriate server is obtained, the resulting fully binding handle is used to complete the connection at that endpoint.

The rpc daemon process maintains the endpoint map for the RPC server host system. A server must register the endpoint assigned by the RPC run-time with the endpoint mapper daemon rpcd to let him know what endpoints belong to the server, so that he can fill in the partial bindings as they arrive and route the incoming RPCs to the proper server process for the first time. Subsequent remote procedure calls executed with the same binding information will go straight to the server, since the binding is now complete.

### 3.4.3 Steps to Advertise the Server

If the server uses the CDS namespace and dynamically assigned endpoints it takes three steps to advertise the server to clients:

1. From the RPC run-time get the information on what binding handles and endpoints are associated with the server. This information is returned by the \texttt{rpc_server_inqBindings()} call.

2. To register the server dynamically created endpoint with the host’s local endpoint mapper daemon, rpcd, use the \texttt{rpc_ep_register()} call.

3. Export an entry for the server into the CDS namespace. The call \texttt{rpc_ns_binding_export()} places an entry into CDS with the server host information.

### 3.4.4 rpc_server_inq_bindings()

A server calls the \texttt{rpc_server_inq_bindings()} routine to obtain a vector of binding handles. These binding handles are created by selecting protocol sequences, for example the \texttt{rpc_server_use_all_protseqs()} call.

The prototype for the \texttt{rpc_server_inq_bindings()} call is declared in the header file \texttt{<dce/rpc.h>}. Below we show the definition of the call with its arguments.
void rpc_server_inq_bindings(
    rpc_binding_vector_t **binding_vec,
    unsigned32 *status
);

The call arguments for rpc_server_inq_bindings() are:

binding_vector
This returns the address of the vector of binding handles.

status
This returns the status code from the rpc_server_inq_bindings() call. This status code is a value that indicates whether the routine completed successfully and, if not, why.

We issued the following call to inquire about the available bindings:
 rpc_server_inq_bindings(&bind_vector_p, &status);

3.4.5 rpc_ep_register()

If the server uses dynamically assigned endpoints, then you must use the rpc_ep_register() call to register the server endpoint to the local host’s endpoint map database. This is not necessary if the server uses a well-known endpoint.

The rpc_ep_register() routine adds or replaces entries in the local host’s endpoint map database. For an existing database entry that matches the provided interface specification, binding handle and object UUID, this routine replaces the entry’s endpoint with the endpoint in the provided binding handle.

The prototype for the rpc_ep_register() call is declared in the header file <dce/rpc.h>. Below we show the definition of the call with its arguments:

void rpc_ep_register(
    rpc_if_handle_t if_handle,
    rpc_binding_vector_t *binding_vec,
    uuid_vector_t *object_uuid_vec,
    unsigned_char_t *annotation,
    unsigned32 *status
);

The call arguments for rpc_server_inq_bindings() are:

if_handle
An interface handle to register with the local endpoint map.

binding_vec
A vector of binding handles over which the server can receive remote procedure calls.

object_uuid_vec
A vector of object UUIDs offered by the server or a NULL argument. This vector is constructed by the server application. A NULL argument indicates there are no object UUIDs to register.

annotation
A character string comment applied to each cross-product element added to the local endpoint map database. The string can be up to 64 characters long, including the \0 terminator.
status

This returns the status code from the `rpc_ep_register()` call. This status code is a value that indicates whether the routine completed successfully and, if not, why.

We issued the following call to register the endpoints with the RPC run-time.

```c
rpc_ep_register( mathx_v1_0_s_ifspec, bind_vector_p, NULL,
    ( unsigned_char_t * ) "mathx server, version 1.0", &status );
```

### 3.4.6 `rpc_ns_binding_export()`

The `rpc_ns_binding_export()` call exports an interface and the binding handles of the server to the CDS namespace. If this information is exported to the namespace a client can lookup the namespace to get binding information to a server that matches his interface specification.

If the exported entry name does not exist in the CDS namespace the `rpc_ns_binding_export()` tries to create it.

A server is not required to export its interfaces to the CDS namespace. But when a server does not export its interfaces, only clients that privately know of the server binding handles can access its interfaces.

The prototype for the `rpc_ns_binding_export()` call is declared in the header file `<dce/rpc.h>`. Below we show the definition of the call with its arguments.

```c
void rpc_ns_binding_export ( unsigned32 entry_name_syntax,
    unsigned_char_t *entry_name,
    rpc_if_handle_t if_handle,
    rpc_binding_vector_t *binding_vec,
    uuid_vector_t *object_uuid_vec,
    unsigned32 *status
    );
```

The call arguments for `rpc_ns_binding_export()` are:

- `entry_name_syntax`
  - The integer argument `entry_name_syntax` specifies the syntax of the following argument `entry_name`. There are several different syntax types available, depending on how you specify the `entry_name`. Valid syntax types are X.500 style and DNS name style. There are some constants defined in the `<dce/rpcbase.h>` header file which is included by the `<dce/rpc.h>` header file. Specify the `rpc_c_ns_syntax_default` constant which establishes the syntax the RPC run-time is using to interpret CDS namespace entries.

- `entry_name`
  - This argument represents the name to which binding information is exported according to the syntax specified by the `entry_name_syntax` argument.

- `if_handle`
  - The server interface handle to export into the CDS namespace.

- `binding_vec`
  - A vector of bindings to export or a NULL value. A Null argument value indicates that there are no binding handles to export.
object_uuid_vec

A vector of object UUIDs offered by the server or a NULL value. A NULL argument value indicates there are no object UUIDs to export.

status

This returns the status code from the rpc_ns_binding_export() call. This status code is a value that indicates whether the routine completed successfully and, if not, why.

We issued the following call to export the binding information into the CDS namespace.

rpc_ns_binding_export ( rpc_c_ns_syntax_dce, ENTRY_NAME, mathx_v1_0_s_ifspec, bind_vector_p, NULL, &status );

3.5 Listening for RPC Calls

After all initialization tasks are done, all that is left is to ask the RPC run-time to listen for requests coming from client applications that wish to use the server. This will be initiated by the rpc_server_listen() call. This call only returns if the server is terminating.

The prototype for the rpc_server_listen() call is declared in the header file <dce/rpc.h>. Below we show the definition of the call with its arguments:

```c
void rpc_server_listen ( 
    unsigned32 max_calls_exec, 
    unsigned32 *status
);
```

The call arguments for rpc_server_listen() are:

max_calls_exec

This number indicates how many concurrent remote procedure calls the server can execute. It implies the number of threads the server can create to handle incoming client requests.

Setting max_calls_exec to 1 indicates this server handles only one request at a time. This is useful if the implementation of the remote procedures is not thread safe.

status

This returns the status code from the rpc_server_listen() call. This status code is a value that indicates whether the routine completed successfully and, if not, why.

Our MATHX server example uses the following call to indicate the RPC run-time to listen for RPC calls from clients.

rpc_server_listen ( MAX_CONC_CALLS_TOTAL, &status );
3.6 Error Handling

Most of the RPC run-time routines we use to set up the server code have a status reference argument to determine whether the routine executed successfully or not. This variable should be checked after each call and the server should terminate its execution if a call fails.

In the MATHX server example this is done by a pseudo function ERRCHK() which prints out the module name, the line where the error occurs, the DCE error text and exits the server.

Here is the definition of the pseudo function ERRCHK() from the header file ERRCHK:

```c
#ifndef _ERRCHK_H
#define _ERRCHK_H

char dce_error_string[256];
error_status_t error_inq_st;

#define ERRCHK(x)
if ( (x) != error_status_ok ) {
  fprintf( stderr, "\%s: error in \%s[d]:\n", argv[0], __FILE__, __LINE__ - 1 );
  dce_error_inq_text(x, dce_error_string, &error_inq_st);
  fprintf(stderr, "\%s:\n", dce_error_string); fflush(stderr);
  exit ( x );
}
#endif
```

Figure 20. Basic Error Check (ERRCHK.h)

After the server is set up to listen to incoming client calls the rpc_server_listen() does not return unless the server is stopped from listening. Any error which occurs during stub code or remote procedure call execution is indicated as an exception. In our MATHX server example we do not handle any exceptions, which means that our server will exit abruptly if an exception occurs.

3.7 Example of Server Initialization Code

In Appendix A, “JCL and Source Listings for MATHX and SBIND Examples” on page 197 we provide full sample code for the MATHX server initialization part. This will be our skeleton on which other examples will be based.

The servers in this book register their interface under a variety of different directories. In order to run the servers, you must create a directory entry in the CDS namespace. You can use the following command to create an entry for the server directory.

cdscp create dir /.:/Servers
This server initialization module has only minor error handling implemented and there is no code to catch signals. Therefore, if any error occurs or if the server receives the signal to stop its execution it only exits without unregistering anything. This means that all binding information in the CDS namespace and the entry in the entry point vector still remain, but are no longer valid. A client which makes a call to a procedure offered by this server will have problems with the invalid binding information. If the client is using automatic binding the RPC run-time gets a timeout and tries to rebind to another server. This could cause serious performance problems. A server with other binding methods will probably dump core after receiving the response timeout.

We recommend removing the server entry in the CDS namespace after running this server with the command:

cdscp delete object /.:/Servers/Math

During our testing of ECIP 2 code, we found that the client and server stubs were not compiled automatically as they were with ECIP 1. Until this is changed it is necessary to compile the stubs separately. These modules together with the server stub object module and the DCE library must be linked together for obtaining the executable server.

The sample MATHX server (mathxs) can be found in Figure 112 on page 204
In this section, the development of a basic client that uses the server seen in Figure 112 on page 204 is described.

The first consideration in developing this client is to figure out how it will reach the server. This might seem like a complex problem but actually, with DCE it isn’t. There are 3 ways in which the client can bind to the server:

- Explicit binding
- Implicit binding
- Automatic binding

The MATHX and SECUR examples supplied in this book will use the explicit binding management. The SBIND example will also use explicit binding management but will use string binding to obtain binding information. Bindings will be explained later in this chapter.

The client code makes the function call to your server. Normally the client obtains the binding information for the server from the CDS namespace. Communications with the server is established by receiving the correct protocol sequence, binding handles, and so on from the DCE run-time. Then your client makes the function call, passing along any required input parameters. Once your client receives the results of the function call from the server, it may process the parameters received or may make additional function calls before ending.
The client code we will be using for our MATHX and SBIND examples look like this:

```c
#include <stdio.h>
#include <stdlib.h>
#include <dce/rpc.h>
#include "errchk.h"
#include "mathx.h"

int main ( int argc, char *argv[] )
{
    long int i, numv, sum;
    value_array_t va;
    /* Check if the user passed the minimum number of parameters. */
    if ( argc < 3 ) {
        printf ( "Usage : %s <values to be added>
        exit ( 1 );
    }
    /* Get values from the command line and load them in array va. */
    for ( i = 0; i < MAX_VALUES && i < argc - 1; i++ )
        va[i] = atoi ( argv[i + 1] );
    numv = i;
    /* Call add passing handle, values, number of values and pointer to result. */
    add ( bh, va, ( long )numv, &sum );
    printf ( "The sum is %d
    }

Figure 21. Basic Client Code (mathxc)

4.1 Binding

To communicate with a server, the client must obtain the server binding information. The binding information consists of data necessary to establish a connection with the server such as:

- Protocol Sequence
  A protocol sequence string. Protocol sequences were explained in section 3.3.3, “Protocol Sequences” on page 40.

- Host Address
  The address of the machine where the server resides. It uses the network address convention defined in the protocol sequence.

- Endpoint
The network address of the server process. Endpoints were explained in section 3.4.1, “Endpoints” on page 42.

- Object UUID

  An identification of an object if object UUIDs are being used, otherwise it is nil.

This information is contained in a data structure and is referenced by a binding handle.

To obtain binding information the client can use two sources:

- CDS (cell directory service)
  
The servers use the CDS as a repository, placing the binding information there so that clients with proper authority can get it.

- String binding
  
  This source implies that the client has some means of getting the binding information in string format and converting it to the appropriate format, and referencing it with a binding handle.

  An example of string binding information could be:
  
  ncadg_ip_udp:129.35.27.87[1976]

Clients can manage binding information of a server by using one or more of these binding handle management methods (See Figure 22).

- Explicit binding
  
  The client code obtains the binding handle for a specific server and uses the handle on each remote procedure call as a parameter.

- Implicit binding
  
  The client code obtains the binding handle for a specific server and stores it in a global binding handle which is used for all subsequent remote procedure calls.

- Automatic binding
  
  The client stub automatically obtains a binding handle that will be used for all subsequent remote procedure calls. This method requires CDS.

Which binding method you choose depends on how you design your application. If the client talks to several different servers in the same part of the application, the explicit binding method must be chosen because you have to specify the server binding handle in each call. Choose implicit or automatic binding methods when converting a local application to a distributed one. Using these methods minimizes the amount of work that must be done to convert the application. It should be noted that a client that uses implicit or automatic binding may make use of explicit binding for individual calls that need to reach specific servers. The choice of binding method does not affect the server setup code. It affects the client code and to a lesser degree the server manager code. The choice on the source of binding information affects server setup and client code since it defines how the server will convey its binding information to the client.

To obtain the binding handle the client calls routines from the DCE library. After the client obtains the binding handle the call to remote procedures can be made
just like local calls, except that for explicit binding the binding handle is required as a parameter for each call.

In the next sections the required code to implement these binding methods will be seen in detail.

Figure 22. Binding Management Methods
4.1.1 Running an Application Using Explicit Binding Management

Figure 23 shows schematically how server and client are working together during the execution of an application that uses explicit binding and CDS.

The client imports binding information from the CDS, obtaining a binding handle. This binding handle is used in the subsequent procedure call to bind to the appropriate server. After the client wakes up from the remote procedure call, a cleanup of the context used for importing information is done before ending.

Figure 23. Running an Application Using Explicit Binding
4.1.2 Implementing Explicit Binding Using CDS

Explicit binding requires that a binding handle be passed from the client to the stub as an operation parameter, enabling the client to address a specific server in a call. The code responsible for managing the binding handle resides in the client setup. The client and server manager code will have to be modified to include the additional binding handle parameter.

These are the steps to write a distributed application using explicit binding:

1. Use a handle_type as the first parameter of the operation declaration in the .idl file. This informs the IDL Compiler that explicit binding will be used:

```idl
interface mathx
{
    const short MAX_VALUES = 12;
    typedef long value_array_t[ MAX_VALUES ];
    void add(
        [ in ] handle_t binding_h;
        [ in ] value_array_t value_a,
        [ in ] long num_values,
        [ out ] long *sum
    );
}
```

*Figure 24. Explicit Binding IDL File*

2. Optionally, the binding method may be specified in a declaration within the .acf file. This information is optional because the IDL Compiler is already aware of the binding method being used from the .idl file:

```acf
[ explicit_handle
]
interface mathx
{
}
```

If an .acf file is used, it is good practice to add it as a dependency of the client and server stubs and of the header file generated by the IDL compiler.

3. Create the client setup code that will obtain the binding information for a selected server from CDS. A binding handle will be associated with the binding information.

4. Include the binding handle as the first argument when calling the remote procedures in the client program:

```c
/* Call add passing handle, values, number of values and pointer to result. */
add ( bh, va, ( long )numv, &sum );
```

5. Include the binding handle as the first parameter of the remote procedures in the server manager code:
```c
void
add (  
    rpc_binding_handle_t bh,
    value_array_t value_a,
    long numv,
    long *total )

This is essentially the only change required on the server side.

The following routines can be used for the client setup code:

4.1.2.1 rpc_ns_binding_import_begin ()
This function creates a context in which binding handles for servers can be imported.

The prototype for the rpc_ns_binding_import_begin () call is declared in the header file <dce/rpc.h>. Below we show the definition of the call with its arguments:

```c
rpc_ns_binding_import_begin (  
    unsigned32 entry_name_syntax,
    unsigned_char_t entry_name,
    rpc_if_handle_t if_spec,
    uuid_t object_uuid,
    rpc_ns_handle_t *import_context,
    unsigned32 *status)
```;

The call arguments for rpc_ns_binding_import_begin () are:

```c
entry_name_syntax
```
Specifies the syntax of the next argument. Valid values are
rpc_c_ns_syntax_default, which specifies that the syntax specified in the RPC_DEFAULT_ENTRY_SYNTAX environment variable should be used, and,
rpc_c_ns_syntax_dce.

```c
entry_name
```
Specifies the entry name where the search for compatible binding handle begins. If NULL is specified, the entry name in the environment variable RPC_DEFAULT_ENTRY is used.

```c
if_spec
```
A data structure generated by the stub that specifies the interface to import. Typical value is <intfc>_v<maj>_c_ifspec where <intfc> is the interface name specified in the .idl file, <maj> is the interface version major number and <min> is the interface version minor number.

```c
obj_uuid:
```
An object UUID. If object UUIDs are not being used, specify NULL.

```c
import_context
```
The handle returned to be used with subsequent rpc_ns_binding_import_next () calls.

```c
status
```
The status code returned.
4.1.2.2 rpc_ns_binding_import_next ()
This function returns a binding handle for a server offering the interface and object UUID specified in the rpc_ns_binding_import_begin () call. The server is randomly selected among all servers registered in the namespace with the same interface.

The prototype for the rpc_ns_binding_import_next () call is declared in the header file <dce/rpc.h>. Below we show the definition of the call with its arguments:

```c
rpc_ns_binding_import_next (
    rpc_ns_handle_t import_context,
    rpc_binding_handle_t *binding_handle,
    unsigned32 *status
);
```

The call arguments for rpc_ns_binding_import_next () are:
- `import_context`
  The import context returned by the rpc_ns_binding_import_begin routine.
- `binding_handle`
  The binding handle returned for the binding information imported.
- `status`
  The status code returned.

4.1.2.3 rpc_ns_binding_import_done ()
This function frees the context created by rpc_ns_binding_import_begin (). It should be used after the client has completed the remote procedure calls.

The prototype for the rpc_ns_binding_import_done () call is declared in the header file <dce/rpc.h>. Below we show the definition of the call with its arguments:

```c
rpc_ns_binding_import_done (
    rpc_ns_handle_t *import_context,
    unsigned32 *status
);
```

The call arguments for rpc_ns_binding_import_done () are:
- `import_context`
  The import context returned by the rpc_ns_binding_import_begin routine.
- `status`
  The status code returned.

The sample MATHX client source can be found in Figure 111 on page 201. It illustrates the use of explicit bindings.
4.1.3 Running Implicit Binding Using CDS

Figure 25 shows schematically how server and client are working together during the execution of an application that uses implicit binding and CDS.

Besides performing the tasks seen before for explicit binding, the client must now place the binding handle in a global variable known by the client stub in order to route the subsequent call implicitly to the appropriate server.

![Diagram of implicit binding process](image)

*Figure 25. Running an Application Using Implicit Binding*
4.1.4 Implementing Implicit Binding Using CDS

The implicit binding method requires that a binding handle be passed to RPC run-time through a global variable, enabling the client to address a specific server for a given interface. The code responsible for managing the binding handle resides in the client setup and in the client stub. The client setup code places the handle in the global variable that was defined through an .acf file. It is called implicit because the client (except for the setup) and the manager code are not affected since the handle is hidden in the global variable.

These are the steps to write a distributed application using implicit binding:

1. The binding method must be informed in a declaration within the .acf file. This information is necessary because the IDL compiler must know the name of the global variable that will store the binding handle to place it into the header file and to refer to it in the client run-time.

```c
[ implicit_handle ( handle_t global_handle ) ]
interface mathx
{
}
```

2. Create the client setup code that will obtain the binding information. This client setup code is like the one used with explicit binding but, in addition, includes the statement to store the binding handle in the global variable defined in the .acf file.

```c
/* Store the binding handle bh in the global variable defined in .acf */
global_handle = bh;
```

As noted previously, the manager code and the rest of the client code remains the same.

4.1.5 Implementing Automatic Binding

Automatic binding is the most straightforward to use since all binding management is “hidden” from the developer on the client side. The steps of getting the binding information and connecting with the server are performed by the client stub. The client stub checks an environment variable RPC_DEFAULT_ENTRY. This environment variable usually points to a group or a profile in the CDS namespace.

For example:

```c
set RPC_DEFAULT_ENTRY=/.:/Servers/mathx
```

This entry is the starting point for a namespace lookup concerning a server entry. This method does not allow the client to choose specific servers. With automatic binding, if the chosen server becomes unavailable, the client stub can automatically locate another server for the client application. Automatic and implicit binding cannot be used in the same interface since these are interface-wide binding methods. Specifying automatic binding in the .acf is optional since it is the default binding method. However, doing this, avoids getting a warning message from the IDL compiler.
4.1.6 Running With String Binding

Figure 26 shows schematically how server and client are working together during the execution of an application that uses explicit binding management with string binding. The tasks performed by the server setup are very similar to the ones seen using explicit binding with the exception that in place of doing an export to CDS of binding information, the server just converts the binding information to a string binding for display. Using the binding displayed as a command line parameter, the client program converts the string binding to binding information, placing the derived binding handle in a global variable known by the client stub.

Figure 26. Running an Application Using String Binding
4.1.7 Implementing String Binding

String binding is convenient in situations such as when a CDS is not available or the location of a server never changes or it takes too long to get information from CDS due to the location of the client with respect to the CDS.

An application that uses string binding must have some way of exchanging a binding string between the client and the server. A simple way of doing it could be the server printing its string binding during startup and this same string binding being provided to the client in the command line.

The client using string binding as a binding information source can use implicit or explicit binding management. The difference from the previous examples would be that the binding handle would come from a string and not imported from CDS. The method you choose to obtain binding information does not affect the manager code.

These are the steps to write a distributed application using string binding:

1. Define the interface for either explicit or implicit binding.
2. Create server setup code that performs all the steps seen in Chapter 3, “Server” on page 37 except that in place of exporting the binding information, generate a string binding to be used in the client program.
3. Create client setup code. This code gathers and converts string binding information to obtain a binding handle.

The following routines can be used for the server setup code:

4.1.7.1 rpc_binding_to_string_binding ()

This function returns the string representation of binding information from a binding handle. It is used by servers to display/store string binding information.

The prototype for the rpc_binding_to_string_binding () call is declared in the header file <dce/rpc.h>. Below we show the definition of the call with its arguments:

```c
void rpc_binding_to_string_binding ( 
    rpc_binding_handle_t binding_handle, 
    unsigned_char_t *string_binding, 
    unsigned32 *status 
);```

The call arguments for rpc_binding_to_string_binding () are:

- binding_handle
  - The binding handle provided.
- string_binding
  - The string representation of binding information returned.

4.1.7.2 rpc_string_free ()

This function frees a character string allocated by the RPC run-time. It should be called once for each character string allocated and returned by the RPC run-time.

The prototype for the rpc_string_free () call is declared in the header file <dce/rpc.h>. Below we show the definition of the call with its arguments:
void rpc_string_free (
    unsigned_char_t  *string,
    unsigned32        *status
);

The call arguments for rpc_string_free () are:

    string
    The string pointer returned in a previous RPC call.
    status
    The status code returned.

The following routine can be used for the client setup code:

4.1.7.3 rpc_string_binding_compose()

This function combines the components of a string binding into a string binding.
It is used by client or server applications

The prototype for the rpc_string_binding_compose () call is declared in the
header file <dce/rpc.h>. Below we show the definition of the call with its
arguments:

void rpc_string_binding_compose(
    unsigned_char_t  *obj_uuid,
    unsigned_char_t  *protseq,
    unsigned_char_t  *network_addr,
    unsigned_char_t  *endpoint,
    unsigned_char_t  *options,
    unsigned_char_t  **string_bindings,
    unsigned32        *status
);

The call arguments for rpc_string_binding_compose () are:

    obj_uuid
    The string representation of an object UUID.
    protseq
    The string representation of a protocol sequence.
    network_addr
    The string representation of a network address.
    endpoint
    The string representation of an endpoint.
    options
    The string representation of network options.
    string_binding
    The string representation of binding information.
4.1.7.4 rpc_binding_from_string_binding()

This function returns a server binding handle from a string representation of binding information. It is used by clients.

The prototype for the rpc_binding_from_string_binding() call is declared in the header file <dce/rpc.h>. Below we show the definition of the call with its arguments:

```c
void rpc_binding_from_string_binding (  
    unsigned_char_t  string_binding,  
    rpc_binding_handle_t *binding_handle,  
    unsigned32 *status  
);
```

The call arguments for rpc_binding_from_string_binding() are:

- `string_binding`
  The string representation of binding information:

- `binding_handle`
  The binding handle returned.

The sample server setup using string bindings can be found in Figure 123 on page 216. This is the SBIND example. It performs the same operation as the MATHX example, but obtains its binding information via string bindings. In this example the binding information is passed through the SBIND run JCL found in Figure 120 on page 211

---

**Note:**

In the SBIND example the server (SBINDS) prints the binding information using a "printf" statement. This binding information is manually passed to the client (SBINDRNC) through run-time JCL. Don’t forget to change the client run JCL to reflect your network address and port #.

In the previous example, the call rpc_ep_register() is not mandatory since the endpoint displayed can be used by the client, bypassing the use of the endpoint mapper.

It should be noted that since the rpc_binding_to_string_binding() call requires a binding handle as a parameter and the server setup works with a binding vector, care must be taken to supply the field that contains the binding handle from the binding vector. The format of the binding vector is defined in <dce/rpcbase.h>.

The sample string binding client (SBINDC) can be found in Figure 122 on page 213.
4.2 Running with a Well-Known Endpoint

Figure 27 shows schematically how server and client are working together during the execution of an application that uses explicit binding management and a well-known endpoint. The tasks performed by the server setup are very similar to the ones seen using explicit binding and CDS with the exception that in place of doing an export to CDS of binding information, the server just converts the binding information to a string binding for display and listens on the well-known endpoint.

4.2.1 Implementing Well-Known Endpoints

Running with a well-known endpoint is convenient when you want to test your client and manager code when a CDS is not available or when you do not want to use security of any type. The disadvantage is that your server will be listening on one port and only one port. If for some reason that port is in use before you start your server you will have to make adjustments. Either change your server and client to use another port or wait for the port to become available. If you know that the port you want to use is always going to be available the well-known endpoint model can be useful and easy to implement.
An application that uses well-known endpoints must have a way of setting up to listen on a specific port. The server sets up the port (in this example “2000”) and the client asks for that same port.

The client needs to know the well-known endpoint and binding information. The difference from the previous examples is that the binding handle is hard coded in the client and not imported from the CDS or passed manually as it is in the string binding example.

These are the steps to write a distributed application using string binding:
1. Define the interface for either explicit or implicit binding.
2. Create server setup code.
   This part is unique in that you will include the endpoint of your choice using the rpc_server_use_protseq_ep() call.
3. Create client setup code.
   This code differs from our other examples in that the ip address and endpoint are included in the rpc_string_binding_compose() call.

Complete client and server code using a well-known endpoint can be found in Appendix C, “MATHX Source (Well-Known Endpoint)” on page 259.

4.2.2 Running in a OpenEdition HFS Environment
Using the OpenEdition shell and the OpenEdition ISPF shell we copied our MATHX example source code into the HFS environment. We compiled, linked and ran MATHX in the OpenEdition shell. This book will not take you through the steps necessary to move the files from the PDS environment to HFS, but it will note the files that need to be moved and a sample makefile. The sample makefile is in Appendix C, “MATHX Source (Well-Known Endpoint)” on page 259. Use this sample as a basis for yours.

What is a makefile? Lets say that you are working on a number of programs that are linked together to create one executable program. While working on this set of programs, you will most certainly have an opportunity to change one, some or all of them at some time before the program is complete and correct. Every time you change one of the files, you have to recompile the file, then relink it with the existing object files to create a new executable program. The makefile makes it possible to recompile only the parts that need to be recompiled and relink with one command.

If you plan to work in the OpenEdition shell the sample makefile in Appendix C, “MATHX Source (Well-Known Endpoint)” on page 259 can help save you valuable time.

---

Make Utility Reference.


The PDS files that need to be moved to the HFS environment are:
Table 1. Files that Need to be Copied or Created in HFS

<table>
<thead>
<tr>
<th>File</th>
<th>Hierarchical File System</th>
<th>PDS Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>client source</td>
<td>mathxc.c</td>
<td>USERPRFX..&lt;APPLNM&gt;.C(MATHXC)</td>
</tr>
<tr>
<td>server source</td>
<td>mathxs.c</td>
<td>USERPRFX..&lt;APPLNM&gt;.C(MATHXS)</td>
</tr>
<tr>
<td>manager source</td>
<td>mathxm.c</td>
<td>USERPRFX..&lt;APPLNM&gt;.C(MATHXM)</td>
</tr>
<tr>
<td>idl</td>
<td>mathx.idl</td>
<td>USERPRFX..&lt;APPLNM&gt;.IDL(MATHX)</td>
</tr>
<tr>
<td>acf</td>
<td>mathx.acf</td>
<td>USERPRFX..&lt;APPLNM&gt;.ACF(MATHX)</td>
</tr>
</tbody>
</table>

With the above files in the HFS environment you will need to do the following to create an executable program:

- Run the IDL compiler to create:
  
  mathx_cstub.c
  mathx_sstub.c
  mathx.h

- Compile your source code using the makefile or using the commands described in the OpenEdition Distributed Computing Environment Base Services MVS: Application Development Guide (SC09-1484).

4.3 Creating Files for Your Application

For DCE applications you typically code the five files contained in Figure 28 on page 66. The HFS files names and the corresponding PDS file member names are listed. The name of your DCE application is represented by <applnm> for HFS files, and <APPLNM> for PDS files.

You do not have to use the naming convention for your application files as presented in the table; the names used in this chapter for the examples are for illustration only. Note that you can generally access either the HFS file or the PDS file when you use the DCE utilities in the shell environment. You can access only PDS files when you use the DCE utilities in the batch or TSO/E environments. You can copy the contents of an HFS file into a PDS by using the TSO/E OGET command and you can copy the contents of a PDS file into an HFS file using the TSO/E OPUT command. Refer to OpenEdition MVS Advanced Application Programming Tools (SC23-3017) for more information on these commands.
### Purpose of user-written files

1. There is usually one IDL file for a simple DCE client and server application. You write the IDL file using the Interface Definition Language (IDL). The IDL file defines all aspects of an interface that affect data passed over the network between a DCE client and the server.

2. The C source files contain the DCE and C language source code for your DCE application. Typically, there are three modules: a server, its manager routines, and the client.

3. This optional file is used to code the Attribute Configuration Language (ACL) that modifies the interaction between application code and stubs, and to declare certain DCE attributes.

### Note:

1. You can have more than one member in any of the above partitioned data sets. For example, you can have multiple members in the IDL file representing many interface definitions, depending on your application’s requirements. You can also write additional header members that contain special definitions for your application. For simple DCE applications, only one member is typically required.

2. When using a partitioned data sets, `<APPLNM>` cannot be greater six characters in length.

During the creation of a DCE application, the files listed in Figure 29 on page 67 are typically generated by IBM OpenEdition DCE Base Services MVS utilities such as the IDL compiler or the c89 compiler. If you use the naming convention of Table 2 on page 71, you will generate most of the files contained in the table, depending on the options you choose when you run the IBM OpenEdition DCE Base Services MVS utilities.
Figure 29. System-Written Files for DCE Applications

**Purpose of system written files:**

1. The stub files, one for your server and one for your client application, contain the code for marshalling and unmarshalling data, message handling, and other details of network communication. The stubs are generated when you compile the IDL file. Depending on the IDL compiler options you choose, you can generate the stubs directly as object code instead of C code.

2. The auxiliary files, one for your server and one for your client application, are special purpose files for marshalling and unmarshalling data structures and handling in-line or out-of-line code. These auxiliary files are generated if you specify certain attributes in your application’s IDL file and ACF file. The auxiliary files are generated when you compile the IDL file.

3. The header file is common across your server and client application. It is obtained from compiling the IDL file. The header file contains declarations and definitions that are for general use in an application. You must compile the header along with the source code when compiling either your client or server application.

4. The object files contain the object code for your application that is obtained from the compile step.

5. The load files contain the client and server application load modules are created by link-editing your application.

**In the Shell:** To enter the shell, enter OMVS on any TSO/E command line. You do not need to allocate any hierarchical file system (HFS) files for your DCE applications, but for your convenience, you may want to make a directory where
you can store all your associated DCE application files. The HFS files are created for you when you first edit them, using the TSO/E OEDIT command. You may consider storing all files that you generate using IBM OpenEdition DCE Base Services MVS utilities in this directory as well.

Following, in Figure 30, are the HFS file extensions that are appended to your application name by the IBM OpenEdition DCE Base Services MVS utilities:

<table>
<thead>
<tr>
<th>File</th>
<th>File extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>header file</td>
<td>&lt;applnm&gt;.h</td>
</tr>
<tr>
<td>server stub</td>
<td>&lt;applnm&gt;_sstub.c</td>
</tr>
<tr>
<td>client stub</td>
<td>&lt;applnm&gt;_cstub.c</td>
</tr>
<tr>
<td>client auxiliary (optional)</td>
<td>&lt;applnm&gt;_caux.c</td>
</tr>
<tr>
<td>server auxiliary (optional)</td>
<td>&lt;applnm&gt;_saux.c</td>
</tr>
</tbody>
</table>

*Figure 30. HFS File Extensions*

**Note:**

There is no restriction on the length of the IDL file name denoted by <applnm> in the shell as there is in the TSO/E and batch environments.

**In TSO/E and Batch:** Set up your environment by allocating all the necessary partitioned data sets for your application. Seven data sets are required:

- USERPRFX.<APPLNM>.JCL
- USERPRFX.<APPLNM>.IDL
- USERPRFX.<APPLNM>.C
- USERPRFX.<APPLNM>.H
- USERPRFX.<APPLNM>.OBJ
- USERPRFX.<APPLNM>.LOAD
- USERPRFX.<APPLNM>.ACF

**Note:**

You do not have to use the above naming convention for your application data sets. The names used in this chapter for the examples are for illustration only.

In the above list, USERPRFX represents your MVS/ESA logon identification, and <APPLNM> represents the name of your application.

Figure 31 on page 69 shows the recommended attributes for the above data sets. You should adhere to the record format and record lengths presented, or you may encounter difficulty running the JCL. You may adjust the block sizes of your data sets as required for your needs (in multiples of the record length).
You can allocate the above data sets manually using the allocate new data set panel (option 3.2 in the ISPF utilities), or you can use the IDLALL OC TSO/E command that is included with IBM OpenEdition DCE Base Services MVS. IDLALLOC allocates the necessary data sets required for your application. Note that IDLALLOC only allocates data sets that do not already exist.

To invoke IDLALLOC in native TSO/E, enter:
```
idlalloc <applnm>
```

From an ISPF command line, enter:
```
tso idlalloc<applnm>
```

The data sets created will be organized as partitioned organization (PO).

Figure 31 describes the data sets created using IDLALLOC. Verify that your data sets conform to these characteristics by using the data set information utility (option 3.2 in ISPF) before proceeding.

---

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Record Format</th>
<th>Record Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>USERPRFX.&lt;APPLNM&gt;.JCL(1)</td>
<td>FB</td>
<td>80</td>
</tr>
<tr>
<td>USERPRFX.&lt;APPLNM&gt;.IDL</td>
<td>FB</td>
<td>80</td>
</tr>
<tr>
<td>USERPRFX.&lt;APPLNM&gt;.C</td>
<td>FB</td>
<td>80</td>
</tr>
<tr>
<td>USERPRFX.&lt;APPLNM&gt;.H</td>
<td>FB</td>
<td>80</td>
</tr>
<tr>
<td>USERPRFX.&lt;APPLNM&gt;.OBJ</td>
<td>FB</td>
<td>80</td>
</tr>
<tr>
<td>USERPRFX.&lt;APPLNM&gt;.LOAD</td>
<td>U</td>
<td>6160</td>
</tr>
<tr>
<td>USERPRFX.&lt;APPLNM&gt;.ACF</td>
<td>FB</td>
<td>80</td>
</tr>
</tbody>
</table>

Figure 31. Recommended DCE Application Data Set Attributes

**Note:**

1. IBM OpenEdition DCE Base Services MVS does not require any particular JCL attributes. These values are given as an example only.
Chapter 5. Compiling and Link Editing

5.1 Compiling the Interface with the IDL Compiler

There are 3 ways to compile your source code. They are:

- In the OpenEdition Shell
- In TSO/E
- In batch

In this chapter we mostly use batch. See OpenEdition Distributed Computing Environment Base Services MVS: Application Development Guide (SC09-1484) for information and examples using the shell and TSO/E.

Upon completing the IDL file for your application, you can compile it using the IDL compiler. The IDL compiler normally outputs three files, and two optional files depending on the contents of your IDL file and compiler options you choose. All these files are compiled with your application source code:

<table>
<thead>
<tr>
<th>File</th>
<th>Hierarchical File System File Name</th>
<th>PDS Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>header</td>
<td>&lt;applNm&gt;.h</td>
<td>USERPRFX.&lt;APPLNM&gt;.H(APPLNM)</td>
</tr>
<tr>
<td>client stub</td>
<td>&lt;applNm&gt;_cstub</td>
<td>USERPRFX.&lt;APPLNM&gt;.C(APPLNMCS)</td>
</tr>
<tr>
<td>server stub</td>
<td>&lt;applNm&gt;_sstub</td>
<td>USERPRFX.&lt;APPLNM&gt;.C(APPLNMSS)</td>
</tr>
<tr>
<td>client auxiliary</td>
<td>&lt;applNm&gt;_caux</td>
<td>USERPRFX.&lt;APPLNM&gt;.C(APPLNMCA)</td>
</tr>
<tr>
<td>server auxiliary</td>
<td>&lt;applNm&gt;_saux</td>
<td>USERPRFX.&lt;APPLNM&gt;.C(APPLNMSSA)</td>
</tr>
</tbody>
</table>

The auxiliary files are generated depending on the contents of the IDL and ACF files.

For PDS names, <APPLNM> cannot be more than six characters long, or the IDL compiler will reject it. There is no similar restriction on HFS IDL file name.

The header contains the definitions and declarations derived from the input IDL file that are for general use in the development source code. The stubs, one for your server and one for your client, contain the code for marshalling and unmarshalling data, message handling, and other details of network communications management.

The two optional auxiliary files, one for your server and one for your client, are special purpose files for marshalling and unmarshalling data structures and handling in-line and out-of-line code. These auxiliary files are generated if you specify certain attributes in your application’s IDL file and ACF file. For more
information on the IDL compiler and stubs, refer to the OpenEdition Distributed Computing Environment Base Services MVS: Application Development Guide (SC09-1484).

The IDL compiler supplied with IBM OpenEdition DCE Base Services MVS uses an input IDL file to define a set of subroutine interfaces. Each set of interfaces is identified by its UUID. Along with the IDL file is the ACF that controls the IDL compiler’s interpretation of the IDL file. With the ACF, you can customize the client or server stubs to suit your local environment.

When you ship a server to your users or customers, you must ship the IDL and ACF files as well. The users can regenerate the client and server stubs after modifying the ACF to suit their local environment. You do not have to re-create the stubs if customization is not required.

5.1.1 Compiling the IDL

5.1.1.1 In Batch
Use the sample JCL shown in Figure 32 to compile the IDL file for your application and produce the files listed above. The PROC that runs is IDL. Pass any parameters to the IDL compiler using the PARM=’idl_compiler_options’ statement.

Note:
The IDL EXEC handles PDSs only.

```
//BMTJ JOB (999,POK),’BMT’,NOTIFY=BMT,
//    CLASS=A,MSGCLASS=T,REGION=8000K,
//    MSGLEVEL=(1,1)
//NEWLKL JCLLIB ORDER=(DCE.SEUVPRC)
//**********************************************************************
//* JCL TO GENERATE THE MATHX IDL STUBS AND IDL HEADER FILES *
//**********************************************************************
//* IDLCOMP EXEC IDL,USERPFX=’USERPFX.DCEMVS’,DCEPRX=’DCE’,
//    PARM=’mathx -v -client stub -server stub -keep all’,
//    REGSIZE=8000K
Figure 32. Sample JCL to Compile the IDL file.
```

When the MATHX.IDL file is compiled using IDL, the object stub files MATHXCS and MATHXSS and the header file MATHX.H are generated.

5.1.1.2 In TSO/E
Use the command shown in Figure 33 to compile the MATHX.IDL file from the TSO command line.

```
idl mathx.idl
Figure 33. TSO/E Command to Compile the IDL File.
```

The following warning message should appear:
Operation add has no binding handle parameter; [auto_handle] assumed

This is not an error. It is just the IDL compiler informing the binding method it will assume by default.

5.2 Compiling the Client and Server Programs

Once you have written the client, server, and manager source code, you are ready to compile them along with the client and server stubs and auxiliaries using the C/370 compiler.

Note:

You can compile your source files individually if you want, or you can combine the compile and link steps into one step.

Appendix A, “JCL and Source Listings for MATHX and SBIND Examples” on page 197 contains sample JCL to compile the client, server and manager code for the MATHX, SBIND and SECUR examples.

The cataloged procedure, EDCC, is supplied by the AD/Cycle C/370 product. Consult the IBM SAA AD/Cycle C/370 User’s Guide (SC09-1763) for more information on EDCC.

There are five major compile steps

• Compile the client stubs generated by the IDL compiler.
• Compile the server stubs generated by the IDL compiler.
• Compile the client portion of your application.
• Compile the server portion of your application.
• Compile the manager portion of your application server.

Each one of the above steps can run on its own, provided the job parameters section and USERLIB section specifying the application’s header file are completed. The input data set for your application source code is specified by the INFILE parameter. The output data set for your application’s object code is specified by OUTFILE parameter. The C/370 and DCE header files required to compile the DCE applications are specified on the SYSLIB parameter. You may compile more than one client, server, or manager at the same time by adding more job steps as required and modifying the input data sets as appropriate.

Specifying Compile Options: The sample JCL contained in Figure 104 on page 197 includes C/370 compiler options in using the CPARM= statement. Generally, all compile options can be entered at your discretion; however, all DCE applications must be compiled as re-entrant to enable the threading used by the RPC run-time. The LONGNAME option should also be used.
5.3 Link-Editing Your Application

Once you receive a return code 0 from your compile step, you are ready to link your object code with the DCE RPC run-time library, the C/370 run-time library, and the TCP/IP run-time library. There are two steps in the linking process:

1. PLKED (prelink step)
2. LINK (link step).

The C/370 prelink step is needed because the code is re-entrant and because LONGNAME support is required.

The sample code in Figure 34 shows the generic JCL to link the server code, manager code, and server stub code to create an executable MATHXS object module.

```jcl
//JOBCARD JOB (999,POK), 'USERID'.NOTIFY=USERID,
// CLASS=A,MSGCLASS=H,REGION=5000K,
// MSGLEVEL=(1,1)
/******************************************************
/* JCL TO LINK THE MATHX SERVER, MANAGER AND */
/* SERVER STUBS OBJECT CODE */
/******************************************************/
/*@*/
/*@ CUSTOMIZABLE SYMBOLIC PARAMETERS */
/*@*/
/*@ CPFX - FOR C/370 OBJECT LIBRARIES */
/*@ DCEPFX - FOR DCE OBJECT LIBRARIES */
/*@*/
/******************************************************
//LKSERVER EXEC DCELK,OUTFILE='USERPRFX.<APPLNM>.LOAD'
//USERLIB DD DSN=USERPRFX.<APPLNM>.OBJ,DISP=SHR
//PLKED.SYSPRINT DD SYSOUT=* 
//PLKED.SYSIN DD *
   INCLUDE USERLIB(MATHXS)
   INCLUDE USERLIB(MATHXM)
   INCLUDE USERLIB(MATHXSS)
//LKED.USERLIB DD DSN=USERPRFX.<APPLNM>.OBJ,DISP=SHR
//LKED.SYSPRINT DD SYSOUT=* 
//LKED.SYSIN DD *
   NAME MATHXS(R) 
/**
Figure 34. Sample JCL to Link the Server, Manager, and Server Stub Code.

The sample code in Figure 35 on page 75 shows the generic JCL to link the MATHX client object code and client stub object code with the various run-time libraries to create an executable MATHXC load module.
Figure 35. Sample JCL to Link the Client and Client Stub Code.

An alternate cataloged procedure that link-edits your client and server is EDCPL. This PROC is described in LE/370 Programming Guide.

**Note.**

The prelink step will end with a return code 4 because the C/370 run-time entry points, such as printf, strcpy, and so on, are unresolved. You should inspect the prelink output to ensure that none of your application’s entry points are unresolved.
6.1 Concepts of DCE Security

One of the features of DCE for building client/server applications is the possibility to use authenticated and authorized RPC calls. This means that if it is necessary both the server and the client can be sure they are communicating with the right party and nobody else can listen to the calls. Both the client and the server must agree on the protection level for the remote procedure calls.

The security server in conjunction with the DCE run-time, provides:
• authentication of client and server
• protection for the communication between the client and the server
• providing the authorization data to the server.

However, higher levels of protection result in greater overhead for the communication between client and server.

Security in DCE is based on two major concepts authentication and authorization of principals. Principals are represented as entries in the registry database. DCE principals include:
• Users, who are also referred to as interactive principals
• Instances of DCE servers
• Instances of application servers
• Computers in a DCE cell
• Authentication service surrogates.

Authentication is the identification of a principal to the security service. Once a principal is authenticated, the server can rely on the identity information it gets from the security service.

Authorization is granting an authenticated principal the right to access a service. Authorization for RPC calls is checked by the server based upon information provided by the security service. A principal must be authenticated for authorization.

Applications using authenticated RPC routines may use the authentication protocol, the authorization protocol and set various protocol-independent protection levels for communication with remote principals.

The DCE security component consists of three services:
• The registry service, which maintains the database of principals, groups, organizations, accounts and administrative policies.
• The authentication service, which verifies the identity of a principal and issues tickets used by the principal to access remote services. A ticket is data about a principal that is presented to the principal providing the service.
• The privilege service, which certifies a principal’s privilege attributes.
Figure 36 conceptually represents how an application calls the security services for authenticated RPC calls.

1. **Authentication:**
   The application which claims to be principal A requests authentication by the authentication service. The authentication service requests from the registry the information needed to authenticate principal A and sends principal A an encrypted ticket. If principal A provides the right secret key to decrypt the ticket the authentication is successful and principal A has a valid ticket.

2. **Retrieving PAC:**
   For authorization, the principal requests its privilege attribute by presenting its ticket. The privilege service sends back the privilege attribute certificate (PAC) which will be used for authorization.

3. **Authorization and RPC Call:**
   Principal A calls a remote procedure provided by the application server principal B. Along with the call and its parameters, principal A sends its PAC to principal B. The application server principal B decides upon the access control information and the PAC whether or not the right for the call is granted to principal A and responds to the request.
This mechanism is referred to as the trusted shared secret authentication for intracell authentication. Its main concept is that all principals trust the security service. The mechanism for intercell authenticated RPC is based on the same concept.

If client and server implement DCE security for authenticated and authorized RPC, this is mostly done by the RPC run-time. The application only has to specify how the security services will be used and provide the authentication information.

6.1.1 Authentication Levels

DCE offers several different types of authentication service. Before a client and server can engage in authenticated RPC, they must agree on the type of authentication service they will use. Specifically, each server and each client involved must associate an authentication service with their principal name in the DCE registry database.

To register a client, you must know the client principal name, the type of authentication service the client will use, and the level of authentication the client will require.

To register a server, you must know the server principal name, and the type of authentication service the server will use.

DCE provides four types of authentication services:

- **rpc_c_authn_none**
  No authentication; client and server agree not to do any authentication checking.

- **rpc_c_authn_dce_secret**
  DCE trusted shared key authentication provided by Kerberos as a part of the security service.

- **rpc_c_authn_default**
  DCE default authentication service; For MVS/ESA OpenEdition DCE this is the same as rpc_c_authn_dce_secret.

- **rpc_c_authn_dce_public**
  DCE public key authentication; This is reserved for future use and is currently not available.

The value rpc_c_authn_none is used to turn off authentication already established for a binding handle. The default authentication is DCE shared secret key authentication.

You specify what authentication service your server application will use when you establish your server principal identity as you set up for authenticated remote procedure calls. A client application indicates what service it will use when it annotates a binding handle with security information. This is also a part of doing authenticated remote procedure calls.
6.1.2 Communication Protection Levels

When a client establishes authenticated RPC, it can specify the level of protection to be applied to its communication with the server. The protection level determines the degree to which client/server messages are actually encrypted. As a rule, the more restrictive the protection level, the greater the impact on performance.

Authenticated remote procedure calls support the following protection levels:

- **rpc_c_protect_level_none**
  No communication protection.

- **rpc_c_protect_level_default**
  This is the default protection level for the specified authentication service. It depends on the protection settings for your cell.

- **rpc_c_protect_level_connect**
  This level performs protection only when the client establishes a relationship with the server. It performs an encrypted handshake the first time the client communicates with the server. Encryption/decryption is not performed on the data sent between the client and server. The fact that the handshake succeeds indicates that the client is active on the network.

- **rpc_c_protect_level_call**
  This level performs protection only at the beginning of each remote procedure call when the server receives the request. It attaches a verifier to each client call and server response.

- **rpc_c_protect_level_pkt**
  This level ensures that all data received is from the expected client. This level attaches a verifier to each message.

- **rpc_c_protect_level_pkt_integ**
  This level ensures and verifies that none of the data transferred between client and server has been modified. It computes a cryptographic checksum of each message to verify that none of the data transferred between client and server has been modified in transmit.

  This is the highest protection level that is guaranteed to be present in RPC run-time.

- **rpc_c_protect_level_pkt_privacy**
  This includes all previous protection levels and also encrypts all remote procedure call arguments. This level encrypts all user data in each call. However this protection level is only available outside the USA to customers who have obtained a license to receive the DCE Data Privacy package.

6.1.3 Authorization

Authorization is the process of checking a client permissions to access services offered by the server. It is the server responsibility to implement the authorization checking appropriate to its needs.

Authenticated RPC supports the following options for making authorization information available to servers for access checking:

- **rpc_c_authz_none**
No authorization information is provided to the server, usually because the server does not perform access checking.

- **rpc_c_authz_dce**
  
The client DCE PAC is provided to the server with each remote procedure call made with a binding handle. The server performs authorization checking using the client PAC to check access against DCE access control lists (ACL). This is referred to as certified authorization.

- **rpc_c_authz_name**
  
The client principal name is provided to the server. The server performs the authorization checking based on the provided principal name. This is referred to as name based authorization.

OSF recommends use of certified, or PAC based, authorization information to check access rights to the server. An option is for the server to use the access control list associated with an entry in the CDS name space to perform the checking. This entry could be, for example, the name exported by the server to the CDS name space. If the server uses its own ACL on objects managed by the server, it must also provide an ACL manager. This, for example, is done for the DCE distributed file system (DFS). DFS provides its own ACL manger for the objects in the DFS subtree /./fs of the cell name space. DFS uses authenticated remote procedure calls to implement the file handling calls and uses its own ACL manager before that.

The client authentication information used to check the authorization for the RPC calls is based on the user who logs into DCE and who subsequently invokes the client application. This information is referred to as the login context of the user. Applications which are not called by login principals can establish their own login context by using the `sec_login_` calls.

ACL managers and setting the login context is not covered in this section. Please refer to the OpenEdition Distributed Computing Environment Base Services MVS: Administration Guide (SC09-1485) for details.

If a user is not authenticated, the login context which is used for the RPC call depends on the platform. For instance, on AIX, non-root users who are not authenticated have no login context for the RPC call. For

- the root user on AIX,
- the user of an OS/2 machine who is not authenticated,
- and for the user of MVS/ESA OpenEdition DCE who is not authenticated,

the user login context will be the context for the principal `hosts/<hostname>/self`. This is the principal for the host machine in the DCE cell.

### 6.1.4 DCE Authorization

DCE Authorization, `rpc_c_authz_dce`, provides the server with the client privilege attribute certificate (PAC). The DCE PAC mechanism offers a trusted base for proving client authorization for calling remote procedures offered by the server.

The DCE security service generates a client PAC in a secure manner. If the server receives a client PAC, he can be sure that the PAC has been certified by the DCE security.
PACs are designed to be used with the DCE ACLs on DCE objects. For authorization, these objects are usually entries in the CDS name space exported by the server.

### 6.1.5 DCE Access Control Lists

Simply knowing a user’s identity doesn’t necessarily solve the problem of whether the user is authorized to use a particular resource. To determine a user’s authorization to use a particular resource, DCE provides an access control list (ACL) facility. More specifically, the ACL facility checks a principal’s access rights to objects in the cell. ACLs specify permissions that a user or group has relative to the objects. Permissions generally take the form of read, write, and execute actions allowed to a user but they are not limited to just these.

The ACL facility distinguishes between two types of objects: container objects and simple objects. Container objects contain other objects, which may be simple objects or other container objects. Simple objects do not contain other objects. Examples of container objects include a file system directory or a database; examples of simple objects include a file or a database entry.

An ACL manager type manages the ACLs for each class of objects for which permissions are uniquely defined. The manager type defines the permissions for those objects whose ACLs it manages. This includes the number of permissions and the meaning of the permissions.

A DCE ACL consists of the following:

- ACL manager type identifier, which identifies the manager type of the ACL
- Default cell identifier, which specifies the cell where local principals or groups are assumed to be members
- At least one ACL entry

DCE authorization defines two types of ACL entries:

- The privilege attribute entries associate a specific privilege attribute with a permission set. The privilege attributes entries includes identity based entries which refer to principal’s name, and others which refer to groups of users.
- The mask entries define a maximum permission set which can be used to limit the permission set specified by the privilege attribute entries.

Below we see an example for the ACLs of a simple object in the CDS name space. The command was issued under the OpenEdition shell. All entries besides the default cell refer to the privilege attributes.

```
    acl_edit -e /.:/servers/MessageBox
    sec_acl_edit>list

    # SEC_ACL for /.:/servers/MessageBox:
    # Default cell = /.../pokcell
    user:MessageBox:rwdtc
    group:subsys/dce/cds-admin:rwdtc
    group:subsys/dce/cds-server:rwdtc
    unauthenticated:r--t-
    any_other:r--t-
```
Each entry consists of:
- The type of the entry (user, group, unauthenticated, and any_other)
- The principal name or group name
- The privilege attributes.

The privilege attributes are dependant on the implementation of the applications ACL manager. The attributes implemented by the Cell Directory Services (CDS) name server, are:

<table>
<thead>
<tr>
<th>Token</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Read entry attributes</td>
</tr>
<tr>
<td>w</td>
<td>Update entry attributes</td>
</tr>
<tr>
<td>d</td>
<td>Delete entry</td>
</tr>
<tr>
<td>t</td>
<td>Test attributes values</td>
</tr>
<tr>
<td>c</td>
<td>Change ACL</td>
</tr>
</tbody>
</table>

For authorization checking, a server application can either:
- implement its own ACL manager, request an ACL handle from the security service for the target object and call its ACL manager which compares the ACLs with the client privilege certificate attribute (PAC).
- or make use of the CDS name server ACL manager. An object in the CD name space can represent a logical function in the application. The privilege attributes shown above can represent the level of access to the logical function. For example, in 6.2.1, "Description of the Sample Application" on page 84, the application determines whether a client is authorized to create a message queue by testing whether the client has "c" (change attribute) to the server binding export object, "/.:/servers/MessageBox" but does not provide its own ACL manager.

### 6.1.6 Name-Based Authorization

The second authorization method supports applications which make the authorization decision based on the principal string name without using the DCE privilege service. This is referred to as name-based authorization.

The DCE security service authenticates the string name representation of a principal. DCE security converts this string representation to the associated UUID which is used by an ACL manager for authorization decisions.

It is assumed that an application which uses name-based authorization has means to associate string names with permissions. In name-based authorization, there is no UUID representation of privilege attribute data, but the DCE ACL manager recognize only UUIDs. If an application uses name-based authorization, a principal’s privilege attributes are represented as an anonymous PAC. Such PAC data can only match the anonymous ACL entry types such as other_obj and are masked with an unauthenticated mask entry. Thus, it makes little sense to use the principal’s privilege attributes for name based authorization together with DCE authorization.

In the following section we provide a message handling example using name based authentication to get the senders name.
6.2 Example

This section shows the implementation of an example application which uses the DCE security service to provide authenticated remote procedure calls. The example uses both DCE authorization and name based authorization for the remote procedure calls.

6.2.1 Description of the Sample Application

The example implements a simple message queuing system which consists of one server which stores the messages in memory and exports some remote procedure calls for the clients to access the messages. The server checks the authentication information from the client to prove the authorization for each RPC call.

The client implements four tasks for the message queuing system:

- Send a message to another user
- Read an incoming message
- Create a new message box for a new user
- Shut down the server

Sending a message causes no authorization checks on the caller. It is assumed that all DCE users are allowed to send messages. The server tries to get the sender’s principal name for reference in the message. If this fails the message is stored without the sender’s name.

If a principal tries to read its messages, the server asks the DCE security service to provide the principal name of the user. The user can only reach his message box, if DCE security provides the name and his message box exists.

The last task implemented by the server is the creation of a message box for a new user, or shutting down the server.

If the name of the principal for whom the message box is being created is "shutdown," the server is shut down.

The corresponding RPC call checks the authorization of the caller based on the ACL entries of the server name space entry. Authorization is granted if the user has the control rights over this entry. This right is always granted to the owner of the name space entry.

The server registers his principal name along with the security settings to the security service. For doing so, the server needs the password for his principal’s account. For easier password management and to prevent the server from being a security hole, the server password should never be hard coded into the source code. DCE provides a facility to store server principal passwords encrypted in a file. The default is the host’s keytab file in which the DCE services store their password information. It is recommended that application servers not use the default keytab file, and the application’s keytab file be protected appropriately.
6.2.2 Interface Definition

The interface for the remote procedure calls between our sample client and server describes three procedures exported by the server. These remote procedures use the security service for authorization checking of the client.

mbox_new()

This routine creates for a given principal name a new message box. Only a principal with control rights on the server name space entry is granted the right to call this routine. This routine terminates the server if the principal name is "shutdown."

mbox_append()

The mbox_append call appends a given message to the message queue of the referred principal. The server asks the security service for the sender's principal name but no access checking is done.

mbox_next()

A principal can look up his message queue by calling this routine. Every call retrieves the oldest message, after delivering, the message will be discarded by the server. The server queries the security service to provide the readers principal name for access checking.

Below we show the full interface definition file for the sample client/server application.

```plaintext
[  
  uuid(003597CE-8E8D-1BFB-A052-10005A4F3057),
  version(2.0)
]

interface MessageBox
{
  const long MAX_CHAR = 255;
  typedef [string] char string_t[MAX_CHAR];

  long int mbox_new (  
    [in] handle_t bh,
    [in] string_t principal
  );

  long int mbox_append (  
    [in] handle_t bh,
    [in] string_t principal,
    [in] string_t message
  );

  long int mbox_next (  
    [in] handle_t bh,
    [out] string_t message
  );
}
```

Figure 37. Security Sample IDL File
The interface definition file for our sample message queuing example defines a type `string_t` for the remote procedures arguments. This typedef statement provides the IDL compiler with information on how many bytes in total the stub code has to transfer for the arguments. In the implementation of the remote procedures these arguments are implemented as pointers to `char`. For C these are equal types but we don’t have to use fixed arrays for the implementation.

### 6.2.3 Attribute Configuration File

The calls for querying the security service require a binding handle. To keep the example simple we used explicit binding between client and server.

The ACF file specifies the binding method for the interface as follows:

```plaintext
[explicit_handle] interface MessageBox
{
}
```

### 6.2.4 Server Implementation

The server is split into three modules representing the three main parts of the server:

- The server module, which registers the server to the RPC run-time.
- The management module, which implements the remote procedures exported by the server and some related procedures.
- The security module, which provides the authorization routines called by the remote procedures.

These three modules will be compiled separately and linked together with the server stub module to get the server program.

### 6.2.5 Server Module

The server module implements the steps for registering the interface to the RPC run-time, registering the endpoint to the endpoint mapper and exporting the binding information to the name space. The server module also sets up minor error handling by installing a signal handler. This is used together with a `TRY { }, FINALLY { }, ENDTTRY` sequence to clean up binding information and to save message box entries and unread messages to a file when the server ends.

This is the skeleton of every DCE server and has been explained in other sections of the document. The main differences are the login sequence using the keytab file for getting the encrypted password, and the `rpc_server_register_auth_info()` call used to register the security properties to the RPC run-time.

Figure 38 on page 87 shows the calling chain of the server module with the setup call for authenticated remote procedure calls.
6.2.5.1 \texttt{sec_login_setup_identity()}

The call \texttt{sec_login_setup_identity()} is used to retrieve sealed network credentials (or certificates) that are unsealed during the operation of validating an identity. In a Kerberos environment, this operation acquires an appropriate ticket-granting ticket.

The \texttt{sec_login_setup_identity} operation and the \texttt{sec_login_validate_identity} operation are two halves of a single logical operation. Together they collect the identity data needed to establish an authenticated identity. Because the operations are independent, the server’s password need not be sent across the network.

The prototype for \texttt{sec_login_setup_identity()} is declared in the header file \texttt{<dce/sec_login.h>}. Below we show the definition of the call with its arguments:

\begin{verbatim}
boolean32 sec_login_setup_identity(
    unsigned_char_p_t principal ,
    sec_login_flags_t flags ,
    sec_login_handle_t *login_context ,
    error_status_t *status
);
\end{verbatim}

The call arguments of \texttt{sec_login_setup_identity()} are:

\begin{verbatim}
principal
\end{verbatim}
A pointer (type unsigned_char_p_t) indicating a character string containing the principal name on the registry account corresponding to the calling process.

flags

A set of flags of type sec_login_flags_t. These flags contain information about how the new network credentials are to be used.

login_context

A pointer to an opaque handle to login-context data. The login context contains, among other data, the account principal name and UUID, account restrictions, records of group membership, and the process home directory.

status

A pointer to the completion status.

In the example for using the security service we specify the PRINCIPAL_NAME and the flags of sec_login_no_flags which places no restrictions on passing the credentials to another process. On return the login_context has been initialized for the principal. We issued the following call to setup the identity in the example server module:

```c
sec_login_setup_identity(
    PRINCIPAL_NAME,
    sec_login_no_flags,
    &login_context,
    &status
);
```

### 6.2.5.2 sec_key_mgmt_get_key()

The call sec_key_mgmt_get_key() is used to extract the specified key from the local key store.

The prototype for sec_key_mgmt_get_key() is declared in the header file <dce/keymgmt.h>. Below we show the definition of the call with its arguments:

```c
void sec_key_mgmt_get_key(
    sec_key_mgmt_authn_service authn_service ,
    void *arg ,
    unsigned char *principal_name ,
    unsigned32 key_vno ,
    void **keydata ,
    error_status_t *status
);
```

The call arguments of sec_key_mgmt_get_key() are:

- **authn_service**
  Identifies the authentication protocol using this key. The possible authentication protocols are as follows:
  - rpc_c_authn_dce_secret DCE shared-secret key authentication.
  - rpc_c_authn_dce_public DCE public key authentication (reserved for future use).
  - rpc_c_authn_dce_private DCE private key authentication (an implementation of the Kerberos system).

- **arg**
This parameter can specify either the local key file or a parameter to the
get_key_fn key acquisition routine of the rpc_server_register_auth_info API.

A value of NULL specifies that the default key file (/krb/v5srvtab) should be
used. A key file name specifies that file should be used as the key file. You
must add FILE: as a prefix to the absolute file name of the file. The file must
have been created with the rgy>Edit ktadd command or the
sec_key_mgmt_set_key function.

Any other value specifies a parameter for the get_key_fn key acquisition
routine.

principal_name
A pointer to a character string indicating the name of the principal to whom
the key belongs.

key_vno
The version number of the desired key.

keydata
A pointer to a value of type sec_passwd_rec_t. The storage for keydata is
allocated dynamically, so the returned pointer actually indicates a pointer to
the key value. The storage for this data may be freed with the
sec_key_mgmt_free_key function.

status
A pointer to the completion status.

In the example for using the security service we specify rpc_c_authn_dce_secret,
the name of the keytab file, and the principal name. On return the server_key
will point to the key structure. We issued the following call to get the key in the
example server module:

sec_key_mgmt_get_key(
    rpc_c_authn_dce_secret,
    KEYTAB,
    PRINCIPAL_NAME,
    0,
    &server_key,
    &status
);

6.2.5.3 sec_login_validate_identity()
The call sec_login_validate_identity() is used to validate the login context
established with sec_login_setup_identity. The caller must know the server’s
password for this operation to succeed. This operation must be called before
the network credentials can be used.

When a network identity is set, only state information for network operations has
been established. The local operating system identity has not been changed.
The caller must establish any local operating identity state.

The sec_login_setup_identity operation and the sec_login_validate_identity
operation are two halves of a single logical operation. Together they collect the
identity data needed to establish an authenticated identity. Because the
operations are independent, the server’s password need not be sent across the
The identity validation performed by sec_login_validate_identity is a local operation.

The prototype for sec_login_validate_identity() is declared in the header file <dce/sec_login.h>. Below we show the definition of the call with its arguments:

```c
boolean32 sec_login_validate_identity(
    sec_login_handle_t login_context,
    sec_passwd_rec_t *passwd,
    boolean32 *reset_passwd,
    sec_login_auth_src_t *auth_src,
    error_status_t *status
);
```

The call arguments of sec_key_mgmt_get_key() are:

- **login_context**
  
  An opaque handle to login context data. The login context contains, among other data, the account principal name and UUID, account restrictions, records of group membership, and the process home directory.

- **passwd**
  
  A password record to be checked against the password in the principal's registry account. The routine returns TRUE if the two match.

  **Note:**
  
  The contents of the passwd parameter are erased after the call has finished processing.

- **reset_passwd**
  
  A pointer to a 32-bit boolean32 value. The routine returns TRUE if the account password has expired and must be reset.

- **auth_src**
  
  How the login context was authorized. The sec_login_auth_src_t data type distinguishes the various ways the login context was authorized. There is one possible value: sec_login_auth_src_network

- **status**
  
  A pointer to the completion status.

In the example for using the security service we specify the server_key returned by the sec_key_mgmt_get_key. This has the encrypted password in it. We issued the following call in the example server module:

```c
identity_valid = sec_login_validate_identity(
    login_context,
    server_key,
    &reset_passwd,
    &auth_src,
    &status
);
```
6.2.5.4  rpc_server_register_auth_info()

The call `rpc_server_register_auth_info()` is used to register the authentication methods used by the server. The prototype for `rpc_server_register_auth_info()` is declared in the header file `<dce/rpc.h>`. Below we show the definition of the call with its arguments:

```c
void rpc_server_register_auth_info (
    unsigned_char_t *server_princ_name,
    unsigned32 authn_svc,
    rpc_authn_key_retrieval_fn get_key_fn,
    void *arg,
    unsigned32 *status
);
```

The call arguments of `rpc_server_register_auth_info()` are:

- **server_princ_name**
  
  The principal name to use for the server when authenticating remote procedure calls. The content of the name and its syntax is defined by the authentication service in use.

- **authn_svc**
  
  This argument specifies the authentication service to use when the server receives a remote procedure call request. That is `rpc_c_authn_name`, `rpc_c_authn_dce_secret`, `rpc_c_authn_default`, or `rpc_c_authn_dce_public`.

- **get_key_fn**
  
  This specifies the address of a server provided routine that returns encryption keys. Specify a NULL argument to use the default method of acquiring encryption keys.

- **arg**
  
  The arguments to pass to the `get_key_fn` routine, if not specified the default method is used.

- **status**
  
  This returns the status code from the interface register operation. This status code is a value that indicates whether the routine completed successfully and, if not, why.

In the example for using the security service we request the default authentication method `rpc_c_authn_default`. In MVS/ESA OpenEdition DCE, OS/2, and AIX this is the same as `rpc_c_authn_dce_secret`. We also request the default method of acquiring encryption keys by issuing a NULL argument for the `get_key_fn` argument. We issued the following call to register the authentication method in the example server module:

```c
rpc_server_register_auth_info(
    PRINCIPAL_NAME,
    rpc_c_authn_default,
    NULL,
    NULL,
    &status
);
```
6.2.5.5 Server Module C Source
The full C source code for the server module of the message box example is in the appendix (see B.1, “Server Module C Source” on page 221).

6.2.6 Manager Module
The manager module declares a data structure `mbox` which keeps the messages in memory while the server is running. All procedures referring to this structure are implemented in this module, particularly the remote procedures exported by the server. The exported procedures are:

- `mbox_new()`
- `mbox_append()`
- `mbox_next()`

The data structure `mbox` keeps the message boxes for all user principals participating on the message box system. It consist of an array of message box entries with the principal name and a pointer to list of messages for this principal. The list of messages is a chain of message entities with sender’s principal name, the message itself and a pointer to the next list entry. Figure 39 shows schematically the memory structure of the `mbox`.

![Figure 39. Mbox Data Structure](image)

When a message box is created by the `mbox_new()` call the next unused `mbox` entry is searched and used for the new user principal by inserting the principals name in the structure. Before creating the new entry the authorization for creating a new message box is checked by a call to the `is_authorized()` routine from the security module.

A new incoming message is appended to a principal’s message box by the `mbox_append()` remote procedure call. A new message entry structure is created...
and appended at the end of the message list. The message and the sender’s principal name is assigned to the entry.

The mbox_next() RPC routine is used by a principal to receive messages from his message box. The routine checks the authorization of the caller by retrieving his principal name from the security service. If the principal is a valid message box user the message with the sender principal name is returned and the message is discarded. Therefore, mbox_next() uses name based authorization.

The routines mbox_export() and mbox_import() are used by the server to save message box entries and messages to a file while the server is not running. The routine mbox_import() is called during server setup to initialize the mbox structure with the previous entries. If the server is terminating its execution, mbox_export() is called to export the entities in the mbox structure and the unread messages.

6.2.6.1 Manager Module C Source

The full C source code for the manager module of the message box example is in the appendix (see B.2, “Manager Module C Source” on page 230).

6.2.7 Security Module

The security module implements the authorization checking routines for the MessageBox example. It implements for the server two authorization checking routines:

- get_principal()
- is_authorized()

These routines implement name based authorization and DCE authorization for the message box server. They are called by the remote procedures to prove the authorization of the caller principal.

The routine get_principal() retrieves from the security service the principal name and returns it to the remote procedure.

The routine is_authorized() implements DCE authorization by checking the clients PAC against the ACL of the server name space entry. The authorization is successful if the principal associated with the binding handle has the control right for the entry. The control right is by default assigned to the owner of the CDS entry, to the group of CDS servers subsys/dce/cds-server and to the administrator group subsys/dce/cds-admin.

Below we show the output of the acl_edit command for the name space entry of the message box server /.:/servers/MessageBox

```
> acl_edit -e /.:/servers/MessageBox
sec_acl_edit> 1

# SEC_ACL for /.:/servers/MessageBox:
# Default cell = /.../pokcell
unauthenticated:r--t-
user:MessageType:rwdtc
group:subsys/dce/cds-admin:rwdtc
group:subsys/dce/cds-server:rwdtc
any_other:r--t-
sec_acl_edit>
```
In this example the principal MessageBox and the members of the groups subsys/dce/cds-server, subsys/dce/cds-admin have the control right on that entry and therefore are allowed to create new message box entries.

Figure 40 shows the calling chain for DCE authorization in our message box example.

Figure 40. Calling Chain for DCE Authorization

6.2.7.1 rpc_binding_inq_auth_client()

The call rpc_binding_inq_auth_client() is used by the server to inquire about the authentication and authorization information associated with the binding handle of the client. The prototype for the rpc_binding_inq_auth_client() call is declared in the header file <dce/rpc.h>. Below we show the definition of the call with its arguments:

```c
void rpc_binding_inq_auth_client(
    rpc_binding_handle_t client_binding,
    rpc_authz_handle_t *privileges,
    unsigned_char_t* server_principal,
    unsigned32 *protect_level,
    unsigned32 *authn_svc,
    unsigned32 *authz_svc,
    unsigned32 *status
);
```

The call arguments of rpc_binding_inq_auth_client() are:

- `client_binding`
  This specifies the binding handle of the client that made this remote procedure call.
- `privileges`
This returns a handle to the privileges information for the client that made the remote procedure call associated with the binding handle.

(server_principal)

This returns a pointer to the server principal name specified by the client. Specify a NULL value to prevent the procedure from returning this argument.

(protect_level)

This returns the level of authentication requested by the client. Specify a NULL value to prevent the procedure from returning this argument.

(authn_svc)

This returns the authentication service requested by the client. Specify a NULL value to prevent the procedure from returning this argument.

(authz_svc)

This identifies the type of data structure pointed to by the privileges argument. Specify a NULL value to prevent the procedure from returning this argument. Valid values are:

- rpc_c_authz_none a NULL value is returned
- rpc_c_authz_name the client principal name
- rpc_c_authz_dce the client PAC information

(status)

This returns the status code from the inquiry authentication operation.

In the security module of the MessageBox server the rpc_binding_inq_auth_client() call is used to get client credentials by the following call:

```c
rpc_binding_inq_auth_client(
    bh,
    &Credentials,
    NULL, /* we provide no principal name */
    NULL, /* no protection level returned */
    NULL, /* no authn_svc returned */
    &authz_svc,
    &status
);
```

6.2.7.2 sec_acl_bind()

Any operation performed on an ACL uses an ACL handle to identify the subject of operation. The call sec_aclBind() returns an ACL handle bound to the indicated object's ACL. The returned handle is required for each other sec_acl_ routine to identify the ACL on which to operate.

The prototype for the sec_acl_bind() call is declared in the header file <dce/daclif.h>. Below we show the definition of the call with its arguments:

```c
void sec_acl_bind(
    unsigned_char_t entry_name,
    boolean32 bind_to_entry,
    sec_acl_handle_t *acl_handle,
    error_status_t *status
);
```

The call arguments of sec_acl_bind() are:
**entry_name**

The entry name of the target object. ACL operations using the returned handle will affect the ACL of this object.

**bind_to_entry.**

The bind indicator, is used when entry_name identifies both an entry in the global name space and an actual object. A value $\neq 0$ binds the handle to the entry in the name space, while a value $= 0$ binds the handle to the actual object.

**acl_handle**

The returned ACL handle which refers to the ACL on the target object.

**status**

This returns the status code from the ACL operation.

In the MessageBox example we called sec_acl_bind() in the security module to get the ACL handle for the name space entry of the server. We issued the following call:

```c
sec_acl_bind(
    "/..:.servers/MessageBox",
    1,
    &acl,
    &status
);
```

### 6.2.7.3 sec_acl_get_manager_types()

The sec_acl_get_manager_types() routine returns a list of ACL managers of the specified type that are protecting the object associated with the ACL handle. We need to know which ACL manager will test the ACL permissions.

The prototype for the sec_acl_get_manager_types() call is declared in header file `<dce/daclif.h>`. Below we show the definition of the call with its arguments:

```c
void sec_acl_get_manager_types(
    sec_acl_handle_t acl_handle,
    sec_acl_type_t acl_object,
    unsigned32 array_size,
    unsigned32 *size_used,
    unsigned32 *mgr_available,
    uuid_t manager_types[],
    error_status_t *status
);
```

The call arguments of sec_acl_get_manager_types() are:

**acl_handle**

The ACL handle referring to the ACL of the target object. Use sec_acl_bind() to create this handle.

**acl_object**

The type of ACL on target. Possible values are:
- `sec_acl_type_object`
- `sec_acl_type_default_object`
- `sec_acl_type_default_container`
array_size

This argument contains the allocated length of the manager_types array provided by the caller.

size_used

This returns the number of assigned UUIDs in the manager_types array.

mgr_available

This returns the number of available manager_types UUIDs. This may be greater than size_used if there was not enough space allocated in the manager_types array.

manager_types

An array of size array_size to contain the UUIDs identifying the different types of ACL managers protecting the target object.

status

This returns the status code from the ACL operation.

In the MessageBox example we called sec_acl_get_manager_types() in the security module to get the first ACL manager for the name space entry of the server. We issued the following call:

\[
\text{sec_acl_get_manager_types(}
\begin{align*}
\text{acl,} \\
\text{sec_acl_type_object,} \\
\text{NUM_ELEMS( mgrs ),} \\
&\text{num_rtn}, \\
&\text{num_avail}, \\
&\text{mgrs,} \\
&\text{&status}
\end{align*}
\];

6.2.7.4 sec_acl_test_access_on_behalf()

The sec_acl_test_access_on_behalf() routine determines if the ACL on the target object granting privileges to a process associated with the given PAC. The call only succeeds if both the calling process and the process referred to by the PAC have sufficient rights granted by the ACL.

The prototype for the sec_acl_test_access_on_behalf() call is declared in the header file <dce/daclif.h>. Below we show the definition of the call with its arguments:

\[
\text{boolean32 sec_acl_test_access_on_behalf(}
\begin{align*}
\text{sec_acl_handle_t } &\text{acl_handle,} \\
\text{uuid_t } &\text{manager_type,} \\
\text{sec_id_pac_t } &\text{subject,} \\
\text{sec_acl_permset_t } &\text{permissions,} \\
\text{error_status_t } &\text{status}
\end{align*}
\];

The call arguments of sec_acl_test_access_on_behalf() are:

acl_handle

The ACL handle referring to the ACL of the target object. Use sec_acl_bind() to create this handle.

manager_type
A UUID identifying the type of ACL manager used to check the ACL. Use 
sec_acl_get_manager_types() to acquire a list of manager types protecting a 
given object.

subject
A privilege attribute certificate (PAC) for the subject process.

permissions
A 32-bit set of permission flags containing the desired privileges.

status
This returns the status code from the ACL operation.

In the MessageBox example we called sec_acl_get_test_access_on_behalf() in 
the security module to check control right of the client application for the name 
space entry of the server. We issued the following call:

```c
accessOK = sec_acl_test_access_on_behalf(
    acl,
    &mgrs[0],
    Credentials,
    CONTROL,    /* permissions to check for    */
    &status
);
```

### 6.2.7.5 Security Module C Source
The full C source code for the security module of the message box example is in 
the appendix (see B.3, "Security Module C Source" on page 241).

#### 6.2.8 Client Module
The client module implements the user interface for our message box example. 
It is used to carry out the four tasks for the message box system.

- Sending a message to another principal
- Reading an incoming message
- Creating a new message box for a new user
- Shutting down the server.

Which task would be done depends on the command line arguments of the client 
program. The four tasks are done by calling the corresponding remote 
procedures exported from the server.

Depending on the task the clients annotate their binding handle to the server 
with different security information.

For creating a new message box the client registers the PAC to the binding 
handle and the server can prove it against the ACLs. For all other tasks the 
client annotates his principal name to the binding handle.

The call syntax for the four tasks are:

```
message <principal> <Message Text> ...
```

Sends a message to principal

```
message
```

Shows the oldest message for caller’s principal
message -c <principal>
Creates a new message box for principal
message -c shutdown
Shuts the server down.

**Note:**
"message" is the name of a REXX exec that invokes the client program, MBOXC (see Appendix B.7, “Message REXX exec” on page 256).

After parsing his command line the client imports the binding information from the CDS name space and calls `rpc_binding_set_auth_info()` according to the task. If this is successful the client calls the remote procedure for carrying out the task.

Figure 41 shows the calling chain of the client module for the remote procedure setup.

![Figure 41. Calling Chain of the Client for Authenticated RPC](image)

**6.2.8.1 rpc_binding_set_auth_info()**
The call `rpc_binding_set_auth_info()` is used by the client to set up a binding handle for making authenticated remote procedure calls. Unless a client calls this routine, all remote procedure calls with that binding handle will be unauthenticated. This routine is the counterpart of the `rpc_binding_inq_auth_client()` routine used by the server.
The prototype for the `rpc_binding_set_auth_info()` call is declared in the header file `<dce/rpc.h>`. Below we show the definition of the call with its arguments:

```c
void rpc_binding_set_auth_info(
    rpc_binding_handle_t bhandle,
    unsigned_char_t *server_principal,
    unsigned32 protect_level,
    unsigned32 authn_svc,
    rpc_auth_identity_handle_t auth_ident,
    unsigned32 authz_svc,
    unsigned32 status
);
```

The call arguments of `rpc_binding_set_auth_info()` are:

- **`bhandle`**
  The binding handle to which authentication and authorization information is set.

- **`server_principal`**
  The principal name of the server referenced by the binding handle in the `bhandle` argument. Specify a NULL to allow one-way authentication. In effect, this means the client does not care which server principal receives the remote procedure call request. However, the server will verify that the client is who the client claims to be.

- **`protect_level`**
  This specifies the level of authentication to be performed on remote procedure calls.

- **`authn_svc`**
  This specifies the authentication service to use. That is `rpc_c_authn_none, rpc_c_authn_dce_secret, rpc_c_authn_default, or rpc_c_authn_dce_public`.

- **`auth_ident`**
  The clients PAC, a handle to a data structure that contains the client authentication credentials appropriate for the selected authorization and authentication service. Specify a NULL value to use the default security login context for this host.

- **`authz_svc`**
  This specifies the authorization service implemented by the server for the remote procedure called by the client. That is `rpc_c_protect_level_none, rpc_c_protect_level_default, rpc_c_protect_level_connect, rpc_c_protect_level_call, rpc_c_protect_level_pkt, rpc_c_protect_level_pkt_integ, or rpc_c_protect_level_pkt_privacy`.

- **`status`**
  This returns the status code from the interface register operation.

In the MessageBox example we called `rpc_binding_set_auth_info()` with request for package checking with checksums (`rpc_c_protect_level_pkt_integ`), the default authentication service (`rpc_c_authn_default`) and the request to use callers login context for authorization checking. Depending on which MessageBox function will be called we requested that the authorization information derived by the login context should be the PAC or clients principal name.
For creating a message box for a new user we issued the following call:

```c
rpc_binding_set_auth_info(
    bh,
    "/.:/servers/MessageBox",
    rpc_c_protect_level_pkt_integ,
    rpc_c_authn_default,
    NULL, /* use login context */
    rpc_c_authz_dce,
    &status
);
```

For all other MessageBox functions the authz_svc parameter is set to name based authorization by issuing the rpc_c_authz_name value.

### 6.2.8.2 Client Module C Source

The full C source code for the client module of the message box example is in the appendix (see B.4, “Client Module C Source” on page 246).

### 6.2.8.3 Common Header File

The full C source code for the common header file of the message box example is in the appendix (see B.5, “Common Header File” on page 253).

### 6.2.9 Running the Example Application

Before running the application there are some setup steps necessary for the server. The server needs its own principal and its password exported to the application’s keytab file.

The following list shows the steps to setup the application.

1. Create an HFS directory /u/MessageBox
2. Login and authenticate as cell_admin
3. Create a principal named MessageBox
4. Create an account for the principal MessageBox
5. Make MessageBox a member of subsys/dce/rpc-servers-group group.
6. Export the server key to the application’s keytab file.

   ```
   > rgy_edit
   Current site is: registry server at /.:/subsys/dce/sec/master
   rgy_edit==> ktadd -p MessageBox -f /u/MessageBox/Message.key
   Enter password:********
   Re-enter password to verify:********
   rgy_edit==> exit
   >
   ```

**Note:**

The keytab file, by default, will have permission bits that will only allow the owner to have access. The owner’s UID is assigned in the RACF OpenEdition segment independent of the fact that you are logged in as cell_admin. If the application is going to be executed under the current RACF userid, there is no problem. However, if it is going to be executed under a different RACF userid, you will have to use the OpenEdition command, CHOWN, to change the ownership and/or use CHMOD to set the group permission bits.
7. Use cdscp to create a directory, /.:servers and an object, /.:servers/MessageBox

8. Use acl_edit to give user:MessageBox:rwdtc access.

After these setup steps, the server is ready to run.

Submit the batch JCL shown in Appendix B.8, "Batch JCL to Run Messagebox Server" on page 257, after tailoring it to the data set names and jobcard requirements of your installation. The server can run in either normal mode or debug mode. To use debug mode, change the PARM field on the EXEC from "posix(on)/" to "posix(on)/-D."

The server uses the DCE sec_login APIs to login as MessageBox using the keytab file, exports its interface, registers its authentication information, tries to import old message entries from a file and starts listening for remote procedure calls.

Below we show extracts from the output of these steps. This output can be seen by using the TSO/E SDSF to browse the executing job's output.

```
Cannot import messages from file message.box
Server /.:servers/MessageBox listening.
```

Now the server is ready. We can create message boxes for some principals.

We authenticate as MessageBox to get control access to the server name space entry.

```
Note:

The message command below is a REXX exec that invokes MBOXC with the posix(on) parameter (see B.7, "Message REXX exec" on page 256).

> dcelogin MessageBox
Enter Password:
>
> message -c ericfin
> message -c vanaers
> message -c bmount
> message -c swanson
> message -c saul
>
The server prints out his actions.
Server /.:servers/MessageBox listening.
Creating new Message Box for Principal ericfin.
Creating new Message Box for Principal vanaers.
Creating new Message Box for Principal bmount.
Creating new Message Box for Principal swanson.
Creating new Message Box for Principal saul.

Now authenticate as a different principal and send a message.

dcelogin ericfin
Enter Password:
> message vanaers How do you do?
```

The server stores the message in the mbox memory structure for the principal vanaers. The principal vanaers can receive this message by authenticating and
calling message without an argument. After the message is received the server deletes the message from the message box of the principal.

Terminate the server by logging in as MessageBox again.

   > dcelogin MessageBox
   Enter Password:
   >
   > message -c shutdown
   >

The server exports the message entries to a file, unexports its interface and ends its execution.

   Requested to shutdown
   Exporting unread messages to file message.box.
6.3 DCE Security and RACF

As you have seen in the previous section, 6.1, “Concepts of DCE Security” on page 77, and the associated examples, DCE provides adequate security that is able to be extended to cover all aspects of an application’s security.

In an MVS/ESA system, security is provided by RACF or its equivalent. How does DCE security and RACF relate to each other? Can they co-exist? Can they complement each other?

RACF and DCE security haven’t been integrated yet. That integration is to be delivered later. In the meantime, we must deliver solutions that do not compromise the security of the MVS/ESA, or DCE and are consistent with the direction of the likely implementation of that integration.

The problems that have to be solved are:

- **Mapping DCE principals to RACF user IDs.** A DCE principal name can be up to 128 characters. A RACF user ID can be up to 8 characters.
- **Mapping DCE authorities to RACF authorities**
- **Implied and explicit login to DCE or RACF** when a user enters the system from either MVS or DCE.
- **Storing of encrypted DCE passwords** to enable implied login.
- **Use of MVS Authorized Program Facility (APF).**
- **Security of MVS HFS files**
- **Access to DCE servers from CICS applications**

The following sections will address these issues from two perspectives:

- A DCE application server on MVS
- A DCE client on MVS

**Note:**

The programs described here are not distributed with this book.
Table 3 makes a comparison between RACF facilities and the equivalent facility in DCE security.

<table>
<thead>
<tr>
<th></th>
<th>RACF</th>
<th>DCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Users</td>
<td>Defined as a user profile. Maximum user ID length is 8. Maximum password length is 8. Password rules apply.</td>
<td>Defined in DCE registry database as a principal. Maximum principal name length is 128. Maximum password length is 128.</td>
</tr>
<tr>
<td>Groups</td>
<td>Defined as a group profile. Maximum group ID length is 8</td>
<td>Defined in DCE registry database as a group. Maximum group name length is 128.</td>
</tr>
<tr>
<td>Organization</td>
<td>Not applicable</td>
<td>Defined in DCE registry database as an organization. Maximum organization name length is 128.</td>
</tr>
<tr>
<td>Dataset/file access</td>
<td>Defined by an explicit or generic dataset profile. Maximum dataset name length is 44. It is checked for access authority on data set open by MVS data management and RACF, transparent to the application. Access is controlled by an access list in the dataset profile containing users and groups and the associated access level.</td>
<td>Undefined when DCE/DFS is not used. A DCE application server can use the CDS name space and define a object that represents the dataset/file. It has to be explicitly checked by the application server code using DCE ACLs associated with the object. They contain a list of users and groups and the associated permission bits.</td>
</tr>
<tr>
<td>Program/transaction execution</td>
<td>A program in a private library can be controlled by a general resource profile in the PROGRAM class and execute access level in the dataset profile so that a program can be executed but not read. Transactions in CICS and IMS also use general resources in the appropriate class to determine whether a user is allowed to execute the transaction.</td>
<td>A server can be controlled by the ACL in the CDS name space that contains the exported bindings of the server. If a client does not have read access to the exported bindings of the server, he/she can not connect to that server.</td>
</tr>
<tr>
<td>General resource</td>
<td>An application can define an application specific class and profiles in that class that represent a logical function in the program.</td>
<td>A server can use an object in the CDS name space to represent a logical function in the program and can use the CDS's ACL manager to manage the meanings of the permission bits. A server can provide its own ACL manager to make the meanings of the permission bits specific to the application. However, the server has to be executing when the ACL_EDIT utility is executed to change the ACLs of the objects.</td>
</tr>
</tbody>
</table>

6.4 A DCE Application Server on MVS

There are several reasons that a DCE application server would be installed on an MVS system:

1. To use a highly available processor with adequate processing capacity,
2. To use data, such as databases and MVS data sets, stored and managed in the MVS system,
3. To use existing transaction processing systems such as CICS, IMS, APPC/MVS servers, and Message Queueing.

This section addresses the use of data stored and managed in the MVS system as follows:
- Application resources controlled by DCE security
- Application resources controlled by RACF

### 6.4.1 Application Resources Controlled by DCE Security

**Appearance of a DCE Server to RACF**

To RACF, a DCE application server appears as an ordinary RACF user, and is restricted to those MVS resources to which that RACF user has access. Those MVS resources may be data sets, or general resources (such as tape volumes, or application defined resources).

DCE security differs from the RACF philosophy in that the implementation of protection has to be built into the design of the application server rather than being external to the application. RACF can control whether a user is allowed to execute a program, gain access to a data set and at what level, from the RACF profiles external to the application. RACF can also enable an application to define resources that can represent processes within the application through RACF general resources.
Access to MVS/ESA OpenEdition DCE HFS files

A DCE application server can make use of files inside the HFS file system, and MVS data sets. By default, the HFS file system provides the local file system for MVS/ESA OpenEdition DCE. Each HFS file set is implemented as a new MVS file type of HFS. Multiple file sets are mounted to form a tree structure of files similar to that in use on UNIX and OS/2 systems. The control of access to files within the HFS file system is based on permission bits as specified in POSIX 1003.1.

Permission bits are associated with each file and are as follows:

- **Owner permissions** - rwx (read, write, execute)
- **Group permissions** - rwx
- **Other permissions** - rwx

The owner permissions apply to the person that created the file. Group permissions apply to the group to which the owner belongs. In a RACF system, each RACF group is assigned a group ID and the group ID recorded for the file is owner’s default group. The other permissions apply to anyone else.

To DCE security, the HFS files are files in the local file system. The next section discusses MVS data sets and applies to HFS files as well.

Access to MVS data sets

To DCE security, MVS data sets (and HFS files) are considered to be files in another local file system, and are outside the scope of DCE security control. However, the manner of use of these local files that the application has access to within the local file system, can be controlled by the DCE ACLs that represent that use.

A DCE application server is expected to use the DCE security to determine whether its client is:

- authenticated
- granted permission by name
- granted permission by being a member of a group
- granted permission by the ACL representing a resource and at what access level.

Access to RACF general resources

Any RACF application can use RACF general resources as a part of its processing to represent a logical function or access to portions of data sets/files to which the server access. However, even though the DCE clients are authenticated and their identity can be trusted, they are not known to RACF. All access checking in RACF is done based on the DCE application server’s RACF user ID. Because a normal DCE application server is running in MVS problem program state, it is not allowed to issue the privileged RACROUTE TYPE=CREATE request to identify a user to RACF. In the next section, these problems are discussed and possible solutions are suggested.
6.4.2 Application Resources Controlled by RACF

In the previous section, 6.1, “Concepts of DCE Security” on page 77, we saw some of the techniques available for a standard DCE server using standard DCE security.

In a typical MVS installation there is already a substantial user population that is governed by security policies implemented through RACF profiles. The introduction of DCE into these systems presents the installation with a need to implement their security in DCE as well as RACF to protect their resources, and maintain those policies in both environments. This also presents a RACF security auditor with a need to inspect and certify DCE application servers code to see that the policies have been implemented correctly.

In this section, we describe a technique that enables a DCE application server the use of RACF security based on a DCE principal’s associated RACF user ID without the DCE application server having to be an MVS authorized program (APF).

An MVS APF authorized program has to reside in extremely well protected, nominated MVS libraries and their contents monitored because an MVS APF authorized program can put itself into supervisor state, change its storage key and do anything it wants in the MVS system, intentionally or unintentionally, including bypassing any security.

To remove the potential of unacceptable exposure of the MVS system, the following design introduces an MVS user SVC (supervisor call). An SVC is given control in Supervisor state, storage protection key zero. Within that routine, it can issue the privileged parameters of the RACROUTE to logon a user onto RACF. To logon to RACF, however, the SVC has to have a means of finding a DCE client principal’s associated RACF user ID.

To map from a DCE principal name to a RACF user ID, we use the DCE PAC (privileged attribute certificate) that accompanies an authenticated DCE RPC. The PAC contains the UUIDs of the principal and the list of DCE groups that the principal belongs.

In the design we are presenting here we assume for each principal (user ID), a DCE group is created with a name of:

\[ \text{RACF\textunderscore sysid\textunderscore userid} \]

with the only member of that DCE group being the DCE principal associated with the RACF user ID. This group provides a mapping between the DCE principal and the user’s RACF user ID.
The following steps, shown in Figure 43, show the use of the installation’s user SVC.

1. An authenticated DCE client of a DCE application server issues an rpc_binding_set_auth_info request to indicate to the DCE run-time the name of the server, and the authentication service to be used. When the authentication service requested is rpc_c_authn_dce_secret or rpc_c_authn_default, a PAC (privilege attribute certificate) is built and forwarded with every RPC issued. These kind of RPCs are called authenticated RPCs. The PAC contains the UUIDs of the principal and a list of groups of which the principal is a member.

2. The DCE application server calls an assembler subroutine, RACF, to create a RACF ACEE (access control environment element) for the client’s RACF user ID. Because DCE security provides third party authentication of the client principal, we can treat it as a trusted peer and do not need to provide the RACF password.

The ACEE is chained from the MVS TCB and all subsequent requests to RACF through the RACROUTE macro for authorization or implied by the use of MVS services will be based on the client’s RACF user ID.

We do not rely on parameters passed by the caller of subroutine apart from the RPC’s binding handle. All other information needed, such as the client’s authentication level, certification, and the client’s PAC, are requested directly by the user SVC (through a C subroutine). In this way, the use of the SVC is restricted to only being able to logon a DCE client’s associated RACF user ID.
The RACF subroutine parameters depend on the type of call. To logon to RACF, and create an ACEE for a user, the parameters are as follows:

```
rc=RACF(<binding_handle>, "CREATE"," NULL, NULL, NULL)
```

where

- `<binding handle>` is the handle passed from DCE run-time for this RPC.
- `<request>` CREATE requests a logon to RACF.
- `<class>` ignored for a create request.
- `<entity>` ignored for a create request.
- `<requested access>` ignored for a create request.

`rc` is the return code of:

- 0 RACF logon OK, ACEE created
- 4 RACF user ID has been revoked
- 8 RACF user ID not known to RACF
- 12 client is not using an authenticated RPC
- 16 client does not have a RACF_<sysid>_userid group from which the RACF user ID can be extracted.

3. For a CREATE request, the RACF subroutine issues the user SVC, passing the type of request and the binding handle.

4. Among other functions that will be discussed in subsequent sections, for a CREATE request, the user SVC establishes the inherited C language enclave from the original caller, and calls the C function, GETUSER.

5. The GETUSER function is passed the binding handle. Using that binding handle, it can make requests to DCE to determine details about the client principal. To protect the integrity of MVS, we do not use any information provided by the caller of the SVC apart from the binding handle which is verified by DCE.

GETUSER will be passing the name of the client principal and his/her RACF user ID back to the user SVC, as follows:

- a. The get_PAC routine issues rpc_binding_inq_auth_client to get the client’s credentials in the PAC. The PAC contains the UUID o the client principal and a list of UUIDs of groups of which the principal is a member. The PAC is provided by the DCE run-time and was part of the original RPC call from the client.

- b. The DCE application server calls FINDUSER, to search the client’s PAC and for each group UUID asks the DCE registry to provide the name of the group until a group name of RACF_sysid_ is found. The RAF user ID is extracted from the group name. The sysid is extrated from the MVS CVT and allows a principal to have different RACF user ID’s on different MVS systems. If there is no RACF user ID, an error code is returned.

6. On return from the GETUSER function,

- a. The user SVC issues RACROUTE REQUEST=VERIFY,ENVR=CREATE to create the ACEE. The SVC can use the NOPASSWD parameter, because the user has been authenticated by a trusted system, DCE. The ACEE is chained of the caller’s TCB (the manager).
b. Now that the client’s RACF ACEE has been built, we check that the principal name stored in the ACEEINST field is the same as the principal name retrieved from DCE (see 6.5.1, “Encrypted DCE Password in RACF Database” on page 114 for a description of the ACEEINST field).

7. Once the RACF environment has been established in the previous step, all access to MVS resources is determined by that RACF user ID or the RACF groups to which he/she belongs. Opening a MVS data set, or allocating a new MVS data set will be allowed or denied through requests by the MVS data management routines to RACF.

If the DCE application server wants to check the authority to use a RACF resource (dataset, tapevol, general resources, and so on), the parameters are as follows:

```
rc=RACF(<binding_handle>, "AUTH," <class>, <entity>, <access requested>)
```

where

- `<binding_handle>` is the DCE server’s binding handle
- `<request>` AUTH requests a authority check
- `<class>` is the class of RACF entity (for example, DATASET)
- `<entity>` is the RACF entity being requested
- `<access requested>` is the access requested (for example, read, write, control, alter).

`rc` is the return code of:

- 0 access granted
- 8 access denied
- 12 resource not known
- 16 Don’t know

8. For an AUTH request, the RACF assembler subroutine does not need to use the SVC. It issues a RACROUTE REQUEST=AUTH using the parameters passed from the caller.

9. The DCE application server must call the subroutine, RACF, to delete the ACEE for user before completing processing. The parameters are as follows:

```
rc=RACROUTE(<binding_handle>, "DELETE," NULL, NULL, NULL)
```

where

- `<binding_handle>` is the DCE server’s binding handle
- `<request>` DELETE requests deletion of the ACEE.
- `<class>` ignored for a delete request.
- `<entity>` ignored for a delete request.
- `<requested access>` ignored for a delete request.

`rc` is the return code of:

- 0 deletion successful
- 8 not logged on

10. For a DELETE, the subroutine, RACF, issues the user SVC requesting that the RACF user ID be logged off.

11. The user SVC issues a RACROUTE REQUEST=VERIFY, ENVR=DELETE.
6.5 A DCE client on MVS

The following lists the probable environments in which DCE clients can work, or would like to work:

- TSO
- OMVS client
- Batch
- CICS
- IMS
- APPC

TSO, OMVS, and batch environments are the most readily useable for DCE clients. TSO, or OMVS users log onto TSO and then issues a DCELOGIN (or DCE_LOGIN) to which they provides their DCE password in clear. Their DCE login context is created and held in the EUVSKRB5 cache data set. Likewise, a batch job is submitted from TSO, and one step executes the DCELOGIN passing the principal and DCE password in the PARM parameter.

However, what about a TSO user who executes a program that becomes a DCE client under the covers? Do all DCE client applications have to assume that the user has logged in to DCE?

What about a CICS application that needs to become a DCE client to a DCE application on a UNIX system? This is the converse of the MVS/ESA OpenEdition DCE optional features, CICS and IMS Application Support Servers, which only provide access to legacy transactions in CICS and IMS from DCE clients on the DCE network.

CICS has the added complication that it is executed in MVS as a single task (TCB) and provides its own dispatcher. As a result, a CICS transaction can never WAIT on an MVS event or, in DCE terms, block the task until the request has been executed. Such activity would put the whole CICS region into a WAIT at the expense of all other users in that CICS region.

The following Figure 44 on page 113 describes a technique that uses APPC/MVS as a tailored proxy or gateway that allows a CICS application to become a DCE client to a DCE server that may be on a UNIX system.

It is also used here to illustrate a technique that will allow an installation to:

1. Login a DCE user on the basis that his identity has been authenticated by RACF.
2. Provide the DCELOGIN with a clear password even though the user cannot readily respond to a prompt for the password (the CICS user who is unaware that the CICS transaction is talking to a DCE server).

The technique is applicable to all the DCE client environments described above.
The following points describe Figure 44.

1. The intent of the CICS transaction is to become a client to the DCE server shown in the diagram as point 5.

   A CICS user has a RACF ACEE created when they logon to CICS. All access to resources, including the execution of transactions, are determined by RACF using the user ID in this ACEE. The transaction executes a request to invoke an APPC/MVS transaction program and passes on the authenticated user ID.

2. During the initialization of CICS, the connection between CICS and APPC/MVS is established as trusted partners. When APPC/MVS receives the connect request to establish a session, it determines that it is to be scheduled to be executed in the dependent address space, ASCH.

3. The scheduler in the dependent address space, ASCH, issues a MVS RACROUTE to establish the RACF ACEE for the requesting user. The transaction program is executed as a normal non-APF program.

   The transaction program is the APPC server requested by the CICS transaction. The APPC server receives the request’s data through one or more receive and sends.

4. The APPC server now wants to login to DCE as a client so that the request can be satisfied by the DCE server after which the data can be forwarded back to the original CICS transaction.

   Now the problem becomes apparent! What is the DCE principal to be used?
• What is its DCE password?
• Can we communicate back to the original CICS transaction to prompt for the principal name and DCE password?
• Do we use the DCE keyfile containing principal names and their encrypted DCE password?
• If we do use a keytab file, how do we associate a RACF user ID and a DCE principal name?
• If we do use a keytab file, the RACF user ID must have read access to the file and as such has access to the encrypted passwords of all other users of this APPC server.

The suggested solution is described in the following section, 6.5.1, "Encrypted DCE Password in RACF Database," but for completeness of this example, the use is shown here (not shown in Figure 44 on page 113).

a. The client’s ACEE contains an installation user data field, ACEEINST, which the RACF administrator has entered as:
   
   DCE(<principal name>://)

b. The DCE principal updates this field to enter the encrypted DCE password and version number, by using the SAVEPW command described in 6.5.2, “SAVEPW Command” on page 118:
   
   DCE(<principal name>/<encrypted DCE password>/<version number>)

c. The routine DCEPRIN extracts the principal name from the current ACEE, ACEEINST field.

d. The routine DCEPASS uses the installation’s user SVC described later, to extract the clear DCE password from the current ACEE, ACEEINST field.

The APPC server can now login to DCE as the nominated principal using his/her password and call the DCE server.

5. The DCE server processes the request according to the client’s PAC and returns the result, and in turn, the APPC server returns the result back to the original CICS transaction.

6.5.1 Encrypted DCE Password in RACF Database

As discussed in the previous section, the RACF database can be used to store the DCE principal name, their encrypted DCE password, and the version of the system secret key in the RACF user profile user data field. This section and the following sections describe:

1. An assembler routine, DCEPRIN that will return the principal name from the current user’s ACEE.

2. An assembler routine, DCEPASS that will return the clear password from the current user’s ACEE.

3. A user SVC that decrypts the user’s password in the current user’s ACEE or encrypts it to be placed in the current user’s ACEE.

4. The SAVEPW command (see 6.5.2, “SAVEPW Command” on page 118), that the DCE principal has to execute to store the encrypted DCE password in the user’s RACF user profile, user data field.

5. 6.5.3, “DCE System Secret Key Utility” on page 122 creates a system secret key under which the user’s user ID and password are encrypted.
The technique described here has made the following assumptions:

- The DCE password and the RACF password may be different.
- The DCE password does not contain the ‘/’ (forward slash) character.
- The synchronization of password changes on DCE and RACF is done by the user.
- The installation has not used the user data field for other purposes.

To set up a user:

1. The RACF administrator enters the user data in the RACF command that changes the content of the user's profile (or the equivalent on the ISPF panels).

   ALTUSER <userid> DATA('DCE(<principal name>/)//')

   The DCE “principal name” is included to allow the RACF user ID and the DCE principal name to be different. The two / (slash) characters are used as field separators which at this stage are null fields.

2. The RACF administrator also has to create a new RACF general resource class of DCEKEY. This class is used to store the current system secret key and previous versions of it. These profiles must specify UACC (universal access authority) of NONE and have nothing in the access lists. The system secret key will be used to encrypt the user ID to form an encryption pad when encrypting/decrypting the user’s password in the user’s RACF user profile (see 6.5.3, “DCE System Secret Key Utility” on page 122 for more details).

3. The DCE principal has to login to DCE and execute the SAVEPW command (see 6.5.2, “SAVEPW Command” on page 118), initially or whenever he/she changes the DCE password, to store the encrypted DCE password in the user’s RACF user profile, user data field.

   DCE(<principal name>/<encrypted DCE password>/<version number>)

   Login to DCE can be done in a TSO session, or on a non-MVS system in the DCE cell. For users that do not have access to either, such as CICS and IMS users that have not been given a TSO session by the installation, the DCE security administrator will have to issue the SAVEPW command on their behalf. Once the initial SAVEPW command has been issued, however, a CICS transaction could be written (using APPC/MVS), or an IMS transaction, that will allow the user to login to DCE, change the DCE password, and issue the SAVEPW as a client.

Figure 45 on page 116 illustrates the process:
1. The application program that wants to issue a login to DCE calls the assembler routine DCEPRIN, passing it a work area that will contain the user’s principal name on return. This can be a maximum of 128 bytes. **DCEPRIN:**
   a. finds the current ACEE from either the current TCB (TCBENV not zero), or from the ASXB (ASXBENV).
   b. extracts the principal name from the ACEE ACEEINST field.

2. The application program then calls the assembler routine DCEPASS, passing it a work area that will contain the user’s clear password on return. The password routine executes a user SVC, whose number will have to be decided by the installation but for ease of reference in this document is shown as usersvc.

3. The user SVC, a fragment of which is shown as pseudo code in the diagram, does the following:
   a. If the current ACEE does not have the DCE data in the ACEEINST field, return to the caller with an error code. Otherwise, Issue the RACROUTE EXTRACT retrieve the system secret key (this macro is available to authorized callers only):

```
RACROUTE REQUEST=EXTRACT,
  TYPE=EXTRACT,
  CLASS=DCEKEY,
  ENTITY=ENTITY,
  BRANCH=YES,
  FIELDS=FIELDS,
  SEGMENT=BASE,
```

---

**Figure 45. Encrypted DCE Password in RACF Database**

1. The application program that wants to issue a login to DCE calls the assembler routine DCEPRIN, passing it a work area that will contain the user’s principal name on return. This can be a maximum of 128 bytes. **DCEPRIN:**
   a. finds the current ACEE from either the current TCB (TCBENV not zero), or from the ASXB (ASXBENV).
   b. extracts the principal name from the ACEE ACEEINST field.

2. The application program then calls the assembler routine DCEPASS, passing it a work area that will contain the user’s clear password on return. The password routine executes a user SVC, whose number will have to be decided by the installation but for ease of reference in this document is shown as usersvc.

3. The user SVC, a fragment of which is shown as pseudo code in the diagram, does the following:
   a. If the current ACEE does not have the DCE data in the ACEEINST field, return to the caller with an error code. Otherwise, Issue the RACROUTE EXTRACT retrieve the system secret key (this macro is available to authorized callers only):
Note:

Because the SVC is re-entrant, the data areas will be in dynamic storage and will be built by code not shown here. The <sysid> would be extracted from the MVS CVT.

RACF returns the address of the results in register 1. If the profile is not found, the SVC will return to the caller with an error code.

Not shown in the diagram for sake of simplicity, if the version of the system secret key does not match the version in the user's ACEE user data field, the RACROUTE is re-issued using an ENTITY of '<sysid>.Vnnn' to retrieve the appropriate key. When the password has been extracted and is in clear, the user's ACEE user data field is updated using the current system secret key as the encryption pad and the user's RACF profile in the RACF database is updated.

b. The SVC then issues a RACROUTE TYPE=ENCRYPT to encrypt the current user ID in the ACEE under the system secret key to be used as an encryption pad, as follows:

```
RACROUTE REQUEST=EXTRACT,
    TYPE=ENCRYPT,
    ENTITY=PAD,
    ENCRYPT=(SECRET,STDDES),
    RELEASE=1.9.2,
    BRANCH=YES
```

```
INSTDATA DS A(*-*) length of result
    DS CL256'<key>/<version>' key and version string
SECENT DS AL1(8)
    DS CL8'<key>' copied from INSTDATA
PAD DS CL8'<userid>'
```

Note:

Because the SVC is re-entrant, the data areas will be in dynamic storage and will be built by code not shown here.

The PAD, which had been set to the current user ID, has now been encrypted under the system secret key and replaced.

4. Because the SVC was requested to decrypt the user's password, we now jump into the "else" code. The SVC takes the encrypted password and
exclusive or’s the PAD generated in the previous step with it. This will expose the user’s clear password and the result is passed back through the calling sequence to the DCE application.

### 6.5.2 SAVEPW Command

The SAVEPW command is a TSO CLIST that invokes the DCE client program, SAVEPW, to store a DCE principal’s DCE password in his/her associated RACF user profile. The command will only store the currently logged in principal’s password in the associated RACF user ID. To ensure that the DCE password in the DCE registry and the DCE password in the user’s RACF password match, both are updated. The following is the syntax of the command:

```
SAVEPW <sysid> <password> <verify password>
```

where

- `<sysid>` is the name of the MVS system to be updated.
- `<password>` is the DCE principal’s password.
- `<verify password>` is the DCE principal’s password re-entered for verification.

Figure 46 illustrates the processing that is done by the SAVEPW command. The fragments of code shown are pseudo code. The fragment of the user SVC is another part of the user SVC that was discussed in the previous section.
1. SAVEPWC is the client portion of the SAVEPW DCE application server. It has three parameters passed to it by the user, <sysid>, <password>, and <verification password>. The SAVEPW application server is replicated on every MVS/RACF system in the DCE cell and provides a RACF user profile updating capability on each RACF system. The server requires the use of authenticated RPCs.

   a. If the <password> and the <verification password> match, the client program, SAVEPWC, must first find the exported bindings for the requested RACF system <sysid>. The exported name in the CDS name space is RACF_<sysid>.

   b. If the bindings are found, bind to the requested RACF system.

   c. The client program then calls SAVEPW passing it the following parameters:

      \[
      rc=\text{SAVEPW}(bh, \text{password})
      \]

      where

      \[
      \begin{align*}
      bh & \quad \text{is the explicit binding handle of the request} \\
      \text{password} & \quad \text{is the password to be stored in the client principal’s associated RACF user profile.}
      \end{align*}
      \]

2. The manager code for SAVEPW calls the following routines:

   a. get_PAC, which gets the client’s PAC as described previously in 5a on page 110.

   b. FINDUSER, which finds the client’s associated RACF user ID, as described previously in 5b on page 110.

   c. RACUPDT, which is an assembler routine that will update the user’s RACF profile.

3. The RACUPDT routine uses privileged parameters of the RACROUTE MVS macro, must reside in an MVS APF authorized library and the SAVEPWS module must be linked as authorized. The routine performs the following:

   a. Issue the RACROUTE macro to extract the client’s RACF user profile, INSTDATA field. This field has been setup by the RACF administrator initially to contain:

      \[
      \text{DCE(<principal name>//})
      \]

      or on subsequent updates,

      \[
      \text{DCE(<principal name>/<encrypted DCE password>/<key version>)}
      \]

      The user’s RACF profile user data field in the RACF database is extracted using RACROUTE REQUEST=EXTRACT,TYPE=REPLACE as follows:

      \[
      \begin{align*}
      \text{RACROUTE} & \quad \text{REQUEST=EXTRACT,} \\
      \text{TYPE=EXTRACT,} & \quad \text{CLASS=USER,} \\
      \text{ENTITY=ENTITY,} & \quad \text{FIELDS=FIELDS,} \\
      \text{SEGMENT=BASE,} & \quad \text{RELEASE=1.9.2,} \\
      \text{BRANCH=YES} & \quad \text{MVC INSTDATA(4),0(R1)} \\
      \text{MVC INSTDATA+4(256),4(R1)}
      \end{align*}
      \]
b. If the client’s principal name matches the principal name that the RACF administrator entered in the user’s RACF user profile, issue the user SVC to request encryption of the password. The encrypted password and the version of the system secret key will be returned in the workarea.

4. For an encryption request, the user SVC tests, using TESTAUTH, whether the caller is APF authorized and returns with an error code if it fails. Otherwise, it continues as follows:

a. Issue a RACROUTE REQUEST=EXTRACT to retrieve the current system secret key, as follows:

```
RACROUTE REQUEST=EXTRACT,
  TYPE=EXTRACT,
  CLASS=DCEKEY,
  ENTITY=ENTITY,
  FIELDS=FIELDS,
  SEGMENT=BASE,
  RELEASE=1.9.2,
  BRANCH=YES
```

```
MVC INSTDATA(4),0(R1)
MVC INSTDATA+4(256),4(R1)
```

b. The SVC then issues a RACROUTE TYPE=ENCRYPT to encrypt the RACF user ID passed in the workarea under the system secret key to be used as an encryption pad, as follows:
RACROUTE REQUEST=EXTRACT,
TYPE=ENCRYPT,
ENTITY=PAD,
ENCRYPT=(SECRET,STDDES),
RELEASE=1.9.2,
BRANCH=YES

..................................
SECRET DS AL1(1)
   DS CL8'<secret key>'
PAD   DS CL8'<userid>'

Note:
Because the SVC is re-entrant, the data areas will be in dynamic
storage and will be built by code not shown here.

The PAD, which had been set to the current user ID, has now been
encrypted under the system secret key and replaced.

c. Exclusive OR the encrypted user ID with the password
d. Put the encrypted password and the version of the system secret key in
   the workarea provided by the caller.

5. On return from the user SVC, the RACUPDT routine then has to update the
   user’s RACF profile user data field in the RACF database using RACROUTE
   REQUEST=EXTRACT,TYPE=REPLACE as follows:

   RACROUTE REQUEST=EXTRACT,
   TYPE=REPLACE,
   CLASS=USER,
   ENTITY=ENTITY,
   FIELDS=FIELDS,
   SEGDATA=SEGDATA,
   SEGMENT=BASE,
   RELEASE=1.9.2,
   BRANCH=YES

..................................
USER DS CL8'USER'
ENTITY DS CL8'<userid>'
FIELDS DS A(1)
   DS CL8'INSTDATA'
SEGDATA DS A(nnn)
   DS CLnnn'DCE(<principal name>/<e(password)>/<version>)'
BASE DS CL8'BASE'

Note: Because the SVC is re-entrant, the data areas will be in dynamic
storage and will be built by code not shown here.

6. On return to the SAVEPW client program from the SAVEPW server, the client’s
   principal password is updated in the DCE registry database with the
   successfully saved DCE password to ensure that the passwords match, using
   sec_rgy_acct_passwd and associated DCE APIs.
6.5.3 DCE System Secret Key Utility

The DCE System Secret Key utility, DCEKEY, is used to create a secret key to be used to encrypt a user’s password. The utility is MVS APF authorized and has to be placed in an authorized library, so that it can issue privileged parameters on the RACF RACROUTE macro.

The key that is generated is chosen using a random number generator and encrypting that number. That key is stored in the user data field of a RACF general resource in the a new class of DCEKEY. The naming convention for the profiles is:

    <sysid>.CURRENT | Vnnn

where:

- `<sysid>` is the system ID as stored in the MVS CVT of this system.
- `CURRENT` or `Vnnn` identifies the key as being the current key or the version number of the system secret key.

The utility assumes the following:

- The RACF administrator has created a RACF class of DCEKEY.
- The RACF administrator has created a pool of general resource profiles in that class conforming to the naming convention above. The minimum number of profiles is two; one for CURRENT and one for the version number copy of that profile. There should be at least one more version number profile than the number of times the utility has been executed.

The following Figure 47 on page 123 illustrates the processing performed by the utility shown is pseudo code.
1. Retrieve the CURRENT profile using RACROUTE REQUEST=EXTRACT to get the key and version number of the key.

   RACROUTE REQUEST=EXTRACT,
   TYPE=EXTRACT,
   CLASS=DCEKEY,
   ENTITY=ENTITY,
   FIELDS=FIELDS,
   SEGMENT=BASE,
   RELEASE=1.9.2,
   BRANCH=YES

   MVC INSTDATA(4),0(R1)
   MVC INSTDATA+4(256),4(R1)

   DCEKEY DS CL8'DCEKEY'
   ENTITY DS CL8'<sysid>.CURRENT'
   FIELDS DS A(1)
   BASE DS CL8'BASE'
   INSTDATA DS A(*-*):
   DS CL256'<key>/<version>'

2. If the user data field is not present, set the version number to 0.

3. Increment the version number.

4. Generate a random number and encrypt the system-id with it using RACROUTE as follows:
RACROUTE REQUEST=EXTRACT,
TYPE=ENCRIPT,
ENTITY=KEY,
ENCRIPT=(SEED,STDDES),
RELEASE=1.9.2

..................................

SEED DS AL1(8)
DS CL8’<random number>’
KEY DS CL8’<sysid>’

5. Update the ‘<sysid>.CURRENT’ profile using RACROUTE as follows:

RACROUTE REQUEST=EXTRACT,
TYPE=REPLACE,
CLASS=DCEKEY,
ENTITY=ENTITY,
FIELDS=FIELDS,
SEGDATA=SEGDATA,
SEGMENT=BASE,
RELEASE=1.9.2

..................................

DCEKEY DS CL8’DCEKEY’
ENTITY DS CL8’<sysid>.CURRENT’
FIELDS DS A(1)
DS CL8’INSTDATA’
SEGDATA DS A(nnn)
DS CLnnn’<key>/<version>)’
BASE DS CL8’BASE’

6. Update the ‘<sysid>.Vnnn’ profile using RACROUTE as follows:

RACROUTE REQUEST=EXTRACT,
TYPE=REPLACE,
CLASS=DCEKEY,
ENTITY=ENTITY,
FIELDS=FIELDS,
SEGDATA=SEGDATA,
SEGMENT=BASE,
RELEASE=1.9.2

..................................

DCEKEY DS CL8’DCEKEY’
ENTITY DS CL8’<sysid>.Vnnn’
FIELDS DS A(1)
DS CL8’INSTDATA’
SEGDATA DS A(nnn)
DS CLnnn’<key>/<version>)’
BASE DS CL8’BASE’
Chapter 7. Threads and Data

This chapter discusses two topics that need to be understood before writing complex applications. These topics are threads and data handling. Threads and data handling are, of course, topics independent of DCE but there are some significant usages in DCE of these technologies that application developers must be aware.

7.1 Threads

A thread is a single, sequential flow of control within a process. An environment with multiple concurrent flows of control within a process is said to be a multi-threaded environment. It is common to see threads in a multi-threaded environment be referred to as lightweight processes. The terminology comes from the UNIX community and may seem a little strange to a long time MVS programmer. But a process roughly correlates to an MVS address space and a thread to a TCB.

DCE threads is a user level interface that gives C programs the ability to create and manipulate threads. DCE threads is an optional part of any specific platform implementation of DCE. This is not because threads aren’t needed for proper DCE operation but because many operating systems already have threads available. MVS/ESA has implemented threads as part of the base system. The MVS/ESA OpenEdition implementation is based on the POSIX standards as is the DCE implementation.

Threads allow multiple tasks to run in a single process/address space. A thread can be thought as a TCB within an address space. It allows for concurrent and asynchronous processing, without the additional overhead associated with creating a new address space. The creation of threads allows DCE and the application programmer to create independent units of work that can share the same storage. Threads can be categorized into three types, heavy, medium and light weight. OpenEdition MVS’s implementation of a thread is to attach a TCB within a process (address space).

- **Heavy Weight Threads**
  A heavy thread has been defined as a task that is attached when needed. When a heavy weight thread terminates, the task (TCB) that supports it is terminated and all end of tasks resource managers are called to clean up. The main reason for a heavy thread is to have complete clean up of resources at termination time.
  Heavy weight threads are managed by OpenEdition MVS

- **Medium Weight Threads**
  Medium weight threads reuse MVS tasks. When a medium weight thread is created, it is dispatched on an MVS task that is maintained in a pool. When this type of tasks terminates normally, the MVS task is recycled in the pool, without going through end of tasks resource managers. This means that the task itself must cleanup its resources.
  Medium weight threads are managed by OpenEdition MVS
Application without Threading

![Diagram of Application without Threading]

Application using Threads

![Diagram of Application using Threads]

Figure 48. Processor Sharing

The upper half of Figure 48 shows the traditional way an application runs. There are two jobs, each consisting of two tasks with a wait to input. When the first job/task completes the second job/task will start. Since the program is in “single thread” mode the jobs/tasks run sequentially. In the lower part of Figure 48 a multi-threaded (two threads) program can be seen. While the first job/task waits for input the second job/task gets control and the first part of it gets started.

In dealing with client/server applications written for a multi-access environment you must ensure that your code allows for multi-access. That the code is reentrant and external and internal resources that are required are accessed via locks/semaphores (thread-safe). Figure 49 on page 127 gives a general view of a single DCE server that handles three clients. The manager code will be run as separate threads within the same address space. Due to this environment your DCE application manager code must be written to be thread-safe.
Multi-threaded programs can share internal and external data. This offers advantages in parallel programming. It also means that everything that is shared by threads must be protected by locks or semaphores in order to serialize access. A mutual exclusion object (mutex) is an object that is used in programming within multiple threads to ensure the integrity of shared resources.

A mutex insures that, at any point in time, only one thread can access the resource with which the mutex is associated. Mutexes are generally held for short durations in critical paths.

A mutex can be in three states:
- uninitialized
- locked
- unlocked

All threads within a DCE environment when accessing the same piece of storage or resource must use the same mutex. Each thread locks the mutex before it accesses the shared data or resource and unlocks it when it has finished access the data or resource. In a DCE shared storage must be allocated by the server code. Once it is allocated by the server, it can be accessed by the manager code. However the manager code must use a mutex in order to ensure synchronization.
7.2 Data Handling

Proper data and type handling is important in any application. Special attention needs to be given to some of the data requirements of distributed systems. DCE services will marshal and unmarshal data as the data is passed from client to server and from server back to the client. This is based on the interface definition language (IDL) descriptions.

The IDL is used to specify the interface between the client and the server in a DCE environment. In section Chapter 2, “Development Process and Interface Development” on page 25 we discussed the mechanics of using IDL. There is actually a lot more to IDL than what was discussed in Chapter 2.

IDL was designed to look very much like C data declarations. A knowledge of C is assumed of anyone using IDL. IDL has extended some existing data types in C. It has also created new data types, for example, boolean. Other data types are supported only with modifications or restrictions. For example, C unions must be discriminate and all arrays must be accompanied by bounds information.

The data types provided by IDL are:

- **Basic data types**
  - integer
  - floating-point
  - char
  - boolean
  - byte
  - void
  - handle_t
  - error_status_t

- **Constructed data types**
  - Structures
  - Unions
  - Enumerations
  - Pipes
  - Arrays
  - Strings

Many of the basic data types, for example integers, floating point, and char look like C data declarations. In C, a data type can map to different sizes on different systems. For example a long data type in C may be 16, 32, or 64 bits, depending on the system. This is, of course, completely unacceptable in a DCE environment where we want heterogeneous systems to interoperate. The size of an IDL data type must be the same on all systems. DCE resolves this conflict by providing a number of typedef statements that relate the DCE “type” to local system C “type” (See Guide to Writing DCE applications by John Shirly for a good treatment of this topic).

Reference OpenEdition Distributed Computing Environment Base Services MVS: Application Development Guide (SC09-1484) for complete information on the basic and constructed data types.

The following data types are of specific interest:
• **Byte**

The byte data type is represented as 8 bits. The data representation format is guaranteed not to change when data is transmitted by the RPC mechanism. This is useful when you are transmitting between equivalent machines. This becomes useful if you need to integrate your C RPC code with code written outside of C.

• **Arrays**

There are three types of arrays that you can use in IDL: fixed, varying, and conformant. Fixed arrays are like C arrays and have their size fixed at compile time. Varying arrays also have their length fixed at compile time but only a portion of the array is sent over the network. Conformant arrays have their size determined at execution time within the structure definition.

• **Pointer**

A pointer is a variable containing the address of another variable or data structure. In a distributed application, the client and server do not share the same address space. This means the data a pointer refers to in the client is not available in the remote server. Therefore, pointer data is copied between the client and server address space during a RPC. IDL has two kinds of pointers: reference pointers and full pointers. A reference pointer is used to refer to existing data and has less overhead. A full pointer has the usual capabilities associated with C pointers but has additional overhead for dynamic storage allocation for marshalling and unmarshalling parameters.

• **Unions.**

A union is a data structure that stores different types and sizes of data in the same area of memory.

We will cover arrays and unions in more detail in the following sections.

### 7.2.1 Conformant Arrays

Many times applications that use arrays do not know how big the array is at compile time. It is something that is determined at execution time. Many times in local processing the application logic is responsible for handling the varying lengths of arrays. This won't work in a distributed client/server environment.

The parameters need to be marshalled and unmarshalled. DCE uses conformant arrays to handle this these situations. The size of a conformant array is determined at run-time. The basic technique is to use another variable to specify the length of the array. This other variable must either be another parameter of the operation or a member of a structure of which the array is also a member. This other variable is specified with the `max_is` or the `size_is` attribute. `max_is` specifies the last element in the array and `size_is` specifies the number of elements in the array.

There can be only one conformant array within a structure definition. We have categorized them into type 1 and type 2 conformant arrays. A basic layout of type 1 is shown in Figure 50 on page 130.
In Figure 50 size_n in both case A and B identify the size of the array. The variable data_e is the conformant array. The value of size_n is set by the manager code. The number of elements in data_e dependant on the value of size_n.

The example in Figure 51 on page 131 is the IDL for a conformant array classified as type 1. It uses the size_is attribute to determine the size of the array. In this example we will invoking the remote procedure ExampA_rpc. It has two parameters: a binding handle and a conformant array.
[ pointer_default(ptr) ]

interface EXA
{
  typedef struct
  {
    long int size_n;
    [size_is(size_n)] long data_e[];
  } LIST;

  void ExampA_rpc (
      [in] handle_t h,
      [in,out] LIST **pp_list
  );
}

Figure 51. Conformant Array/IDL - Example A with Long int - Type 1

Figure 52 shows a segment of the IDL compiler generated header.

```c
#ifndef exa_v1_0_included
#define exa_v1_0_included
.
.
typedef struct {
  idl_long_int size_n;
  idl_long_int date_a[1];
} LIST;

extern void ExampA_rpc ( 
  #ifdef IDL_PROTOTYPES
  /* [in] */ handle_t h,
  /* [in, out] */ LIST **pp_list
  #endif
);

typedef struct exa_v1_0_epv_t { 
  void (*ExampA_rpc)( 
    #ifdef IDL_PROTOTYPES
    /* [in] */ handle_t h,
    /* [in, out] */ LIST **pp_list
    #endif
  );
} exa_v1_0_epv_t;

extern rpc_if_handle_t exa_v1_0_c_ifspec;
extern rpc_if_handle_t exa_v1_0_s_ifspec;
```

Figure 52. Conformant Array - IDL Compiled/Header File - Type 1
The following skeleton code on page Figure 53 on page 132 is extracted from a client/manager code. It uses the IDL on Figure 51 as the conformant array header.

```c
#include "exampa.h"

int main(int ac, char*av[])
{
    /* define as a struct pointer*/
    LIST *plist
    
    /* set pointer to NULL */
    plist = NULL;
    
    /* pass using reference of */
    ExampA_rpc(h,&plist); ---call to ---> void ExampA_rpc(handle_t h, LIST **pp_list )
    
    
    /* allocate 10 entries */
    /* for conformant array */
    *pp_list = rpc_ss_allocate
    ( size_n =
    sizeof(LIST) +
    (9 * sizeof(idl_long_int)
    );
}
```

Figure 53. Client/Server Execution

The skeleton code on page Figure 53 shows you how the manager code allocates storage. The DCE rpc_ss_allocate is passed the amount of storage to retrieve. The returning value is an address to a RPC GETMAINed area (we will discuss rpc_ss_allocate further in a later section of this chapter). The amount requested is based on the size of the conformant array structure plus the number of entries needed, times the size of the conformant array data type (idl_long_int is the C code data type defined by DCE to represent IDL data type long). The conformant structure generated by the IDL compiler already has an array of one element, so the new memory allocated for the array elements is one less than the number in the array (in this case 9).

The example on page Figure 54 on page 133 is another example of a type 1 conformant array using pointers. The number of pointers will be dynamic. The data area that the pointers access will be allocated/retrieved at run time by the manager code. See Figure 50 on page 130, example B for a general storage overview.
interface EXB {
    typedef struct {
        long int size_n;
        [size_is(size_n)] char *data_e[];
    } LIST;

    void ExampB_rpc (]
    [in] handle_t h,
    [in,out] LIST **pp_list
    );

}

Figure 54. Conformant Array/IDL - Example B with char Pointer

#ifndef EXB_v1_0_included
#define EXB_v1_0_included
.

typedef struct {
    idl_long_int size_rtn;
    idl_char *data_e[1];
} PTR_LIST;

extern void ExampB_rpc(
    #ifdef IDL_PROTOTYPES
    /* [in] */ handle_t h,
    /* [in, out] */ LIST **pp_list
    #endif
    );

typedef struct EXB_v1_0_epv_t {
    void (*ExampB_rpc)(
        #ifdef IDL_PROTOTYPES
        /* [in] */ handle_t h,
        /* [in, out] */ LIST **pp_list
        #endif
    );
} EXB_v1_0_epv_t;

extern rpc_if_handle_t EXB_v1_0_c_ifspec;
extern rpc_if_handle_t EXB_v1_0_s_ifspec;
.
.
Figure 55. Conformant Array - IDL Compiled/Header File

Chapter 7. Threads and Data 133
Client Manager - Server

```c
#include "exampb.h"

int main(int ac, char*av[]) {

    /* define as a struct pointer*/
    LIST *plist;

    /* set pointer to NULL */
    plist = NULL;

    /* pass using reference of */
    ExampB_rpc(h,&plist); --call to ---> void ExampB_rpc(handle_t h, LIST **pp_list)

    { 

    LIST *p_list1;

    } /* allocate 4 entries */
    /* for conformant array */
    *pp_list = rpc_ss_allocate
    (sizeof(LIST) +
     (3 * sizeof(idl_char))
    )

    /* map pointer not to use */
    pos = 0; /* start at 0 */
    p_list1 = *pp_rex_list;

    /* get storage for */
    /* set pointer(0,1,2) = */
    /* LOOP for 0,1,2,3 ) = */
    p_list1->data_e[pos] =
     rpc_ss_allocate(30);

    } /* allocate 4 entries */
    /* for conformant array */
```

Figure 56. Client/Server Execution

When using conformant arrays you can vary the amount of data to be passed between client and manager by decreasing the variable that indicates the size of the array. This will allow you to pass a subset of the array. You can also increase the size of the conformant array by allocating a new instance of the conformant array. Once the new instance has been allocated you will need to copy the old data and free its storage.
Figure 57 depicts a type 2 conformant array. The IDL is depicted on page Figure 58 on page 136. The example in Figure 58 on page 136 is a simple IDL conformant array of type 2. It has a set structure called ColInfo. Within ColInfo are reference pointers to a varying lengths of data. Element one (colWidth) is a pointer to a variable number of short integers. The number of elements within colWidth is based on Ncols. Element two (colWidth2) is a pointer to a variable number of long integers. The number of elements within colWidth2 is based on Mcols. The third element (colLabel) is a pointer to a variable number of char pointers. The number of elements within colLabel is based on Ncols. Type 2 is a fixed structure type of pointers. The vary number is used to indicate the number of items/rows that the pointed to structure contains.

Conformant array data types can be any IDL data types (integer, pointers structures, unions and so on).
[pointer_default(ptr)]

interface Pointer1
{

typedef [ptr,string] char *string;

typedef struct ColInfo {
    short int mcols;
    short int ncols;
    [ptr, size_is(ncols)] unsigned short *colWidth;
    [ptr, size_is(mcols)] unsigned long *colWidth2;
    [ptr, size_is(ncols)] string *colLabel;
} ColInfo;

long int GET_COLWIDTH(
    [in] handle_t h,
    [in,out] ColInfo *result
);

} /* end of Definition */

Figure 58. Conformat Array/IDL - Example A - Type 2
7.2.2 Unions

In C, the purpose of a union is to be able to define a number of data types that are based on the same storage location. The program is responsible for keeping track of the type of data currently being used or stored. IDL also provides for unions for the same purpose, however the means of determining the current data type must be specified because the data must be marshalled and unmarshalled. This is called a discriminated union. The example on page Figure 61 on page 138 is a simple IDL union definition. The basic idea is to generate a structure that contains both discriminator variables (discriminator_name) and the actual union. This will ensure that the client and server will agree on the data types. This technique is similar to Cobol’s redefine, PL/I’s based or define and assembler’s DSECT mappings. Also Reference OpenEdition Distributed Computing Environment Base Services MVS: Application Development Guide (SC09-1484).
typedef
union switch (unsigned short discriminator_name) A_union_name
{
    case 1: char name_1[20];
    case 2: short int number_1;
    case 3: float number_2;
} un_ty;

Figure 61. Union Structure - IDL Definition
typedef struct {
    idl_ushort_int discrimator_name;
    union {
        /* case(s): 1 */
        idl_char name_1[20];
        /* case(s): 2 */
        idl_short_int number_1;
        /* case(s): 3 */
        idl_short_float number_2;
    } A_union_name;
} un_ty;

Figure 62. Union structure - IDL Compiled

Figure 60 on page 138 show a conceptual storage mapping of the IDL union structure defined on page Figure 61 on page 138. The variables name_1, number_1 and number_2 can all map onto the same storage location. The discrimator_name will identify which variable type should be used to access the data. A union structure is used to pass different data types between clients and servers.

Discriminated unions and conformant arrays can be combined to form a very powerful means of passing varying amount of data within a DCE client/server environment.

Reference the DB2 and SQL code extracts in the following chapter for examples of conformant arrays and unions.

7.3 Dynamic Storage allocation

When you use dynamic allocation you must be aware of how storage will be used and the effects of remoteness. If the storage and it’s data will be used within a specific function then the normal dynamic allocation routines and functions such as malloc() and free() can be used. On the other hand if the data needs to be passed within the DCE environment, then you may have to use RPC memory management routines.

Manager code should use rpc_ss_allocate() routine to allocate storage for storage used to pass data back to the client. The manager can not free the storage itself because then the data would not be available to be marshalled to send back to the client. On the other hand, if the storage is not freed it will not be available to the system. Storage that is allocated by rpc_ss_allocate() is released by using rpc_ss_free() or after any output parameters have been marshalled by the stubs to the client.

At times it will be necessary to enable and disable RPC memory allocation. Under normal client/server usage RPC memory allocation is enabled by the server stub. If the application is a client as well as a server it becomes necessary to enable RPC memory allocation. The routine that must be invoked to initialize a memory allocation environment is rpc_ss_enable_allocate(). The environment once set up can be released by rpc_ss_disable_allocate(). This routine will release all memory allocated by the calls to rpc_ss_allocate() within the scope of the enable. Reference OpenEdition Distributed Computing
Environment Base Services MVS: Application Development Guide (SC09-1484) for more information on storage management routines.
Chapter 8. DL/I and DB2 Server Considerations

DCE itself does not provide services to perform transaction management or logical units of work. However, the DCE server application can use DB2 resources by using SQL statements to address relational databases directly from server programs. In this environment call attach facility (CAF) provides the unit of work function. It is also possible to create an interface similar to the CAF using the database control (DBCTL) interface provided by IMS/ESA. In this scenario the DBCTL interface provides the unit of work function for DL/I databases.

![Diagram of IMS and DB2 Overview](image)

**Figure 63. IMS and DB2 Overview**

If you use DB2 and DL/I from the same DCE thread, you must write data to only one system in any one unit of work. If you write to both systems within the same unit, a system failure may leave the two databases inconsistent with no possibility of automatic recovery. This will not corrupt the data stored in the database but may leave it in an inconsistent state. To end a unit of work in DB2, executes the SQL COMMIT statement; to end a unit of work in DL/I, issue the SYNCPOINT command. You can’t simultaneously end a unit of work in DB2 and DL/I.
8.1.1 Ensuring the Integrity of Data Base Updates

Ensuring the integrity of a RPC request is complex and highly application dependent. A RPC request may fail at any point for several different reasons, examples include network, machine and application failures. A failure may occur after DB2 or DL/I resources have been committed in the client or manager.

Although many failures can occur, the basic problems can be reduced to the following areas:

- The RPC client and server program must be able to determine when a failure is sufficiently severe that a request must be aborted.

- The RPC server program must be able to tolerate updates that are requested by the client more than once.

Ultimately, the application system must be able to react appropriately to correct each of the just-mentioned failures. When the failure is not entirely clear, which is often the case, the application must be able to determine the extent of the failure, and be tolerant of retransmitted (duplicate) RPC request.

A RPC request may actually run to completion even though the communication session failed before the reply message could be sent to the client program or the client may fail before local resources are updated. In this scenario the server database update(s) will have completed but the client will not be aware of this state.

The RPC client and server program must be able to recognize and recover from any of these failures in the server or client process:

- The RPC request failed to reach the server. The client program must reissue RPC request.

- The RPC request reached the server but the response did not arrive at the client. Here, the server data base update was successful.

This case can be particularly confusing, since the update was successful but the client has no response indicating success.

The RPC request failing does not necessarily mean that the database update failed.

The client will reissue the RPC request and the server program must be designed to inform the client that the update has already occurred successfully.

- The RPC request reached the Server and the response arrived at the client, but the client failed while attempting to commit the local database resources.

This requires that the DCE client design be stateless, this places a substantial burden on the application programmers.

There are many techniques to eliminate RPC and database failure exposures. Although the details of the implementation are application dependent, the following techniques are possible candidates:

- The simplest and most reliable solution is to code your server application to recognize duplicate RPC requests (for example, time stamps in the input
messages, or unique keys in the database). If you do this, the logic on the client is simpler. When the outcome of the transaction is in question, the client can always send the transaction again. The server application is then responsible for deciding how to process any duplicate requests that it receives. For example, the server application might return an error message to the client when a duplicate time stamp or key is received.

- If your application makes an identifiable change to a database, the client can query to determine the success or failure of the transaction. For example, if a failure occurred when storing a new invoice on a database, the transaction program might issue a RPC request to determine if the failing invoice number was stored in the database successfully. The client program could then use the result of the query to determine if the failing invoice must be re-transmitted.

### 8.2 DB2 Server Overview

We elected to build an application that will exploit the CAF interface for DB2. We developed a set of modules that access DB2 and DL/I databases from C. The C server is always executed as a heavy thread. A heavy thread runs under its own TCB, this allows a server to exploit CAF and the DBCTL interface. The CAF and DBCTL interface requires that all database access calls occur from the same TCB or thread of execution.

The RPC server and database access entry points for DB2 are identified below.

<table>
<thead>
<tr>
<th>Table 4. DB2 Client/Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
</tr>
<tr>
<td>DB2SQLCL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5. DL/I Client/Server Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
</tr>
<tr>
<td>IMSDLICL</td>
</tr>
</tbody>
</table>

You will find the code for these modules listed in the appendix.

#### 8.2.1 Call Attach Facility (CAF) Overview

The call attachment facility (CAF) is the recommended way for RPC applications to connect with DB2. CAF provides tight control over the session environment. Applications using CAF can explicitly control the state of their connections to DB2 by using connection functions supplied by CAF. A server may only connect to one DB2 subsystem but individual managers may execute different DB2 Plans. All DB2 service requests issued by programs running under a given task are associated with that task’s connection and operate independently of any DB2 activity under any other task.

Each connected thread can run a plan. Multiple threads in a single address space can specify the same plan, but each instance of a plan is run independently from the others. A thread can terminate its plan and run a different plan without fully breaking its connection to DB2.
Using multiple simultaneous connections can increase the possibility of deadlocks and DB2 resource contention. Your client and manager application design must consider this possibility.

### 8.2.2 Call Attach Facility Usage

The CAF requires that you use the following steps:

- Open the DB2 Plan.
- Execute the appropriate SQL statements.
- Close the DB2 Plan.

With Figure 64 we have shown the declarations required by the manager fragments shown in Figure 65 on page 145 and Figure 66 on page 145. The manager fragments perform the CAF open and the CAF close function. Part of CAF is a DB2 load module known as the call attach facility language interface. It is named dsnali. Its parameters are as follows. The DB2 subsystem is stored by the server in a global variable DB2SSID. The value is 4-bytes in length and padded on the right with blanks. In this example we always specify term_opt set to SYNC versus ABRT. The SYNC option results in a commit of any modified data while ABRT specifies roll back to the previous commit point. The return_code and the reason_code should be inspected and reported back to the user of the manager.

```c
#pragma linkage(dsnali, OS)

long int fnret;
char opnfunc[13] = "OPEN ";
char clsfunc[13] = "CLOSE ";
char plan[9] = "DB2SQL ";
char term_opt[5] = "SYNC";
long int return_code;
long int reason_code;
extern char DB2SSID[];
```

*Figure 64. Declaration for DB2 Call Attach Facility*

OPEN (See Figure 65 on page 145) allocates resources to run the specified plan. Optionally, a DB2 connection for the issuing task will be requested. If the requesting server does not already have a connection to the named DB2 subsystem, then OPEN establishes it. In the code fragment shown in Figure 65 on page 145 we open the DB2 Plan and possibly establish the servers connection with DB2.
Open the plans connections with DB2.

```
fnret = dsnali(opnfunc, /* OPEN */
               DB2SSID, /* subsystem */
               plan, /* plan name */
               &return_code, /* return code */
               &reason_code); /* reason code */

printf("Open request fnret = %d\n", fnret);
fflush(NULL);
if (fnret != 0)
{
    sprintf(result_msg,
            "*** UNABLE TO OPEN DB2 FOR RETURN: ",
            "%ld REASON: %ld",
            return_code,
            reason_code);
    *rpc_fault_status = 1;
    return;
}
```

Figure 65. Opening a DB2 Plan

CLOSE (See Figure 66) deallocates the plan and disconnects the server if this is the last manager currently accessing DB2. If the OPEN fails, do not use CLOSE, your manager will fail, and this will result in abnormal termination of your server. In the code fragment shown in Figure 66 we CLOSE the DB2 plan and possibly disconnect the server from DB2.

```
fnret = dsnali(clsfunc, /* CLOSE */
               term_opt, /* term option */
               &return_code, /* return code */
               &reason_code); /* reason code */

if (fnret != 0)
{
    sprintf(result_msg,
            "*** UNABLE TO CLOSE DB2 FOR RETURN: ",
            "%ld REASON: %ld",
            return_code,
            reason_code);
    *rpc_fault_status = 1;
    return;
}
```

Figure 66. Closing a DB2 Plan

A tracing facility provides diagnostic messages that aid in debugging programs and diagnosing errors in the CAF code. In particular, attempts to use CAF incorrectly cause error messages in the trace stream. To produce CAF error message you must allocate a DSNTRACE data set. You may allocate the
DSNTRACE data set dynamically or by including a //DSNTRACE DD statement in your server JCL as shown by Figure 67 on page 146. CAF writes diagnostic trace messages to that data set.

```
//**********************************************************
//* *
//* JCL TO RUN THE DB2SQL SERVER *
//* *
//**********************************************************

//RUN EXEC PGM=DB2SQLS,PARM='POSIX(ON)/DB3B'
//STEPLIB DD DSN=VANAERS.DCE.LOAD,DISP=SHR
// DD DSN=DSN310.SDSNLOAD,DISP=SHR
// DD DSN=CEE.V1R3M0.SCEERUN,DISP=SHR
// DD DSN=DCE.SEUVEXEC,DISP=SHR
//SYSIN DD *
//SYSERR DD SYSOUT=*  
//SYSPRINT DD SYSOUT=*  
//DSNTRACE DD SYSOUT=*  
```  

Figure 67. Execution JCL required for DB2SQL Server

Preparing your application program to run CAF is similar to preparing it to run in other environments such as CICS, IMS, and TSO. You can prepare a CAF application either in the batch environment or by using the DB2 program preparation process. It should be noted that the DB2 precompiler generates fixed block with a logical record length of 80 bytes. If you attempt to send the output to a VB file it may damage the dataset.

For more information on CAF and preparing and executing your program see IBM DATABASE 2 Version 3 DB2 Application Programming and SQL Guide, SC26-4889.

### 8.2.3 DB2 Manager Code

The CAF interface, as used in our implementation, requires that the manager code open and close the plan. The connect and disconnect is implicitly done by the open and close. This simplifies the design of the manager and server. It may be necessary in a robust design to explicitly connect and disconnect from the DB2 subsystem. An extract of the code is shown in Figure 68 on page 147.
```c
#pragma linkage(dsnali, OS)

void gcc( idl_char *c_last,
          idl_long_int *sql_code,
          idl_char *result_msg,
          CUSLST *ppCusLst);

void get_customersC( handle_t IDL_handle,
                      idl_char *c_last,
                      idl_long_int *sql_code,
                      idl_char *result_msg,
                      CUSLST **ppCusLst,
                      error_status_t *rpc_comm_status,
                      error_status_t *rpc_fault_status)
{
    long int fnret;
    char opnfunc[13] = "OPEN "; /* OPEN */
    char clsfunc[13] = "CLOSE "; /* CLOSE */
    char plan[9] = "DB2SQL "; /* plan name */
    char term_opt[5] = "SYNC"; /* term option*/
    long int return_code;
    long int reason_code;
    extern char DB2SSID[];

    fnret = dsnali( opnfunc, /* OPEN */
                    DB2SSID, /* subsystem */
                    plan, /* plan name */
                    &return_code, /* return code */
                    &reason_code ); /* reason code */
    printf("Open request fnret = %d\n", fnret);
    fflush(NULL);
}
```

Figure 68 (Part 1 of 2). DB2 Manager Code Extract
8.2.4 DB2 Functional Code

The functional code for the DB2 server is very simple with the exception of the conformant array (see Figure 70 on page 150 for an example). To communicate with DB2, you need only do the standard things:

- Declare a communication area (SQLCA)
- Declare the tables you use
- Declare the data items used to pass data between DB2 and C
- Code SQL statements to access DB2 data
- Handle exceptional conditions that are indicated with return codes from DB2 in the SQLCA (see DB2 Application Programming and SQL Guide SC26-4889).

The module, gcc, returns the last significant SQLCODE detected (sql_code). A string is returned with a result message (result_msg). The message is used to provide a user friendly status message to the client. A pointer to a pointer (ppCusLst) that points to the CUSLST structure. It is passed to the gcc to allow the functional code to dynamically allocate a CUSLST structure. This allows a pointer to the conformant array CUSLST to be returned to the client program. The conformant array contains a size and a vector of pointers to CUSTOMER records. The IDL definitions required to build the CUSLST conformant area is shown in Figure 69 on page 149 (see 7.2.1, “Conformant Arrays” on page 129 for more on conformant arrays).

The pointer ppCusLst is assigned the results of an rpc_ss_allocate that request enough storage to return 20 customer pointers and the size element for the
vector. Remember that a robust design should expand this area if more than 20 customers match, the design should also inspect the results of the rpc_ss_allocate request for possible failure and return an appropriate response to the client. The cursor C1 is opened to select all LAST NAMES that are like the input parameter c_last. A while loop is entered until the SQLCODE is not equal to zero (0). Each row that is fetched is placed in the next available element within the CUSLST structure and the size of CUSLST is incremented. Finally the C1 cursor is closed and the results returned to the manager. The extract of the functional code is shown in Figure 70 on page 150.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct
{
    char C_FIRST[17];

    ............
    char C_PHONE[17];
} CUSTOMER;

typedef struct
{
    long int size;
    [size_is(size)] CUSTOMER *pCustomers[*];
} CUSLST;

void get_customersC
(
    [in,string] char c_last[17],
    [out] long int *sql_code,
    [out,string] char result_msg[101],
    [out] CUSLST **ppCusLst
);
```

*Figure 69. Interface Definition Language for CUSLST*


```c
#include <db2sql.h>

void gcc( idl_char *c_last,
    idl_long_int *sql_code,
    idl_char *result_msg,
    CUSLST **ppCusLst)
{

EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE CUSTOMER;
EXEC SQL BEGIN DECLARE SECTION;
    char FIRST[17];
EXEC SQL END DECLARE SECTION;
*ppCusLst = rpc_ss_allocate(sizeof(CUSTOMER) +
   (99 * sizeof(CUSLST *)));

char PHONEY17":
EXEC SQL END DECLARE SECTION;

memset(last, '\0', sizeof(last));
memset(last, '.', sizeof(last)-1);
strcpy(last, c_last);
```

Figure 70 (Part 1 of 2). DB2 Functional Code
/*******************************************************************/ /* */ /* Declare the Cursor... */ /* */ /*******************************************************************/ EXEC SQL DECLARE C1 CURSOR FOR SELECT C_FIRST, ........ C_PHONE FROM DAHLE.CUSTOMER WHERE C_LAST LIKE :last; EXEC SQL OPEN C1; ........ while (SQLCODE == 0) { EXEC SQL FETCH C1 INTO :FIRST, ........ :PHONE; if (SQLCODE == 0) { (*ppCusLst)->pCustomers[(*ppCusLst)->size] = rpc_ss_allocate(sizeof(CUSTOMER)); strcpy( (*ppCusLst)->pCustomer[(*ppCusLst)->size]->C_FIRST, FIRST); ........ strcpy( (*ppCusLst)->pCustomers[(*ppCusLst)->size]->C_PHONE, PHONE); (*ppCusLst)->size = (*ppCusList)->size + 1; } } ........ EXEC SQL CLOSE C1; ........}
8.3 DL/I Access from the DCE Server to DL/I Databases

People often mention DL/I databases by using the word IMS. Using IMS in this context is misleading. The IMS product is composed of several components, which can be selected or not in a particular environment, for example, IMS transaction manager, the DL/I database, the fast/path (FP) database, or DB2 database. DL/I databases, including fast path databases, in IMS/V3x and above, are controlled by a separate IMS/VS component called DBCTL (Database Control). This component can be considered as a counterpart of the DB2 database manager. DBCTL can exist on its own, or can be used in a closed combination with the IMS transaction monitor Data Communication Control (DCCTL). The database and transaction logging is done in common.

Traditionally, access to DL/I and FP databases from user code, is by using dependant regions such as message processing regions (MPR), batch message processing regions (BMP), or wait for input regions (WFIR). The code can be written in almost all existing high level languages. This code, running in the dependant region, is “single threaded” and as such can not be used in a DCE server, which by definition is a “multi-threaded” environment.

Although the current DCE environment can not be called a complete transaction monitor, it nevertheless offers a set of functions which are part of a transaction monitor. Transactions (remote procedures) can be dispatched in parallel on several threads, but DCE lacks the concept of “unit of work.”

Each DCE manager thread in the server application must be able to open an independent DL/I or FP access thread towards a given (common for the server) DBCTL, which controls a set of DL/I and FP databases. Direct access to the DBCTL is possible from outside of IMS, even when the DBCTL is used in combination with the DCCTL. This DBCTL Interface is published in the IMS/ESA Customization Guide: Database SC26-3064 and has been used successfully in the implementation of the CICS to DL/I interface. One solution for the DCE server access to DL/I and FP databases consists of implementing this DBCTL interface within the DCE server, and making it available from the server manager code by designing an API. The design of this API is a challenge because it has to be efficient and should allow the user to write DL/I calls as documented in the normal IMS application manuals. Such an implementation is called coordinator controller (CCTL).

DCE/MVS offers another possibility to access the DL/I and FP by using the AS (application support server for IMS or CICS). This solution is a worthwhile consideration when the transaction program already exists. The DCE application support server acts as an intermediate generic server (gateway) between the DCE client and the IMS or CICS transaction monitor. Fig Figure 71 on page 153 shows the different solutions for DL/I and FP access.
The use of the application support server approach has the advantage that the resource updates to DL/I and FP and others (DB2, files messages) are coordinated by the resource manager, which is part of the transaction manager (TM). The TM coordinates the unit of work by using two-phase-commit techniques, but the DCE connection is not involved in this process. Another advantage is that most of the existing application code can be reused. Keep in mind that probably some changes will have to be made in the message processing code to accommodate the message layout as exchanged with the AS by the ISC (inter systems coupling). The application support servers are part of a standard offering by IBM. Full documentation can be found in the following Manuals:

- **MVS/ESA OpenEdition DCE: Application Support Server for IMS and CICS**, GG24-4482

Direct CCTL approach has the advantage of accessing the database managers directly (DL/I, FP, DB2), but resource coordination is not yet available. Each subsystem access is sync pointed separately. This happens in DB2 by a synchronize (one-phase-commit), but in DBCTL, a two-phase-commit interface is available. Unfortunately the absence of a resource coordinator in DCE means this two-phase-commit can not be exploited easily. In this chapter we will concentrate on the implementation.
of a CCTL within the DCE server. We will not highlight the details of the implementation, which are contained in a few assembler routines, which have to be included at link edit time, but we will focus on the use of this implementation (API) by the server and the manager code (The assembler routines are on the diskette that accompanies this book). The details of a DBCTL/CCTL implementation are completely described and outlined in Chapter 11, Database Control(DBCTL) interface, in SC26-3064 IMS/ESA Customization Guide: Database).

### 8.3.1 Implementation of a CCTL in a DCE Server

Basically the DCE server has to be enlarged with a piece of code, called CCTL (coordinator controller), which hosts a database resource adapter (DRA). The DRA is the interface between DBCTL and CCTL. The extension as will be explained, has been implemented by two modules.

- **DCECCTL**, executed once by the server code and provides some initialization and termination functions for the application.
- **DFCDCEI** with several entry points, is called by the manager code when needed. It provides functions used by each manager routine.

Figure 72 shows the structure.

![Figure 72. Including CCTL and DRA Within the Server.](image)

The key functions for this CCTL are delivered by a start-up table which describes the DRA characteristics and an entry into a start-up/router routine (DFSPRRC0) shipped with the IMS product. These elements are used in our implementation but the only exported visible API's to be used by the server code are

- **CTLINIT** to initiate the DRA
- **CTLTERM** to terminate the CCTL function.

The DRA start-up table is prepared by the assembly of the DFSPRP macro (See Figure 73 on page 155). The result of assembly and LKEDIT is a DFSPZPxx loadmodule. Several DRA start-up tables can be prepared specifying different parameters, for example, IMSID and IMSUSER. It is our advice to have a
separate DFSPZPxx for each IMSID/IMSUSER combination. As we will see later
the DFSPZP suffix is one of the parameters that will be passed to the server code.
Find below one example of the DFSPRP macro (See Figure 73). for connection to
IMSID=IM41 with USERID=DCEIPOK. As a result of the DRA Table preparation
a load module (DFSPZPxx) is created which will be used during the initialize of the
DRA adapter.

DFSPZP41 CSECT
  DFSPRP DSECT=NO,DBCTLID=IM41,DSNAME=IMS41.RESLIB,
  USERID=DCEIDPOK,
  MAXTHRD=5,MINTHRD=5
END

Figure 73. DFSPZP

The DCE server is started using the following EXEC JCL statement (See
Figure 74).

Figure 74. IMS Server JCL

The parameter field contains three items
• POSIX(ON) is a required parameter to let the server run in the open POSIX
  environment.
    / ends the run time options and is mandatory.
• DB3B is the first user parameter and indicates the SSID of the DB2 CAF
  interface as explained in an earlier section.
• 41 is the second user parameter and will be passed to the CCTL
  initialization, and appended to DFSPZP to get the name of the DRA start-up
  table.

The initialization of the DRA will use the start-up table. The details are
explained in the *IMS/ESA Customization Guide database SC26-3064. In our
implementation the initialization is done in load module DCEECCCTL, that is link
edited with the server code. The INIT phase is requested by calling the API
CTLINIT from the server code after picking up via the server parameters (second
parameter) the DFSPZP suffix. It is quite logical to do this in the server code as
we execute this code only once.
long CTLINIT (char *, long, char *);
#define OK 1
short ims=0;
long numthrds;
char imsuser[9]; /* if imsuser to be overridden */
char IMSSUF[2]; /* contains DFSPZP suffix */

/*------------------------------*/
/*----Connect to DBCTL---------*/
/*------------------------------*/
numthrds = 5;
strcpy(imsuser,SERVER);
/*------------------------------------------*/
/* Initialize CCTL and DRA structure */
/*------------------------------------------*/
rc = CTLINIT(IMSSUF,numthrds,imsuser);
printf(“after CTLINIT rc %d \n”, rc);
fflush(stdout);
/*------------------------------------------*/
if ( rc > 0 ) {
    printf(“Server ended no connection with IMS \n” , rc);
    fflush(stdout);
    return(rc);
}
ims = OK;

Figure 75. Initializing CCTL and DRA

The extract above (See Figure 75) shows the code as used in our examples. The return code from CTLINIT together with some console messages shows the (un)successful completion of the “INIT” and the reason. A rc (return code) of 0 allows us to continue with further DCE registering, knowing that we prepared a DRA infrastructure as depicted in Figure 76 on page 157. This structure will be at the disposal of DCE managers, allowing them to have a separate “DL/I path” to the resources controlled by DBCTL with IMSID =imsid.
The termination of the CCTL-DBCTL connection has to be done after terminating the server listen state. The following extract (See Figure 77) shows how it has been integrated in the server code termination.

```c
if (ims == OK) {
    rc = CTLTERM();
    printf(" after CTLTERM rc %d \n", rc);
    fflush(stdout);
}
```

**Figure 77. Terminating the CCTL-DBCTL Connection**

### 8.3.2 Use of the CCTL Interface in the DCE Managers

All API's required from the DCE manager are provided by a load module, DFSDCEI, which has to be used in conjunction with DCECCTL. Both programs have to be LKEDIT with server and manager code. DCEILC has also to be included. Examples of API's are:

- `DFSSCHED` (psbname) to schedule a PSB (Program Specification Block)
- `DFSTRMCM` to commit and terminate a PSB scheduling
- `DFSTRMAB` to abort and terminate a PSB scheduling
CTPLI to access DL/I and FP and to get the PCB (Program Communication Block) addresses.

The DCE manager is running as a dispatchable unit on a heavy thread TCB. Each manager should be able to schedule a PSB as defined within the IMS environment and execute DL/I calls against the DL/I and FP databases referenced within this PSB. We suggest splitting up the manager code between a small manager code stub which fulfills the link with the DCE server stub (IDL signature) and the real functional code where the DL/I and DB2 accesses are done (See Figure 78).

8.3.2.1 Manager Code stub
The CCTL interface, as used in our implementation, requires a schedule of the PSB. The PSB has to be predefined as a Batch PSB within the IMS system. Also the PSB has to be adapted to the high level language. The C language requires a C/COBOL/ASM type PSB. The scheduling of the PSB and the termination of the PSB belongs in the small introductory manager code. An extract of the required code is shown (See Figure 79)
#include <imsDL/I.h>

#pragma linkage(DFSSCHED,OS)/*required */
#pragma linkage(DFSTRMCM,OS)
#pragma linkage(DFSTRMAB,OS)

void GPC( char *part_number,
         idl_char *result_msg,
         PPARTSTOK *ppPartStok); /* prototype Funcode */

/*--------DCE Manager Entry----------------------*/
void get_partC( handle_t IDL_handle,
                idl_char *part_number,
                idl_char *result_msg,
                PPARTSTOK *ppPartStok,
                error_status_t *rpc_comm_status,
                error_status_t *rpc_fault_status)
{
    int rc;

    printf("Entry to get_partC. . .\n");
    fflush(NULL);
    rc = DFSSCHED("DFSSAM08"); /* PSB Scheduling */
    printf("DFSSCHED rc = %d\n", rc);
    fflush(NULL);

    /*--------------------->
    GPC( part_number, /* <==== call to Function Code */
         result_msg,
         ppPartStok);
    rc = DFSTRMCM(); /* PSB Termination */
    printf("DFSTRMCM rc = %d\n", rc);
    fflush(NULL);
    printf("Exit from get_partC. . .\n");
}

..........................

Figure 79. Scheduling the PSB

- DFSSCHED is an entry point in the module DFSDCEI with code to handle the PSB scheduling as required by the CCTL interface. The DFSSCHED call requires one parameter, the PSB name. Remember that the language type of this PSB has to be in agreement with the language of the functional code which is going to use it.

- DFSTRMCM is another entry point in the module DFSDCEI with code to terminate (commit) the PSB scheduling. The DFSTRMCM call includes several basic CCTL calls.
  - A Prepare, first step of the two-phase-commit
  - A Commit, second step of two-phase-commit. The two steps are executed after each other as the two-phase approach offered by CCTL can not be exploited due to a lack of a resource manager. DFSTRMAB has also been provided as an API, with "abort" as the second step.
The combined first two calls execute what could be called in IMS terms a syncpoint. IMS resources like locks have been freed. In our implementation the two calls are always followed by a additional terminate which deschedules the PSB.

8.3.2.2 Functional Code

Traditionally, when a IMS program gets control from the IMS start-up code, parameters are passed, which hold handles towards the Program Communication Blocks (PCB). The PCB points directly to the DL/I and FP databases and their access (by index or not). The C interface looks like:

```c
rc = ctdli (function, db pcb, i/o area, ssai);
```

This approach does not work in the DCE environment because the manager code signature has to match exactly the prototype declaration from the IDL definition. To solve this problem we introduced an additional parameter, FUNC, within the DL/I function calls. This function, GPSB, is intended to get the handles to the PCB’s (that is to fill in PCB pointers as normally passed by the routine parameters).

```c
#include <ims.h>

............
void GPC(char *part_number,
         idl_char *result_msg,
         PPARTSTOK *ppPartStok)
{

  IO_PCB_TYPE *io_ptr = NULL;/*PCB declares */
  PCB_STRUCT_8_TYPE *di21part_ptr = NULL;
  ............

  char *GPSB = "GPSB";

  ............
  int rc;
  int three = 3;
  rc = ctdli(three,GPSB,&io_ptr,&di21part_ptr); /* <==== */

  printf("ctdli GPSB rc = %d \n", rc);
  fflush(NULL);
  if (rc != 0)
  {
    return;
  }
  ............
}
```

*Figure 80. Handling the PCB*

After this GPSB call has been successfully completed, all other DL/I calls should be business as usual.
8.4 Split-up of the DCE Server

In a previous chapter we suggested having the manager code split-up in two parts:

1. Manager code stub
   This code gets control from DCE, opens and closes DB2 connections, schedules and deschedules PSB’s, and calls the functional code.

2. Functional code
   This routine does the real DB2 and DL/I access and other application related processing. It is obvious that in a DCE client/server environment DCE services like rpc_ss_allocate are also required in this code.

The manager stub could be very small, but the functional code could be very extensive, and also subjected to frequent changes. We also have to consider that a server is not limited to one manager, but could contain several managers. This would lead to many stubs and many functional codes. If we build the server as one load module, each change in one piece of code would require the link edit of the DCE server load module. To avoid this setup, a more modular approach is sought. This approach consists of changing the manager code stub’s call of the functional routine into a load-call of this routine. Although this change seems to be very simple and straightforward, it requires other issues to be solved. The “fetched” routine has some requirements.

- The fetched routine needs access to DL/I and DB2 and to some DCE services. In practice this translates into access to some external entry points.
  - ctdli for DL/I access
  - DSNHLI for DB2 access (result of DB2 preprocessing)
  - rcp_ss_allocate, rcp_ss_free

When everything is link edited together, these references are resolved by the DCE library routines and our DL/I stubs (DFSDCEI, DCECCTL). Here we have the same requirement but the solution will be different.

- The fetched load module has to be thread safe as it could be used by several threads in parallel.

- To optimize the fetch we also introduce some caching mechanism, which are based on tables, reserved by the server code. This allows to optimize the load and call of the functional load modules.

The fetch of the subroutine is not done directly by the manager code stub but indirectly by a “glue” code which is part of DFSDCEI. This intermediate function is called using the entry point GOFETCH as shown in the following extract (See Figure 81).
rc = DFSSCHED("DFSSAM08");
printf("DFSSCHED rc = %d\n", rc);
fflush(NULL);

/*
printf("Calling GPC. . .\n");
GPC( part_number,
    result_msg,
    ppPartStok);
*/
/*=================================================
printf("Fetching GPC. . .\n"); /* GPC is name of fetched loadmodule */
rc = GOFETCH("GPC",
    part_number,
    result_msg,
    ppPartStok);
printf("After fetch of GPC. . .\n");
/*=================================================

rc = DFSTRMCM();
printf("DFSTRMCM rc = %d\n", rc);

---

Figure 81. Fetch Routine

The usual parameters are preceded by one additional parameter which contains the name of the functional load module.

To fulfill the requirements of the external entry points used by the functional code, the module is link edited with a shell routine (DCEREXT), which contains entries like ctdli, DSNHLI, RPC@SS@A (which is the short form of rpc_ss_allocate). This is shown in Figure 82 on page 163.
Below is an example of the link edited control statements. Note that the load module is prepared as NON-MAIN by explicitly specifying the ENTRY.

```
INCLUDE DL/ILIB(DCEREXT)
MODE AMODE(31)
MODE RMODE(24)
ENTRY GPC
NAME GPC(R)
```

The entry points in DCEREXT are only "place holders," and have to be redirected at execution time to the real entry points which are located in the base Server.

This redirection is done by the logic in the shell routine (DCEREXT) with assistance of another non-executable load module, loaded and filled by DCECCTL. This standalone loadmodule (DCECTEXT) has a RENT attribute, so that only one copy is loaded, and each subsequent load will get a handle to it. This load module contains all the real entry point addresses and DCEREXT will map the dummy entry points to the corresponding real after getting a handle to DCECTEXT by LOAD.
The DCE server, besides the Server code and manager stub code has to be link edited with the following modules.

- **DCECCTL** - Initialize and terminate IMS connection from server code.
- **DFSDCEI** - Manager API’s for DL/I access.
- **DCEILC** - API’s for DCE run-time services like rpc_ss_allocate.

Load module **DCECTEXT**, which will be loaded and prepared by the server code, has to be available as an independent load module in a non-authorized library with attribute RENT.
8.5 Summary on DL/I and DB2 Access from Manager Code

It is possible to combine DB2 and DL/I access in one manager. DL/I uses the CCTL/DBCTL approach; DB2 uses CAF. In one server it is possible to connect to one (only one) DB2 and IMSID. That is, all managers work with the same DB2/IMS as selected directly by DB2SSID in the parameter field of the EXEC statement, and indirectly by the suffix of the DFSPZP module, also specified by the second parameter field. The complete TCB layout is shown in Figure 84. The TCB structure built-up by the DRA is composed of sub TCBs of the main server TCB, while the manager code threads are real heavy POSIX threads. DB2 access by CAF is executed directly from each of those threads. The DL/I access is executed through the DRA work threads. Each manager, requiring access to DL/I, gets an associated work TCB. This association is done when the PSB is scheduled and cleared at termination.

DB2 and DL/I access require a particular preparation of the DCE server. The server has to be link edited with a few additional modules. This modules can be included on the LKED step (PLKED is not necessary).

To utilize the DBCTL interface the following items must be specified when you configure your IMS system.

- DBRC shared control must be specified.
- The DL/I SAS address space is required.
The composition of a DCE server with DB2 and DL/I access is shown in Figure 85.

- DCECCTL contains the CTLINIT and CTLTERM API's. CTLINIT builds some tables with addresses and references, reserves the association tables between the manager Pthread TCB and the DL/I work TCB. It also has several references to entry points which are defined in DFSDCEI.

- DFSDCEI contains other API's, for example DFSCHED, DFSTRMCM, and CTPLI. This module has to be included together with DCECCTL.

- DSNALI provides all entry points required by DB2.

In the Figure 86 on page 167 we show in detail all steps which are processed by server code.
Server Code Initialization

* Get DB2SSID from PARM and PUT it in Global for Manager Code

* Get Suf for DFSPZP member describing CCTL to IMS interface

* CTLINIT (suffix, imuser..)
  - Initialize DRA with PAPL and verify Acceptance
  - Wait for Adapter Setup Completion via asynchronous Control Exit
  - Allocate USER and PAPL areas for Manager Threads
  - Allocate and Fill table with external EP References

* do DCE with/without CDS and Security (register)

* RPC_SERVER_LISTEN

  *

* CTLTERM
  - Close CCTL connection with IMS

* Do DCE (unregister)

* Exit

Several accompanying messages are displayed at the console showing the progress of the server. These messages are issued by DCECCTL, which builds up the DRA with IMSID 41 using USERID IMSDL/ISV.

```plaintext
+++>IDE105.IMS CONNECT SUF 41 WITH USERID IMSDL/ISV
+++>IDE112 DCECCTL INIT 1 OK
+++>IDE113 DCECCTL TEST INIT STATE
+++>IDE101 DCECCTL ENTERING CONTRL EXIT
+++>IDE102 DCECCTL RESYNC REQUEST
+++>IDE000 DCECCTL NO RESYNC
+++>IDE100 DCECCTL INTERFACE OK
+++>IDE174 DCECCTL NR ANCHRS IS 0005
+++>IDE178 DCECCTL ANCHORS OK
+++>IDE181 DCECCTL LMOD CACHING OK
+++>IDE201 DCECCTL LOADING DCEXTADR TAB
+++>IDE202 DCECCTL DCEXTADR TABLE OK
```

Figure 87. DCECCTL Messages

The server listing shows the following messages.
Starting SERVER IMSDL/ISV
Managers IMSDL/ISV can connect to DB2 DB3B
Server IMSDL/ISV will initialize IMS with member 41
  go to CTLINIT for IMSDL/ISV
  after CTLINIT rc 0
  ( IMSDL/ISV )Use protocol sequence OK -
  ( IMSDL/ISV )Register interface OK -
  ( IMSDL/ISV ) Bindings:
    ncadg_ip_udp:9.12.13.69 1095
  ( IMSDL/ISV )listening...

Figure 88. Server Messages

In Figure 89 we show the pseudo code for a manager that accesses DB2 and DL/I databases. Remember that this is a dangerous configuration for update and may result in difficulties with the recovery of in-doubt units of work. However, for inquiry it is quite safe. A manager that executes both DL/I and DB2 calls must use CAF and the DBCTL interface.

Manager Code BuildUp

- Get DB2SSID from Global in Server Code
- DSNALI (open PLAN)
- DFSSCHED (PsbName)
- CALL functional Program
  - LangGPSB to GET PCBaddresses
  - Execute LangTDLI via CCTL adapter
  - Execute SQL statements Via CAF
- DFSTRMXX
- DSNALI (close PLAN)
- Return

Figure 89. Manager Code
Dependant regions of an IMS control region all run in their own address space and as a consequence under their own TCB.

Figure 9.0. IMS Dependant Regions as DCE Clients

All types of dependent regions can run as a DCE client. Dependant regions are not called directly by MVS, but are loaded by an IMS program stub, which is also called “region controller.” As an example we show here the JCL, required to start-up a batch message program (BMP). The user program is called MATHXICP and works in connection with a program specification block (PSB). Although all dependant regions have a particular way to be started-up, it always occurs through a region controller.
The user program currently can be written in C, PL/I or COBOL, but has to be POSIX compliant and all DCE API calls have to be in C. Note that in the extracts shown in this chapter we do not show any DLI statements. The objective is only to indicate how an IMS dependant program (BMP, MPP, IFP, and so on) can be made to be a DCE client. However keep in mind that especially for message processing programs the transit time within the region will increase due to the additional RPC calls(s). The region controller starts as a normal, non-POSIX, program. The POSIX environment is turned on in the C-PL/I and C-COBOL program(s). This is shown explicitly in the extracts, where we split up between functional code (Main) and a C subroutine, responsible for DCE API and RPC call.

9.1 BMP Main Program in C

The following code (See Figure 92 form=textonly) shows the C Main. Note the 
#pragma statement which sets up the run time options. The 
#pragma statement has the options POSIX(ON) to establish a POSIX compliant DCE environment.

```c
#pragma runopts(env(IMS),plist(IMS),posix(on),stack(12000))
#pragma nomargins
#include <ims.h>

main() {
    #define ioPcb (IO_PCB_TYPE *)(_pcblist[0])
    #define dbPcb (_pcblist[1])

    /* call the RPC client program */
    rc = MATHXIC( c1, c2, &result);
    /* back from RPC client program */

    return(0);
}
```

Figure 92. BMP Main Program in C
9.1.1 C Subroutine for BMP C Main

Next is the skeleton code for the C subroutine called by the C BMP main program.

```c
int MATHXIC ( int c1, int c2, int *presult)
/***************************************************************************/
/* Compose */
/***************************************************************************/
 rpc_string_binding_compose(
 ........
/***************************************************************************/
/* Generate a bhandle from the string binding */
/***************************************************************************/
 printf("Generate a binding handle\n");
 fflush(NULL);
 rpc_binding_from_string_binding ( 
 ........
/***************************************************************************/
/* Call add with bh,values,number of vals,ptr to result.*/
/***************************************************************************/
 printf("Call add, pass handle and values, point to result...\n");
 fflush(NULL);
 add ( bh, va, ( long )numv, &sum ); /* RPC call to server */
 *presult = sum;
 ........
 return(0);
}
```

Figure 93. Skeleton for Subroutine in C

9.2 BMP Main Program in PL/I

The following code (Figure 94) shows the PL/I main. Note the PLIXOPT statement which sets up the run time options. The statement has the options POSIX(ON) to establish a POSIX compliant DCE environment. You can also put in a #pragma runopts(posix(on)) within the C subroutine being called from this PLI main program.
MATHXB: PROC(IO_PTR, DB1_PTR) OPTIONS(MAIN);

DCL PLIXOPT CHAR(100) VARYING
    INIT('POSIX(ON)') STATIC EXTERNAL;
DCL IO_PTR PTR,
    DB1_PTR PTR;
DCL 1 IO_PCB BASED(IO_PTR),
    ....
    ....
DCL 1 DB1_PCB BASED(DB1_PTR),
    ....
    ....

/* call the RPC client program */
RESULT_P = ADDR(RESULT);
CALL MATHXI(C1, C2, RESULT_P);
RC = PLIRETV();
/* back from RPC client program */
.
..
The \#pragma statements are required in order to ensure that the C code has the correct environment. The \#pragma statement linkage is necessary to set up a PL/I interfacing environment for this C code. Also note in passing a PL/I pointer to a C subroutine, that the C routine must receive it as a pointer to a pointer (double indirection).

It is also important to note that when PL/I main and the C subroutine are link edit together, a statement ENTRY CEESTART should be included as input to the linkage editor.
The following code shows the COBOL Main. POSIX compliant code is established by the C subroutine.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. MATHXIMS.
  AUTHOR. ........
  INSTALLATION POK.
  DATE-WRITTEN AUG 1994.

*===============================================================================
* THIS IS A MAIN COBOL PGM WHO CALLS A "C" DCE/RPC PROGRAM
*===============================================================================

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
  FILE-CONTROL.
  * ASSOCIATE FUNCTIONS WITH IO TYPES
  .
  .
  .
  .
  DATA DIVISION.
  .
  .
  .
  .
  FILE SECTION.
  .
  .

WORKING-STORAGE SECTION.
  01 RESULTP PIC S9(9) COMP.
  01 C1 PIC S9(9) COMP VALUE 20.
  01 C2 PIC S9(9) COMP VALUE 40.
  .
  .
  .

PROCEDURE DIVISION.
  DISPLAY "ENTERED MATHXIC1 - CLIENT STARTUP".
  PERFORM 100-INITIALIZE-PARAGRAPH.
  PERFORM 100-MAIN-PARAGRAPH.
  DISPLAY "PROGRAM MATHXIC1 - NORMAL END".
  .
  STOP RUN.
```

Figure 96 (Part 1 of 2). BMP Main Program in COBOL
100-INITIALIZE-PARAGRAPH.
    DISPLAY "IN-->100-INITIALIZE-PARAGRAPH".
    .
    .
100-MAIN-PARAGRAPH.
    DISPLAY "IN  100-MAIN-PARAGRAPH".
    .
    PERFORM 200-CALL-CLIENT.
    PERFORM 200-LIST-INFORMATION.
    .
    .
200-CALL-CLIENT.
    DISPLAY "IN-->200-CALL-CLIENT".
    DISPLAY "BEFORE-->MATHCICC".
    DISPLAY C1.
    DISPLAY C2.

    CALL "MATHXICB" USING BY CONTENT C1 C2 REFERENCE RESULTP.

    DISPLAY "AFTER <-MATHCICC".
    DISPLAY C1 C2.
    DISPLAY RESULTP.
    .
    .
200-LIST-INFORMATION.
    DISPLAY "IN-->200-LIST-INFO".
    .
    .

Figure 96 (Part 2 of 2). BMP Main Program in COBOL

9.3.1 C Subroutine for BMP COBOL Main

Next is the skeleton code for the C subroutine called by the COBOL BMP main program.
#pragma linkage(MATHXICB,cobol)
#include .........
#include .........
#include <dce/rpc.h>
#include "mathx.h"
.
.
int MATHXICB ( int c1, int c2, int *ppresult)
{
   ...
   ...
   rpc_binding_handle_t bh;
   unsigned_char_t *string_binding = NULL;

   printf ("In MATHXICB ...\n");
   fflush(NULL);

   rpc_string_binding_compose(
      .... ,
      .... ,
      .... ,
      .... ,
      .... ,
      ...
   )
   /* Check the status from there compose */
   .
   .

Figure 97 (Part 1 of 2). Skeleton Subroutine in C from COBOL
Figure 97 (Part 2 of 2). Skeleton Subroutine in C from COBOL

The two pragma statements are required in order to ensure that the C code has the correct environment. The pragma statement with the runopts ensures that we get a POSIX environment. The pragma statement linkage is necessary to set up a COBOL interfacing environment for this C code.
Chapter 10. DB2 Dynamic SQL Server

In today’s business environment customers require an interface that provides ad-hoc queries of relational data. This service can be provided with DCE/RPC and dynamic SQL. A design and sample program is presented that allows a user to query a relational table and return any valid DB2 data types. The DB2 Query is transmitted to the server as a string and the results are returned by a set of conformant arrays, one of the arrays contains a discriminated union.

What if a program must execute different types and structures of SQL statements? If there are so many types and structures that it cannot contain a model of each one, your program might require dynamic SQL.

We recommend that you use dynamic SQL only if you really require its flexibility. Using a few dynamic SQL statements in a program that runs for a long time will not make a significant difference, but using dynamic SQL for the same number of statements in a small transaction is much more noticeable.

10.1 Dynamic SQL Overview

A dynamic SQL statement is prepared during the execution of an SQL application, and the operational form of the statement is not persistent. The source form of the statement is a character string passed to DB2 by an application program using the static SQL statement PREPARE or EXECUTE IMMEDIATE.

DB2 support three types of dynamic SQL: dynamic SQL for non-SELECT statements, dynamic SQL for fixed-list SELECT statements and dynamic SQL for varying-list SELECT statements. The first type includes DELETE, INSERT and UPDATE statements, the second returns rows containing a known number of values of known type. The final type, the varying-list SELECT statement returns rows containing an unknown number of values of unknown type. When you use one, you do not know in advance exactly what kinds of host variables you need to declare in order to store the results. This example addresses the varying-list SELECT statement.

To execute a varying-list SELECT statement dynamically, your program must follow these steps:

1. Include an SQL Communication Area (SQLCA)

2. Load the input SQL statement into a data area

3. Prepare and execute the statement involves the following steps:

   a. Include an SQL descriptor area (SQLDA).
   b. Declare a cursor and prepare the variable statement.
   c. Obtain information about the data type of each column of the result table.
   d. Determine the main storage needed to hold a row of retrieved data.
   e. Put storage addresses in the SQLDA to tell where to put each item of retrieved data.
f. Open the cursor.
g. Fetch a row.
h. Eventually close the cursor and free main storage.

4. Handle any errors that might result.

The source for the dynamic SQL server is included in the sample diskette provided with this document. A detailed explanation of varying-list SELECT statements may be found in *DB2 Application Programming and SQL Guide*. The description of the client/server interface is described in the following section.
10.2 Dynamic SQL IDL

In Table 6 the parameters passed to the dynamic SQL server are self-explanatory with the exception of the result data structure. The prototype for SQL_SRV_EXEC, the dynamic SQL server, is shown in Figure 100 on page 182. The parameters passed to and from the server are further illustrated with Figure 98 on page 180 and Figure 99. The parameter result of type ValueList * is a complex data structure and will be explained in this section.

<table>
<thead>
<tr>
<th>Table 6. DB2 Dynamic Server Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
</tr>
<tr>
<td>h</td>
</tr>
<tr>
<td>dbplname</td>
</tr>
<tr>
<td>sql_stmt</td>
</tr>
<tr>
<td>result</td>
</tr>
<tr>
<td>pcond</td>
</tr>
<tr>
<td>output</td>
</tr>
<tr>
<td>operation</td>
</tr>
</tbody>
</table>
Figure 100. Parameters for the Dynamic SQL Server

The ValueList structure (Figure 101) is composed of three conformant arrays, colWidth, colLabel and colData each of length ncols. This illustrates the concept that multiple conformant arrays may use the same dimension variable as an argument to the size_is attribute, in this example the variable used is ncols. The element ncols also represents the number of columns that a dynamic SQL statement returned and nrowsProcessed represents the number of rows returned by the request.

typedef struct ValueList
{
    [ptr,ignore] ValueListExtHeader *header;
    unsigned short ncols;
    unsigned short nrows;
    unsigned short nrowsProcessed;
    unsigned short flags;

    [ptr, size_is(ncols)] unsigned short *colWidth;
    [ptr, size_is(ncols)] string *colLabel;
    [ptr, size_is(ncols)] ColDataType *colData;
} ValueList;

Figure 101. ValueList Structure

The colWidth pointer references a conformant array of typeidl_ushort_int that represents the length element of each column, this value is used to determine the amount of storage required for each data element retrieved from the table. The colLabel pointer references a conformant array of type string, each element points at a string containing the SQLNAME of the column. The colData pointer references a conformant array of type ColDataType, the array contains ncols elements. The ColDataType as shown in Figure 102 on page 183 is a discriminated union, a discriminator is included as part of the data structure, so that the correct data type is transmitted with the union by the DCE runtime environment. A union cannot contain a pipe, a conformant array, a varying array, or any structure that contains a conformant or varying array. A union cannot contain a ref pointer or any structure that contains a ref pointer. A union can contain a full pointer.
```c
typedef union ColDataType
switch (unsigned short colDataType) ColDataUnion
{
    case CDT_undefined:
        ;
    case CDT_integer:
        struct IntegerColType integerCol;
    case CDT_string:
        struct StringColType stringCol;

    ..........
    case CDT_packed:
        struct ByteColType packedCol;
    default:
        ;
} ColDataType;
```

Figure 102. ColData Discriminated Union

The elements of the ColDataType defined in Figure 103 on page 184 is valid because indicator and data have been defined as full pointers with length nrows. The indicator array is used to tag if a value is NULL and is derived from the SQLIND field. In the typical union members such as IntegerColType and StringColType the data pointer is used to reference an array of pointers, the pointers reference the corresponding element. StringColType contains an array of pointers that reference elements of type string and IntegerColType contains an array of pointers that reference elements of type long.

The remaining element ByteColType allows a slightly more efficient usage of storage for small strings or arrays of characters. If a row consists of very short character elements the application allocates an area capable of holding all of the elements from the table. This adds a slight complexity to the manager but reduces the number of rpc_ss_allocate calls required.
typedef struct IntegerColType
{
    unsigned short nrows;
    [ptr, size_is(nrows)] short *indicator;
    [ptr, size_is(nrows)] long *data;
} IntegerColType;

typedef struct StringColType
{
    unsigned short nrows;
    [ptr, size_is(nrows)] short *indicator;
    [ptr, size_is(nrows)] string *data;
} StringColType;

............

typedef struct ByteColType
{
    unsigned short nrows;
    [ptr, size_is(nrows)] short *indicator;
    unsigned short nbytes;
    [ptr, size_is(nbytes)] byte *data;
} ByteColType;

Figure 103. ColData Union Member Structures
Chapter 11. Configuring an Application Client and Server

A number of things have to be done to DCE servers in order to allow an application to use the directory and security facilities. In a production environment these steps will probably be performed by the system administrator. In a test/development environment they will probably be done by the application programmer.

11.1.1 Steps to Configure an Application Client and Server

- Log on as cell_admin and use regedit (the registry editor) to perform the following:
  1. Add your organization to the DCE registry.
  2. Add your group to the DCE registry.
  3. Add the principal to the DCE registry.
  4. Add the account to the DCE registry.
  5. Add the server to the group created in step 2.
  6. Add the server’s password to the keytab file.

- At this point it will be necessary to work with CDS control program (cdscp) to give your server authority to write to the CDS namespace.
  7. Create a directory for your objects.
  8. Create the object/s in the directory.

- Now use the ACL editor (acledit) to grant permissions to the server. This will make it possible for the server to Export it’s bindings to the CDS namespace.
  9. Modify the server to include read and write permissions at a minimum.

- Once you have done the above steps for your server you can customize the runtime environment for your DCE environment by setting specific DCE environment variables.

Note.

The examples in this section will use the following names:
- Organization name = IBM
- Group name = servers
- Principal name = mathxs
- Account name = mathxs
- Password = secret
- CDS Directory name = mathxdir
- CDS Object name = mathxs

You can either use these or use some of your own choosing.
11.2 Configuring an Application Server

This section describes how to use the registry editor (rgyedit) to create the DCE account for server mathxs, and add its password to the keytab file.

Configuration the Registry

To create a DCE account, the principal must have been created, as well as the security group and organization to which the account will be a member. The procedures for creating a DCE account described here assumes that the principal, group, and organization have not been created prior to the creation of the account. Skip the relevant steps if the principal, group, or organization have been created.

1. Add Organization

   • After starting rgyedit change to the org domain:
     
     rgy_edit=> domain org
     Domain changed to: org

   • Add the organization to which the user will belong. For example, if the organization is named IBM:
     
     rgy_edit=> add
     Add Org=> Enter name: IBM
     Enter UNIX number: (auto assign)
     Enter full name: ()

     You may opt to accept the defaults for the UNIX number and full name by pressing the Return key at these prompts. If you wish to enter values other than the defaults, see the security part of the OpenEdition Distributed Computing Environment Base Services MVS: Administration Guide (SC09-1485) for details on assigning values to these prompts.

2. Add Group

   • Change to the group domain:
     
     rgy_edit=> domain group
     Domain changed to: group

   • Add the group (or groups) to which the user will belong. For example, if the name of the group is servers:
     
     rgy_edit=> add
     Add Group=>
     Enter name: servers
     Enter UNIX number: (auto assign)
     Enter full name: ()
     Include group on PROJLIST [y/n]? (y)

     You may opt to accept the defaults for the UNIX number, full name, and the inclusion of the group into the project list by pressing the return key at these prompts. If you wish to enter values other than the defaults, see the Security part of the OpenEdition Distributed Computing Environment Base Services MVS: Administration Guide (SC09-1485) for details on assigning values to these prompts.

3. Add Principal Name

   It is at this point that you add your server principal name to the registry. Any time that you want to add another principal to the same group and organization you would begin at this step.
• Change to the principal domain:
  rgy_edit=> domain principal
  Domain changed to: principal

• Use the add subcommand to create the server principal. For example, if
  the server is named mathxs:
  rgy_edit=> add
  Add Principal=> Enter name: mathxs
  Enter UNIX number: (auto assign)
  Enter full name: ()
  Enter object creation quota: (unlimited)
  You may opt to accept the defaults (values that are enclosed in
  parenthesis after each prompt) for the UNIX number, full name, and
  object creation quota by pressing the return key at these prompts.

4. Add Account

At this point you have an organization (IBM), a group (servers) and a server
principal name (mathxs). This step will link them together under a unique
account, in this case mathxs.

• Change to the account domain:
  rgy_edit=> domain account
  Domain changed to: account

• Add the account for the user. For example, for the server whose
  principal name is mathxs:
  rgy_edit=> add
  Add Account=> Enter account id [pname]: mathxs
  Enter account group [gname]: servers
  Enter account organization [oname]: IBM
  Enter password:
  Retype password:
  Enter your password:
  Enter misc info: ()
  Enter home directory: (/)
  Enter shell: ()
  Password valid [y/n]? (y)
  Enter expiration date [yy/mm/dd or 'none']: (none)
  Allow account to be server principal [y/n]? (y)
  Allow account to be client principal [y/n]? (y)
  Account valid for login [y/n]? (n) y
  Allow account to obtain post-dated certificates [y/n]? (n)
  Allow account to obtain forwardable certificates [y/n]? (y)
  Allow certificates to this account to be issued via TGT authentication [y/n]? (y)
  Allow account to obtain renewable certificates [y/n]? (y)
  Allow account to obtain proxyable certificates [y/n]? (n)
  Allow account to obtain duplicate session keys [y/n]? (n)
  Good since date [yy/mm/dd]: (1993/06/01.11:34)
  • Enter n (for no) when prompted if you want to change authorization policy
    for this account:
    Create/change auth policy for this acct [y/n]? (n) y

5. Add the principal name to the group

Now it is time to add your server principal to two groups, the group servers
you added in step 2 and to another already existing group called
rpc-server-group.
• Change to the group domain:
  
  ```
  rgy_edit=> domain group
  Domain changed to: group
  ```

• Add the mathxs principal to the group servers using the member subcommand:
  
  ```
  rgy_edit=> member
  Enter group name: servers
  Enter name to add: mathxs
  Enter name to add: enter
  Enter name to remove: enter
  Enter group name: subsys/dce/rpc-server-group
  Enter name to add: mathxs
  Enter name to add: enter
  Enter name to remove: enter
  ```

In OpenEdition DCE Base Services MVS, access to the endpoint map is controlled through access control lists (ACLs). To facilitate the administration of authorized application servers that can access the endpoint map, a default security group, referred in this book as the RPC server Group, is created during OpenEdition DCE Base Services MVS configuration. The name of this group is subsys/dce/rpc-server-group. For an application server to have the appropriate permissions to the endpoint map, it must be made a member of this group. For more information on the RPC server group, refer to OpenEdition Distributed Computing Environment Base Services MVS: Administration Guide (SC09-1485) and the OpenEdition DCE Base Services MVS: Configuration and Getting Started (SC09-1490).

6. Adding Password to the keytab file

To possess a password, a server must be a principal. That is, it must have an account in the registry. There are two ways a server becomes a principal:

• The server inherits the login context, including the principal identity, of the user who calls it. In this case there is no need for server key management as such. The key is derived from the user's password, and the user is responsible for the key management. The need for this is not likely to arise over the server's usually brief lifetime.

• The server gets its own registry account. This is done by a system administrator running the rgyedit command with the ktadd subcommand. This process, consisting of two separate steps:
  
  Adding the account
  Creating the servers key

A server stores its password in a local key table file, called the keytab file. The keytab file facilitates the entry of passwords when the server logs in to DCE.

You must create an entry in the keytab file for each instance of a server that runs under its own identity. You can also create separate keytab files for each instance of the server, but the recommended method is a single keytab file.

The ktadd subcommand of the registry editor is used to create the keytab file and to add the password of the principal to the keytab file. In the following example:
rgy_edit=> ktadd -p mathxs -pw secret -f ktabfile

the ktadd command created the keytab file ktabfile to store the password secret for the principal mathxs.

You can also use the -a and the -r options of ktadd to generate the password randomly.

The ktadd subcommand of rgyedit is discussed in more detail in the Security part of the OpenEdition Distributed Computing Environment Base Services MVS: Administration Guide (SC09-1485).

---

### Securing the Keytab File:

Since the keytab file contains the password of the server principal, it must be protected from access by unauthorized users.

The keytab file is an HFS file and can be protected by manipulating the permission bits of the file (using the chmod command).

Only the following users must have read and write access to the keytab file:

- The root user.
- The address space running the server that uses that keytab file.

---

### Configuring the CDS Namespace

Configuring the CDS namespace entries for the server involves:

- Creating the CDS directories that will make up the pathname to the mathxs object.
- Giving access to these namespace entries.

The procedures described here must be performed by the DCE administrator (sometimes referred to as the cell administrator or cell_admin).

#### Creating a CDS Directory

You can design the structure of the CDS namespace entry in a manner that best suits your needs. The structure of the namespace entry is determined by the directory or directories that you create as containers for your objects.

The CDS directory entry that will contain the mathxs object must first be created using the CDS control program (cdscp). Depending on your requirements and preference, you can decide on either a simple or a lengthy pathname.

This example will use a simple pathname, with only one directory path to the mathxs object. The steps are:

- Run the CDS control program
- At the prompt enter the following:
  
  cdscp=> create dir /./mathxdir

  where mathxdir is the CDS directory that will hold the mathxs object entry.

#### Creating a CDS Object

---

Chapter 11. Configuring an Application Client and Server 189
The following steps describe how to create an object called mathxs in the mathxdir directory. This example will use a simple pathname, with only one directory path to the mathxs object. The steps are:

- Run the CDS control program.
- At the prompt enter the following:

  cdscp=> create obj ./mathxdir/mathxs

  where mathxs is the object name.

Access to the Namespace Entries

The mathxs principal must be given the appropriate access privileges (also known as permissions) to the namespace entries of the CDS.

The mathxs principal requires the read and write permissions on the mathxs object entry.

The DCE clients who will access the CDS require the read permission on the mathxs object entry.

You can specify the permissions to the directory and object namespace entries of the CDS using the ACL Editor. In a directory, the initial object ACL is used to set the ACL permissions for new object entries that will be created in that directory. For more information on the permissions supported by CDS, refer to the OpenEdition Distributed Computing Environment Base Services MVS: Administration Guide (SC09-1485).

Giving the mathxs principal appropriate permissions to the CDS namespace entry is done by running the ACL editor on the CDS mathxs object.

```
Note: Make sure that you have exited rgyedit before executing acledit.
```

- Run the ACL editor and use the -e option of the ACL Editor to modify the ACL of the mathxs object entry in the CDS namespace.

```
Note: Type the following command and parameters in as one string.

ACLEDIT -e ./mathxdir/mathxs

read and write permissions for the mathxs principal.
sec_acl_edit> modify user:mathxs:rw
sec_acl_edit> exit
```
11.3 OpenEdition DCE Environment Variables

An environment variable is a variable that stores information such as a name about a particular environment. You can customize the runtime environment for your DCE application by setting certain DCE specific environment variables. As with OSF DCE, IBM OpenEdition DCE Base Services MVS accepts configuration parameters using environment variables. In many cases, your applications do not need to set any environment variables, in which case, they use the defaults set up on your system.

There are five environment variables that you need to consider for your DCE applications.

- **NLSPATH**
  Points to the catalog of DCE messages. This must be set to use the `dce-error_inq_text()` API so that IBM OpenEdition DCE Base Services MVS can locate the catalog of DCE messages. Normally, this variable is set by the DCE administrator as a default for your system. If not, you should set this variable to `/opt/dcelocal/lib/nls/msg/En_US.IBM-1047` prior to running your DCE applications.

- **RPC_DEFAULT_ENTRY**
  Specifies the default entry in the name service database that the RPC NSI import and lookup routines use as a starting point to search for binding information for a compatible server. An application that uses a default entry name must define this environment variable. The DCE runtime does not provide a default.

- **RPC_DEFAULT_ENTRY_SYNTAX**
  Specifies the syntax of the name provided in the `RPC_DEFAULT_ENTRY` environment variable. In addition, it provides the syntax for those RPC NSI routines that allow a default value for the name syntax parameter. If you do not define `RPC_DEFAULT_ENTRY_SYNTAX`, the DCE runtime uses the `rpc_c_ns_syntax_dce` name syntax. It can be set to one of the following values:

  1. Use the default value
  2. Unknown
  3. Use DCEdns syntax
  4. Use DCE syntax (the default value)
  5. Use ISO OSI X.500 syntax
  6. Use DOD Internet Domain Name Server
  7. Use a UUID string

- **_EUV_SVC_API_DUMPS**
  Specifies if a CEEDUMP occurs if invalid parameters are present in any DCE API in your program. It can be one of the following values:

  1. Dumping is enabled
  2. Dumping is disabled


- **_EUV_RPC_DYNAMIC_POOL**
  Specifies whether a dynamic pool of executor threads is created. It can be one of the following values:
1. Dynamic pool is used (the default on IBM OpenEdition DCE Base Services MVS)
2. Static pool is used (based on the OSF model)

- **_EUV_EXC_DUMPS**

  Specifies if a dump is taken on uncaught exceptions. It can be one of the following values:

  1. Dumping enabled (the default value)
  2. Dumping disabled

There are also additional environment IBM OpenEdition DCE Base Services MVS variables that are used for serviceability and debugging purposes.

Table 7 lists the OpenEdition DCE environment variables, including the possible values of each. This list does not include the Application Support server-specific environment variables.

<table>
<thead>
<tr>
<th>Table 7 (Page 1 of 5). OpenEdition DCE Environment Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
</tr>
<tr>
<td><strong>POSIX Environment Variables Used by DCE</strong></td>
</tr>
</tbody>
</table>
| NLSPATH | A POSIX environment variable used by DCE that sets the search path for message catalogs. This environment variable is optional if only messages in the English language are required. The default path appears below. (The %L is the LANG environment variable which has a default value of En_US, and %N is the catalog name.)
  
  /usr/lib/nls/msg/En_US.IBM-1047/%N:
  /usr/lib/nls/msg/%L/%N:
  /usr/lib/nls/msg/prime/%N |
| TZ | Sets the time zone value used by DTS. This variable denotes the time zone wherein the user, process or machine runs. It can be a POSIX TZ value or a time zone file. The time zone files are stored in /etc/zoneinfo. By default, DTS services use the localtime time zone file (which is created during the installation of DCE). If the TZ environment variable is not set, the localtime file is used. |
| **Client and Server Controls** |
| BIND_PE_SITE | Determines if the Security server is looked up from the namespace or by reading the pe_site file. The acceptable values are:
  
  1. The Security server is looked up by reading the pe_site file.
  
  0. The Security server is looked up through a regular CDS query (the default action). |
<p>| KRBSCCNAME | Specifies the name of the user’s Security credentials cache file. When the user logs in to DCE, the sec_login_set_context API is issued and the Security service changes the value of KRBSCCNAME. This environment variable is automatically managed within the DCE library and should not be set by the user. |
| RPC_DEFAULT_ENTRY | Designates the namespace entry that is used as a starting point for searching binding information of compatible servers by import and lookup routines. There is no default value. |</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPC_DEFAULT_ENTRY_SYNTAX</td>
<td>Specifies the syntax of the name provided in the environment variable RPC_DEFAULT_ENTRY. It is also used by the NSI routines which allow a default value for the name syntax parameter. Valid values are defined in the include file <code>&lt;dce/rpcbase.idl&gt;</code>:&lt;br&gt;0  use default&lt;br&gt;1  unknown&lt;br&gt;2  DECdns&lt;br&gt;3  DCE (the default value)&lt;br&gt;4  ISO OSI X.500&lt;br&gt;5  DOD Internet Domain Name Server&lt;br&gt;6  UUID string</td>
</tr>
<tr>
<td>_EUV_HOME</td>
<td>Used to override the home directory value specified in the POSIX segment of a RACF userid. The default value is the home directory.</td>
</tr>
<tr>
<td>_EUV_ENVAR_FILE</td>
<td>Used to override the envar file name. The default value is $HOME/envar.</td>
</tr>
<tr>
<td>_EUV_RPC_COMM_TIMEOUT</td>
<td>Used to override the communication timeout default value. The timeout value specifies the relative amount of time a client spends attempting to communicate with a server. The timeout value can be any integer from 0 (zero) to 10, which is the same range of integers accepted by the <code>rpc_mgmt_set_com_timeout</code> API. These integers represent a relative amount of time to spend establishing a client-server relationship. The values are:&lt;br&gt;0  Attempts to communicate for 1 second.&lt;br&gt;1  Attempts to communicate for 2 seconds.&lt;br&gt;2  Attempts to communicate for 4 seconds.&lt;br&gt;3  Attempts to communicate for 8 seconds.&lt;br&gt;4  Attempts to communicate for 15 seconds.&lt;br&gt;5  Attempts to communicate for 30 seconds.&lt;br&gt;6  Attempts to communicate for 60 seconds.&lt;br&gt;7  Attempts to communicate for 120 seconds.&lt;br&gt;8  Attempts to communicate for 240 seconds.&lt;br&gt;9  Attempts to communicate for 480 seconds.&lt;br&gt;10  Attempts to communicate infinitely.</td>
</tr>
<tr>
<td>_EUV_ECHO_STDIN</td>
<td>Used by cdscp, rpccp, dtscp, acl_edit and rgy_edit. Intended for batch execution of these interactive utilities. Echoes input commands to the standard output file. Valid values are:&lt;br&gt;1  Enabled&lt;br&gt;0  Disabled&lt;br&gt;The default value is 0.</td>
</tr>
<tr>
<td>_EUV_SEC_KRBSCCNAME_FILE</td>
<td>Specifies the file that contains the KRB5CCNAME environment variable. The KRBSCCNAME environment variable contains the name of the Security credentials cache file. This environment variable enables a user to switch between multiple user identities. Primarily intended for non-shell environments (for example TSO, batch). The default value is $HOME/krb5ccname.</td>
</tr>
</tbody>
</table>
### Table 7 (Page 3 of 5). OpenEdition DCE Environment Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| `_EUV_EXC_SW_DUMPS`                     | A client and server control environment variable that specifies if a dump is taken during an exception raised by software. The valid values are:  
  0  No dump is taken for an exception.  
  1  A dump is only taken for an uncaught exception (if no CATCH or CATCHALL clause exists). This is the default value.  
  2  A dump is taken in all cases except for an explicit catch of an exception (if no CATCH clause exists). |
| `_EUV_EXC_ABEND_DUMPS`                  | A client and server control environment variable that specifies if a dump is taken during an ABEND exception. The valid values are:  
  0  No dump is taken for an exception.  
  1  A dump is only taken for an uncaught exception (if no CATCH or CATCHALL clause exists).  
  2  A dump is taken in all cases except for an explicit catch of an exception (if no CATCH clause exists). This is the default value. |
| `_EUV_FTRACE`                           | Reserved: Activates function tracing. Valid values are:  
  1  Enabled  
  0  Disabled |
| `_EUV_CDS_CLIENT_CLERK_LINK_TIMEOUT`   | Sets the inactivity time limit used by the clerk before flushing client binding information. The value is in minutes. The default value is 20 minutes. |
| **Server Controls**                     | **Server Controls**                                                          |
| `_EUV_RPC_ACL_FILE`                     | Specifies the name of the ACL database file used by a server. No default exists. |
| `_EUV_RPC_DYNAMIC_POOL`                 | Specifies whether a dynamic pool of executor threads is created. Valid values are:  
  1  Dynamic pool used (the MVS default value).  
  0  Static pool used (the original OSF model). |
| `_EUV_RPC_MAX_THRESHOLD`                | Thread pool damping control.                                                |
| `_EUV_RPC_MIN_THRESHOLD`                | Thread pool damping control.                                                |
| RPC_SUPPORTED_PROTSEQS                  | Reserved: OSF/DCE 1.0.3.                                                    |
| RPC_RESTRICTED_PORTS                    | Reserved: OSF/DCE 1.0.3.                                                    |
| **MVS Kernel Controls**                | **MVS Kernel Controls**                                                     |
| `_EUV_RACF_FACILITY_NAME`              | Specifies the name of the RACF facility that DCEKERN looks up to determine if a TSO user has DCEKERN start and stop permission. The default value is DCEKERN.START.REQUESTS. |
| **Message Controls**                    | **Message Controls**                                                         |
Table 7 (Page 4 of 5). OpenEdition DCE Environment Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_EUV_SVC_MSG_LEVEL</td>
<td>A message control environment variable that sets the minimum severity level of messages that are actually logged and displayed. The valid values are:</td>
</tr>
<tr>
<td></td>
<td><strong>NONE</strong></td>
</tr>
<tr>
<td></td>
<td>No messages are logged.</td>
</tr>
<tr>
<td></td>
<td><strong>FATAL</strong></td>
</tr>
<tr>
<td></td>
<td>Only fatal messages are logged. This value controls messages about non-recoverable errors. Usually, these messages are issued when a certain degree of permanent loss or damage occurs, such as the corruption of the database.</td>
</tr>
<tr>
<td></td>
<td><strong>ERROR</strong></td>
</tr>
<tr>
<td></td>
<td>Only error and fatal messages are logged. This value controls messages about unexpected events that are recoverable or that can be corrected by manual intervention.</td>
</tr>
<tr>
<td></td>
<td><strong>USER</strong></td>
</tr>
<tr>
<td></td>
<td>Only user, error, and fatal messages are logged. This value controls messages about errors detected in the use of a DCE API.</td>
</tr>
<tr>
<td></td>
<td><strong>WARNING</strong></td>
</tr>
<tr>
<td></td>
<td>Only warning, user, error, and fatal messages are logged. This value controls messages about one of the following conditions:</td>
</tr>
<tr>
<td></td>
<td>• An error occurred that was automatically corrected by the program or system.</td>
</tr>
<tr>
<td></td>
<td>• A condition was detected which may be an error depending on whether the effects of the condition are acceptable.</td>
</tr>
<tr>
<td></td>
<td>• A condition exists that if left uncorrected will eventually result in an error.</td>
</tr>
<tr>
<td></td>
<td><strong>NOTICE</strong></td>
</tr>
<tr>
<td></td>
<td>Only key informational messages are logged, along with all warning, error, user, and fatal messages. This value controls informational messages about major events such as the startup of a server.</td>
</tr>
<tr>
<td></td>
<td><strong>VERBOSE</strong></td>
</tr>
<tr>
<td></td>
<td>All messages are logged. This is the default action. This value controls informational messages about events which are important in monitoring DCE, such as the creation and deletion of RPC endpoints.</td>
</tr>
<tr>
<td>_EUV_SVC_MSG_LOGGING</td>
<td>A message control environment variable that controls where the messages are routed and displayed. Messages are routed to different destinations depending on whether the environment is shell, TSO, or batch. The valid values are:</td>
</tr>
<tr>
<td></td>
<td><strong>NO_LOGGING</strong></td>
</tr>
<tr>
<td></td>
<td>All messages are suppressed.</td>
</tr>
<tr>
<td></td>
<td><strong>STDOUT_LOGGING</strong></td>
</tr>
<tr>
<td></td>
<td>VERBOSE, NOTICE, and WARNING messages are routed to the standard output file, either the screen or the standard output DD card, while error and other messages are routed to the standard error file, either the screen or the standard error DD card. This is the default value.</td>
</tr>
<tr>
<td></td>
<td><strong>CONSOLE_LOGGING</strong></td>
</tr>
<tr>
<td></td>
<td>NOTICE, ERROR, and FATAL messages are routed to the operator console. All other messages are routed to the standard output file, either the screen or the standard output DD card.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| _EUV_SVC_API_DUMPS          | A message control environment variable that specifies if a dump is taken on parameter errors to public APIs. The valid values are:  
  1  Dumping is enabled. This is the default value.  
  0  Dumping is disabled. |

### Debug Controls

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVC_CDS_DBG</td>
<td>Sets the CDS component debug level.</td>
</tr>
<tr>
<td>SVC_DTS_DBG</td>
<td>Sets the DTS component debug level.</td>
</tr>
<tr>
<td>SVC_PLT_DBG</td>
<td>Sets the platform component debug level.</td>
</tr>
<tr>
<td>SVC_RPC_DBG</td>
<td>Sets the RPC component debug level.</td>
</tr>
<tr>
<td>SVC_SEC_DBG</td>
<td>Sets the security component debug level. Valid values are 1 to 9.</td>
</tr>
</tbody>
</table>
| _EUV_SVCDBG_MSG_LOGGING  | Turns debug tracing on and off. Valid values are:  
  1  On  
  0  Off (the default value). |

### DCECONF Controls

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_EUV_CFG_CELL_NAME</td>
<td>Specifies the default cell name on the configuration panel. There is no default.</td>
</tr>
<tr>
<td>_EUV_CFG_CELL_ID</td>
<td>Specifies the default cell administrator id on the configuration panel. The default value is cell_admin.</td>
</tr>
<tr>
<td>_EUV_CFG_CDSMACHINE_NAME</td>
<td>Specifies the default cdsmachine name on the configuration panel. There is no default.</td>
</tr>
<tr>
<td>_EUV_CFG_SEC_MACHINENAME</td>
<td>Specifies the default secdmachine name on the configuration panel. There is no default.</td>
</tr>
</tbody>
</table>
| _EUV_CFG_INFORM_LEVEL | Specifies the level of informational data displayed during configuration. Valid values are:  
  0  Only progress messages are displayed (the default value).  
  1  Progress messages and commands are displayed.  
  2  Progress messages, commands and command output are displayed. |
| _EUV_CFG_LOG_FILE     | Specifies the default history log file name. The default value is $HOME/dceconf.log. |

### IDL Compiler Controls

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_EUV_IDL_PRELINKER_NAME</td>
<td>Specifies a module of the &amp;c. preprocessor. The default value is EDCDC120.</td>
</tr>
<tr>
<td>_EUV_IDL_WORKDA_UNIT</td>
<td>Specifies the work dataset unit for internal C compiler files. The default value is VIO.</td>
</tr>
</tbody>
</table>
Appendix A. JCL and Source Listings for MATHX and SBIND Examples

A.1 MATHX JCL and Source Listings

The following is sample JCL and source listings for the MATHX example. This example is a simple math program without security. It includes the JCL to compile, link-edit and run the client, manager, and server application code. After the JCL there is sample Source for the client, server, and manger using explicit binding.

A.1.1 Compile MATHX Client and MATHX Client Stub

```
//JOB CARD JOB (999,POK), 'USERID'.NOTIFY=USERID,
//      CLASS=A,MSGCLASS=H,REGION=5000K,
//      MSGLEVEL=(1,1)
//******************************************************
// JCL TO COMPILE THE MATHX CLIENT
//******************************************************
//MATHXC EXEC DCECC,
// CPARM='SO,LO,NOMAR,NOSEQ'
//COMPILE.SYSLIN DD DSN=USERPRFX.<APPLNM>.OBJ(MATHXC),DISP=SHR
//COMPILE.USERLIB2 DD DSN=USERPRFX.<APPLNM>.H,DISP=SHR
//COMPILE.SYSIN DD DSN=USERPRFX.<APPLNM>.C(MATHXC),DISP=SHR
//******************************************************
// JCL TO COMPILE THE CLIENT STUB. NOT NECESSARY WITH ECIP 1
//******************************************************
//MATHXCS EXEC DCECC,
// CPARM='SO,LO,NOMAR,NOSEQ'
//COMPILE.SYSLIN DD DSN=USERPRFX.<APPLNM>.OBJ(MATHXCS),DISP=SHR
//COMPILE.USERLIB2 DD DSN=USERPRFX.<APPLNM>.H,DISP=SHR
//COMPILE.SYSIN DD DSN=USERPRFX.<APPLNM>.C(MATHXCS),DISP=SHR
```

Figure 104. JCL to compile the MATHXC client code.
A.1.2 Compile MATHXS Server and MATHX Server Stub

```plaintext
//JOBCLASS JOB (999,POK),’USERID’,NOTIFY=USERID,
// CLASS=A,MSGCLASS=T,REGION=5000K,
// MSGLEVEL=(1,1)
//NEWLK JCLLIB ORDER=(DCE.SYS.PROCLIB)
//*****************************************************************************
//*** JCL TO COMPILE THE MATHX SERVER ***
//*****************************************************************************
//DTSNULLP EXEC DCECC,INFILE=’USERPRFX.<APPLNM>.C(MATHXS)’,
// OUTFILE=’USERPRFX.<APPLNM>.OBJECT(MATHXS)’,
// OUTCLASS=’*’,CPARM=’SOURCE’,
// DEFS=’DEF(SAA,MVS)’
//USERLIB DD DSN=DCE.V1R1M0.SEUVHDR,DISP=SHR
// DD DSN=USERPRFX.<APPLNM>.HEADER,DISP=SHR
/*SYSCPRT DD DSN=USERPRFX.LIST3,DISP=SHR
//*****************************************************************************
//*** JCL TO COMPILE THE MATHX SERVER STUB ***
//*****************************************************************************
//MATHXSS EXEC DCECC,
// CPARM=’SO,LO,NOMAR,NOSEQ’
//COMPILE.SYSLIN DD DSN=USERPRFX.<APPLNM>.OBJ(MATHXSS),DISP=SHR
//COMPILE.USERLIB2 DD DSN=USERPRFX.<APPLNM>.H,DISP=SHR
//COMPILE.SYSIN DD DSN=USERPRFX.<APPLNM>.C(MATHXSS),DISP=SHR
```

Figure 105. JCL to Compile the MATHXS Server Code.

A.1.3 Compile MATHXM Manager

```plaintext
//JOBCLASS JOB (999,POK),’USERID’,NOTIFY=USERID,
// CLASS=A,MSGCLASS=T,REGION=5000K,
// MSGLEVEL=(1,1)
//*****************************************************************************
//*** JCL TO COMPILE THE MATHX MANAGER ***
//*****************************************************************************
//MATHXM EXEC DCECC,
// CPARM=’SO,LO,NOMAR,NOSEQ’
//COMPILE.SYSLIN DD DSN=USERPRFX.<APPLNM>.OBJ(MATHXM),DISP=SHR
//COMPILE.USERLIB2 DD DSN=USERPRFX.<APPLNM>.H,DISP=SHR
//COMPILE.SYSIN DD DSN=USERPRFX.<APPLNM>.C(MATHXM),DISP=SHR
```

Figure 106. JCL to Compile the MATHXM Manager Code.
A.1.4 LINKEDIT MATHX Client

//JOBCARD JOB (999,POK),'USERID'.NOTIFY=USERID,
//   CLASS=A,MSGCLASS=H,REGION=5000K,
//   MSGLEVEL=(1,1)
Ichกรรมนี้ต้องกำหนดให้ถูกต้อง
// JCL TO LINK THE MATHX CLIENT

Figure 107. JCL to LINKEDIT the MATHX Client Code.

A.1.5 LINKEDIT MATHX Server

//JOBCARD JOB (999,POK),'USERID'.NOTIFY=USERID,
//   CLASS=A,MSGCLASS=H,REGION=5000K,
//   MSGLEVEL=(1,1)
Ichกรรมนี้ต้องกำหนดให้ถูกต้อง
// JCL TO LINK THE MATHX SERVER

Figure 108. JCL to LINKEDIT the MATHX Server Code
A.1.6 RUN MATHX Client

//JOBcard JOB (999,POK),USERID,NOTIFY=USERID,
//  CLASS=A,MSGCLASS=H,REGION=5000K,
//  MSGLEVEL=1,1

********************************************************************
//* JCL TO RUN THE MATHX CLIENT *
********************************************************************/

//LOGIN EXEC PROC=DCELOGIN,
//  PARMS='MATHX MATHX'
//RUN EXEC PGM=MATHX,PARM='POSIX(ON)/20 25'
//STEPLIB DD DSN=USERPRFX.<APPLNM>.LOAD,DISP=SHR
// DD DSN=CEE.V1R3M0.SCEERUN,DISP=SHR
// DD DSN=DCE.SEUVEXEC,DISP=SHR
//SYSIN DD *
//SYSERR DD SYSOUT=* 
//SYSPRINT DD SYSOUT=* 

Figure 109. JCL to RUN the MATHX Client

A.1.7 RUN MATHX Server

//JOBcard JOB (999,POK),USERID,NOTIFY=USERID,
//  CLASS=A,MSGCLASS=T,REGION=5000K,
//  MSGLEVEL=1,1

********************************************************************
//* JCL TO RUN THE MATHX SERVER *
********************************************************************/

//LOGIN EXEC PROC=DCELOGIN,
//  PARMS='mathxs mathxs'
//DCEENV DD DSN=MATHX.DCEENV,DISP=SHR
//RUN EXEC PGM=MATHXS,REGION=2048K
// ENVVAR DD DSN=DCE.V1R1M0.SEUVENV(MATHX),DISP=SHR
//DCEENV DD DSN=MATHX.DCEENV,DISP=SHR
//MAPFILE DD DSN=DCE.V1R1M0.DCELOCAL.DTSCPMAP,DISP=SHR
//CLOCTNET DD DSN=DCE.V1R1M0.SEUVCPT(C037T819),DISP=SHR
//CNETTLOC DD DSN=DCE.V1R1M0.SEUVCPT(C819T037),DISP=SHR
//SEUVCAT DD DSN=DCE.V1R1M0.SEUVCAT,DISP=SHR
//SEUVCFG DD DSN=DCE.V1R1M0.SEUVCFG,DISP=SHR
//DCECFDB DD DSN=DCE.V1R1M0.DCELOCAL.DCECFDB,DISP=SHR
//V5SRVTAB DD DSN=DCE.V1R1M0.DCELOCAL.V5SRVTAB,DISP=SHR
//STEPLIB DD DSN=USERPRFX.<APPLNM>.LOAD,DISP=SHR
// DD DSN=SYS1.EDC.V2R1M0.SEDCLINK,DISP=SHR
// DD DSN=SYS1.PLL.V2R3M0.SIBMLINK,DISP=SHR
//SYSIN DD *

//SYSERR DD SYSOUT=* 
//SYSPRINT DD SYSOUT=* 

Figure 110. JCL to RUN the MATHX Server
A.2 Source for MATHX Example

The following are sample client/server and manager code for the MATHX example. The example very simply and without security adds 2 numerical values and returns the result to the client. This example uses explicit binding.

A.2.1 Client Source (MATHXC)

```c
#include <stdio.h>
#include <stdlib.h>
#include <dce/rpc.h>
#include "errchk.h"
#include "mathx.h"

#define ENTRY_NAME "/:.servers/pok/mathxs" /* Server entry name. */

int main ( int argc, char *argv[]) {
    long int i, numv, sum;
    value_array_t va;
    rpc_binding_handle_t bh;
    unsigned32 status;
    rpc_ns_handle_t imp_ctxt;

    /* Check if the user passed the minimum number of parameters. */
    if ( argc < 3 ) {
        printf ( "Usage : %s <values to be added>\n", argv[0]);
        fflush(NULL);
        exit ( 1);
    }
}
```

Figure 111 (Part 1 of 3). Basic Client Code (MATHXC)
/* Set up the context to import the bindings. */

printf ("Set up context to import the bindings...\n");
fflush(NULL);

type = 5;

rpc_ns_binding_import_begin (  
    rpc_c_ns_syntax_dce,  
    ENTRY_NAME,  
    mathx_v1_0_c_ifspec,  
    NULL,  
    &imp_ctxt,  
    &status );  
  ERRCHK ( status );

/* Get the first binding handle. */

printf ("Get the first binding handle...\n");
fflush(NULL);

type = 6;

rpc_ns_binding_import_next (  
    imp_ctxt,  
    &bh,  
    &status );  
  ERRCHK ( status );

/* Get values from the command line and load them in array va. */

printf ("Get values and load into the array...\n");
fflush(NULL);

for ( i = 0; i < MAX_VALUES && i < argc - 1; i++ )  
    va[ i ] = atoi ( argv[ i + 1 ] );
numv = i;

/* Call add passing handle, values, number of values and pointer to result. */

printf ("Call add routine, pass handle and values, and point to result...\n");
fflush(NULL);

add ( bh, va, ( long )numv,  
    &sum );  
printf ("The sum is %d\n", sum );
fflush(NULL);

Figure 111 (Part 2 of 3). Basic Client Code (MATHXC)
/***************************************************************************/
/* Release the context. */
(['/** Release the context...\n*/

printf ("Release the context...\n");
fflush(NULL);

rpc_ns_binding_import_done (  
    &imp_ctxt,  
    &status );
ERRCHK (status);
}

Figure 111 (Part 3 of 3). Basic Client Code (MATHXC)
A.2.2 Server Source (MATHXS)

```c
#include <stdio.h>
#include <stdlib.h>
#include <dce/rpc.h>
#include <pthread.h>
#include <dce/dce_cf.h>
#include <dce/exc_handling.h>
#include "errchk.h"
#include "mathx.h"

#define MAX_CONC_CALLS_PROTSEQ 1 /* Max concurrent calls per protocol. */
#define MAX_CONC_CALLS_TOTAL 2 /* Max concurrent calls total. */
#define ENTRY_NAME "/.:/servers/pok/mathxs" /* Server entry name. */

int main ( int argc, char *argv[])
{
    rpc_binding_vector_t *bv_p;
    unsigned32 status;
    int  i;
    char *string_binding;
    char SERVER = "MATHXS";
    
    /* Register interface/epv associations with RPC runtime. */
    
    printf("Registering server interface with RPC runtime...\n");
    fflush(NULL);

    rpc_server_register_if (
        mathx_v1_0_s_ifspec,
        NULL,
        NULL,
        &status );
    ERRCHK ( status );
```

Figure 112 (Part 1 of 3). Basic Server Code (MATHXS)
printf("Use supported protocol sequences...\n");
fflush(NULL);

rpc_server_use_protseq(
    "ncadg_ip_udp",
    MAX_CONC_CALLS_PROTSEQ,
    &status);
ERRCHK ( status );

printf("Get binding handle vector from RPC runtime...");
fflush(NULL);

rpc_server_inq_bindings(
    &bv_p,
    &status);
ERRCHK ( status );

printf("( %s )Bindings:\n",SERVER);
fflush(stdout);
for (i=0; i< bv_p->count; i++)
{
    rpc_binding_to_string_binding(
        bv_p->binding_h[i],
        &string_binding,&status);
    ERRCHK ( status );

    printf("%s\n", (char *)string_binding);
    fflush(stdout);
}

rpc_string_free(
    &string_binding,&status);
ERRCHK ( status );

Figure 112 (Part 2 of 3). Basic Server Code (MATHXS)
/* Register server endpoints with endpoint mapper. */

printf("Registering server endpoints with endpoint mapper (RPCD)...\n");
fflush(NULL);

rpc_ep_register (
    mathx_v1_0_s_ifspec,
    bv_p,
    NULL,
    (unsigned_char_t *)&"Explicit math server, version 1.0",
    &status);
ERRCHK (status);

/***************************************************************************/
/* Export binding information to the namespace. */
/***************************************************************************/

printf("Exporting server bindings into CDS namespace...\n");
fflush(NULL);

rpc_ns_binding_export (
    rpc_c_ns_syntax_dce,
    ENTRY_NAME,
    mathx_v1_0_s_ifspec,
    bv_p,
    NULL,
    &status);
ERRCHK (status);

/***************************************************************************/
/* Listen for service requests. */
/***************************************************************************/

TRY {
    printf("Server %s listening...\n", ENTRY_NAME);
    fflush(NULL);

    rpc_server_listen (
        MAX_CONC_CALLS_TOTAL,
        &status);
    ERRCHK (status);
}
CATCH_ALL {
    printf(" CATCH ALL \n");
}
ENDTRY;

Figure 112 (Part 3 of 3). Basic Server Code (MATHXS)
A.2.3 Manager Source (MATHXM)

/******************************************************************************/
/* Module : MATHXM */
/* Purpose : Server manager code. The add procedure that will be */
/* called from the client MATHXC. */
/******************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "mathx.h"

/******************************************************************************/
/* Procedure : add */
/* Purpose : Add numv values passed in array value_a placing the sum in */
/* total. Uses explicit binding. */
/******************************************************************************/

void add ( rpc_binding_handle_t bh,
        value_array_t value_a,
        long numv,
        long *total )
{
    int i;

    /**************************************************************************/
    /* Display the parameters that have been passed. */
    /**************************************************************************/

    printf ( "Add has been called with numv = %ld\n", numv );
    fflush(NULL);

    for ( i = 0; i < numv; i++ )
        printf ( "Value_a[%d] = %ld\n", i, value_a[i] );

    /**************************************************************************/
    /* Zero the accumulator. */
    /**************************************************************************/

    printf ( "Zero the accumulator \n" );
    fflush(NULL);

    *total = 0;

    /**************************************************************************/
    /* Add the values in the array of size numv placing the result in total. */
    /**************************************************************************/

    printf ( "Add the values and place results in total...\n" );
    fflush(NULL);

    for ( i = 0; i < MAX_VALUES && i < numv; i++ )
        *total += value_a[i];
}

Figure 113. Basic Manager Code (MATHXM)
A.2.4 MATHX OUTPUT

************* TOP OF DATA FOR MATHX SERVER (MATHXS)***************
Use supported protocol sequences...
Registering server interface with RPC runtime...
Get binding handle from RPC runtime...
( MATHXS )Bindings:
ncadg_ip_udp:9.12.13.69 4667
Registering server endpoints with endpoint mapper (RPCD)...
Exporting server bindings into CDS namespace...
Server ./:/servers/pok/mathxs listening...
************************** BOTTOM OF DATA ****************************

************* TOP OF DATA FOR MATHX CLIENT (MATHXC)***************
Set up context to import the bindings...
Get the first binding handle...
Get values and load into the array...
Call add routine, pass handle and values, and point to result...
The sum is 45
Release the context...
************************** BOTTOM OF DATA ****************************

Figure 114. PRINTF statements for MATHX Example
A.3 SBIND JCL and Source Listings

The following is sample JCL and source listings for the SBIND example. This example is a simple math program without security. It includes the JCL to compile, link-edit and run the client, manager, and server application code. After the JCL there is sample source for the client, server and manager using string binding.

A.3.1 Compile SBIND Client and SBIND Client Stub

```
//JOBCARD JOB (999,POK),'USERID',NOTIFY=USERID,
// CLASS=A,MSGCLASS=H,REGION=5000K,
// MSGLEVEL=(1,1)
//===================================================================
//* JCL TO COMPILE THE SBINDC CLIENT EXAMPLE
//===================================================================
//SBINDC EXEC DCECC,CPARM='SO,LO,NOMAR,NOSEQ'
//COMPILE.SYSLIN DD DSN=USERPRFX.<APPLNM>.OBJ(SBINDC),DISP=SHR
//COMPILE.USERLIB2 DD DSN=USERPRFX.<APPLNM>.H,DISP=SHR
//COMPILE.SYSIN DD DSN=USERPRFX.<APPLNM>.C(SBINDC),DISP=SHR
//===================================================================
//*JCL TO COMPILE THE CLINET STUB. NOT NECESSARY WITH ECIP 1
//===================================================================
//SBINDCS EXEC DCECC,CPARM='SO,LO,NOMAR,NOSEQ'
//COMPILE.SYSLIN DD DSN=USERPRFX.<APPLNM>.OBJ(SBINDCS),DISP=SHR
//COMPILE.USERLIB2 DD DSN=USERPRFX.<APPLNM>.H,DISP=SHR
//COMPILE.SYSIN DD DSN=USERPRFX.<APPLNM>.C(SBINDCS),DISP=SHR
```

Figure 115. JCL to Compile the SBINDC Client Code.

A.3.2 Compile SBIND Server and SBIND Server Stub

```
//JOBCARD JOB (999,POK),'USERID',NOTIFY=USERID,
// CLASS=A,MSGCLASS=H,REGION=5000K,
// MSGLEVEL=(1,1)
//===================================================================
//* JCL TO COMPILE THE SBIND SERVER EXAMPLE
//===================================================================
//SBINDS EXEC DCECC,
// CPARM='SO,LO,NOMAR,NOSEQ'
//COMPILE.SYSLIN DD DSN=USERPRFX.<APPLNM>.OBJ(SBINDS),DISP=SHR
//COMPILE.USERLIB2 DD DSN=USERPRFX.<APPLNM>.H,DISP=SHR
//COMPILE.SYSIN DD DSN=USERPRFX.<APPLNM>.C(SBINDS),DISP=SHR
//===================================================================
//* JCL TO COMPILE THE SBIND SERVER STUB
//===================================================================
//SBINDSS EXEC DCECC,
// CPARM='SO,LO,NOMAR,NOSEQ'
//COMPILE.SYSLIN DD DSN=USERPRFX.<APPLNM>.OBJ(SBINDSS),DISP=SHR
//COMPILE.USERLIB2 DD DSN=USERPRFX.<APPLNM>.H,DISP=SHR
//COMPILE.SYSIN DD DSN=USERPRFX.<APPLNM>.C(SBINDSS),DISP=SHR
```

Figure 116. JCL to Compile the SBINDS Server Code.
A.3.3 Compile SBINDM Manager

```jcl
//JOBCARD JOB (999,POK),'USERID',NOTIFY=USERID,
//   CLASS=A,MSGCLASS=H,REGION=5000K,
//   MSGLEVEL=(1,1)
//*****************************************************************************
//* JCL TO COMPILE THE SBIND MANAGER *
//--------------------------------------------------------------------------
//SBINDM EXEC DCECC,
//  CPARM='SO,LO,NOMAR,NOSEQ'
//COMPILE.SYSLIN DD DSN=USERPRFX.<APPLNM>.OBJ(SBINDM),DISP=SHR
//COMPILE.USERLIB2 DD DSN=USERPRFX.<APPLNM>.H,DISP=SHR
//COMPILE.SYSIN DD DSN=USERPRFX.<APPLNM>.C(SBINDM),DISP=SHR
```

Figure 117. JCL to Compile the SBINDM Manager Code.

A.3.4 LINKEDIT SBIND Client

```jcl
//JOBCARD JOB (999,POK),'USERID',NOTIFY=USERID,
//   CLASS=A,MSGCLASS=H,REGION=5000K,
//   MSGLEVEL=(1,1)
//*****************************************************************************
//* JCL TO LINK THE SBIND CLIENT *
//--------------------------------------------------------------------------
//LKSBINDC EXEC DCELK,OUTFILE='USERPRFX.<APPLNM>.LOAD'
//PLKED.USERLIB DD DSN=USERPRFX.<APPLNM>.OBJ,DISP=SHR
//PLKED.SYSIN DD *
#include USERLIB(SBINDC)
#include USERLIB2(DCESERVN)
//LKED.USERLIB DD DSN=USERPRFX.<APPLNM>.OBJ,DISP=SHR
//LKED.SYSPRINT DD SYSOUT=* 
//LKED.SYSIN DD *
   NAME SBINDC(R)
```

Figure 118. JCL to LINKEDIT the SBIND Client Code.
A.3.5 LINKEDIT SBIND Server

//JOBCARD JOB (999,POK), 'USERID'.NOTIFY=USERID,
// CLASS=A,MSGCLASS=H,REGION=5000K,
// MSGLEVEL=(1,1)
//*********************************************************************
//* JCL TO LINK THE SBIND SERVER *
//*********************************************************************
//LKSBINDS EXEC DCELK,OUTFILE='USERPRFX.<APPLNM>.LOAD'
//USERLIB DD DSN=USERPRFX.<APPLNM>.OBJ,DISP=SHR
//PLKED.SYSIN DD *
//INCLUDE USERLIB(SBINDS)
//INCLUDE USERLIB(SBINDM)
//INCLUDE USERLIB(SBINDSS)
//LKED.USERLIB DD DSN=USERPRFX.<APPLNM>.OBJ,DISP=SHR
//LKED.SYSPRINT DD SYSOUT=*  
//LKED.SYSIN DD *
//NAME SBINDS(R)

Figure 119. JCL to LINKEDIT the SBIND Server Code.

A.3.6 RUN SBIND Client

//JOBCARD JOB (999,POK), 'USERID'.NOTIFY=USERID,
// CLASS=A,MSGCLASS=T,REGION=5000K,
// MSGLEVEL=(1,1)
//*********************************************************************
//* JCL TO RUN THE SBIND CLIENT *
//*********************************************************************
//LOGIN EXEC PROC=DCELOGIN,
// PARMS='sbindc sbindc'
//RUN EXEC PGM=SBINDC,
// PARM='POSIX(ON) "ncadg_ip_udp" 9.12.13.69 "XXXX" 50 100'
// STEPLIB DD DSN=USERPRFX.<APPLNM>.LOAD,DISP=SHR
// DD DSN=CEE.V1R3M0.SCEERUN,DISP=SHR
// DD DSN=DCE.SEUVEXEC,DISP=SHR
// SYSSIN DD *
// SYSSERR DD SYSOUT=* 
// SYSPRINT DD SYSOUT=*  
// *DSNTRACE DD SYSOUT=* 

Figure 120. JCL to RUN the SBIND Client.
A.3.7 RUN SBIND Server

//JOBCARD JOB (999,POK),'USERID',NOTIFY=USERID,
// CLASS=A,MSGCLASS=H,
// MSGLEVEL=(1,1),REGION=OM
//*********************************************************************
//* JCL TO RUN THE STRING SERVER *
//*********************************************************************
//LOGIN EXEC PROC=DCELOGIN,
// PARMS='sbinds sbinds'
//DCEENV DD DSN=USERPRFX.DCEENV,DISP=SHR
//RUN EXEC PGM=SBINDS,PARM='POSIX(ON)/',REGION=OM
//STEPLIB DD DSN=USERPRFX.<APPLNM>.LOAD,DISP=SHR
// DD DSN=CEE.V1R3M0.SCEERUN,DISP=SHR
// DD DSN=DCE.SEUVEXEC,DISP=SHR
//SYSIN DD *
//SYSERR DD SYSOUT=* 
//SYSPRINT DD SYSOUT=* 
//DSNTRACE DD SYSOUT=* 

Figure 121. JCL to RUN the SBIND Server
A.4 Source for SBIND Example

The following are sample client/server and manager code for the SBIND example. The example very simply and without security adds 2 numerical values and returns the result to the client. This example uses string binding.

A.4.1 Client Source (SBINDC)

```c
#include <stdio.h>
#include <stdlib.h>
#include <dce/rpc.h>
#include "errchk.h"
#include "sbind.h"

int main ( int argc, char *argv[])
{
    long int i, numv, sum;
    value_array_t va;
    rpc_binding_handle_t bh;
    rpc_binding_handle_t global_handle;
    unsigned32 status;
    unsigned_char_t prot[80];
    unsigned_char_t ip[80];
    unsigned_char_t port[80];
    unsigned_char_t *string_binding = NULL;
    unsigned32 st;
    error_status_t c_error;
    error_status_t f_error;

    printf("Program: %s Arguments: %s %s %s\n", argv[0], argv[1], argv[2], argv[3rbrk.);
    fflush(NULL);
```

Figure 122 (Part 1 of 3). Client Code (SBINDC)
/***************************************************************************/
/* Check if the user passed the minimum number of parameters. */
/***************************************************************************/
if ( argc < 6 ) {

    printf
    ("Usage: %s <string binding> <values to be added>
    "%s", argv[0]);
    fflush(NULL);

    exit ( 1 );
}

strcpy(prot,argv[1]);
strcpy(ip,argv[2]);
strcpy(port,argv[3]);
/***************************************************************************/
/* String Binding Compose */
/***************************************************************************/
printf("compose 
    ");
fflush(NULL);
rpc_string_binding_compose(
    ",
    prot,
    ip,
    port,
    "",
    &string_binding,
    &status);
ERRCHK ( status );

printf("st=%d
    ");
fflush(NULL);

printf("string_binding = %s
    ");
fflush(NULL);

/***************************************************************************/
/* Generate a binding handle from the string binding in the command line. */
/***************************************************************************/
printf("Generate a binding handle
    ");
fflush(NULL);

rpc_binding_from_string_binding ( 
    string_binding,
    &bh,
    &status );
ERRCHK ( status );
Figure 122 (Part 3 of 3). Client Code (SBINDC)
A.4.2 Server Source (SBINDS)

```c
#include <stdio.h>
#include <stdlib.h>
#include <dce/rpc.h>
#include <pthread.h>
#include <dce/exc_handling.h>
#include "errchk.h"
#include "sbind.h"

#define MAX_CONC_CALLS_PROTSEQ 1 /* Max concurrent calls per protocol. */
#define MAX_CONC_CALLS_TOTAL 2 /* Max concurrent calls total. */

int main ( int argc, char *argv[] )
{
    int i;
    rpc_binding_vector_t *bv_p;
    unsigned32 status;
    char *str_bind_p;

    printf("Registering server interface with RPC runtime...

    rpc_server_register_if ( 
        sbind_v1_0_s_ifspec,
        NULL, 
        NULL, 
        &status );
    ERRCHK ( status );
}
```

Figure 123 (Part 1 of 3). Server Code (SBINDS)
/** Inform RPC runtime to use all supported protocol sequences. */

printf("Inform RPC runtime to use all supported prot...

fflush (NULL);

rpc_server_use_all_protseqs ( 
    MAX_CONC_CALLS_PROTSEQ, 
    &status );

ERRCHK ( status );

/** Get the binding handle vector from RPC runtime. */

printf("Get the binding handle vector from RPC runtime.

fflush (NULL);

rpc_server_inq_bindings ( 
    &bv_p, 
    &status );

ERRCHK ( status );

/** Register binding information with endpoint map. */

printf("\nRegistering server endpoints with endpoint mapper (RPCD)...

fflush (NULL);

rpc_ep_register ( 
    sbind_v1_0_s_ifspec, 
    bv_p, 
    NULL, 
    ( unsigned_char_t * )"String math server, version 1.0". 
    &status );

ERRCHK ( status );

/** Display string binding. */

for ( i = 0 ; i < bv_p->count ; i++ ) {

rpc_binding_to_string_binding ( 
    bv_p->binding_h[i], 
    &str_bind_p, 
    &status );

ERRCHK ( status );

printf ( "String binding is : %s\n", str_bind_p );

fflush (NULL);

Figure 123 (Part 2 of 3). Server Code (SBINDS)
rpc_string_free(
    &str_bind_p,
    &status);
    ERRCHK ( status );
}

rpc_server_listen (
    MAX_CONC_CALLS_TOTAL,
    &status );
    ERRCHK ( status );

Figure 123 (Part 3 of 3). Server Code (SBINDS)
A.4.3 Manager Source (SBINDM)

/* Module : SBINDM */
/* Purpose : Server manager code. Implements the add procedure that will be */
/* called from the client SBINDC */
/******************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "sbind.h"
/******************************************************************************/
/* Procedure : add */
/* Purpose : Add numv values passed in array value_a placing the sum in */
/* total. */
/******************************************************************************/

void add ( 
    rpc_binding_handle_t bh, 
    value_array_t value_a, 
    long numv, 
    long *total ) 
{ 
    int i; 

    /* Display the parameters that have been passed. */ 
    printf ( "%Add has been called with numv = %d\n", numv ); 
    fflush(NULL); 

    for ( i = 0; i < numv; i++ ) 
        printf ( "Value_a[ %d ] = %d\n", i, value_a[ i ] ); 
    fflush(NULL); 

    /* Zero the accumulator. */ 
    *total = 0; 

    /* Add the values in the array of size numv placing the result in total. */ 
    printf ("Add values in the array and place results in total...\n" ); 
    fflush(NULL); 

    for ( i = 0; i < MAX_VALUES && i < numv; i++ ) 
        *total += value_a[ i ]; 
}

Figure 124. Manager Code (SBINDM)
A.4.4 SBIND OUTPUT

************* TOP OF DATA FOR SBIND SERVER (SBINDS)***********************
Registering server interface with RPC runtime...
Inform RPC runtime to use all supported prot...
Get the binding handle vector from RPC runtime.
Registering server endpoints with endpoint mapper (RPCD)... 
String binding is : ncadg_ip_udp:9.12.13.69 4677
Server String listening...

************* BOTTOM OF DATA *************

************* TOP OF DATA FOR SBIND CLIENT (SBINDC)***********************
compose
st=76045504
string_binding = ncadg_ip_udp:9.12.13.69 4677
Generate a binding handle
Store binding handle in global varialbe
Load values in array va
V 0 = 50
V 1 = 100
Call add passing values and pointer to result
  Error Print Comerror 0 Faulterror 0
The sum is 150

************* BOTTOM OF DATA *************

Figure 125. PRINTF statements for SBIND Example
Appendix B. DCE Security Example

B.1 Server Module C Source

The following is the full C source code for the server module of the message box example.

```c
/* ************************************************** */
/* */
/* Module: mboxs.c */
/* */
/* Description: DCE RPC setup for the MessageBox application. */
/* */
/* Catches CTRL-C (for OS/2) or client shutdown */
/* */
/* 1. cleans up binding information */
/* */
/* 2. exports the mbox structure to a local file. */
/* */
/* */
/* ************************************************** */

/* ---------------------------------------- */
/* included headers */
/* ---------------------------------------- */
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <dce/rpc.h>
#include <dce/sec_login.h>
#include <dce/keymgmt.h>
#ifdef _WINDOWS
#include <dce/dcewin.h>
#endif
#ifdef MVS
#include <exc@hand.h>
#endif
#include "mbox.h"
#include "mboxcom.h"
```

Figure 126 (Part 1 of 9). Security Sample Server Code (mboxs.c)
/* defined constants */
#define MAX_CONC_CALLS PROTSEQ 2
#define MAX_CONC_CALLS TOTAL 4
#define IF_HANDLE MessageBox_v2_0_s_ifspec
#if defined(_AIX)
#define FNAME /tmp/message.box
#else
#define FNAME message.box
#endif
#define KEYTAB Message.key

/* prototypes */
int mbox_import ( const char * );
int mbox_export ( const char * );

/* Main */
int main (int argc, char *argv[]) {
    /* local variables */
    unsigned_char_t *server_name;
    rpc_binding_vector_t *bind_vector_p;
    sec_login_handle_t login_context;
    void *server_key;
    sec_login_auth_src_t auth_src;
    boolean32 identity_valid;
    boolean32 reset_passwd;
    unsigned32 status;
    #if !defined(_WINDOWS) && !defined(MVS)
    sigset_t sigset;
    pthread_t this_thread = pthread_self();
    #endif
    /* parse the command line parameters */
debug = 0;
if (argc > 1 && strcmp(argv[1], "-d")) {
    debug = 1;
}

/* ----------------------------------- */
/* setup thread exception handling */
/* ----------------------------------- */

#ifdef IBMOS2
    pthread_inst_exception_handler();
#endif

#if !defined(MVS)
    this_thread = pthread_self();
#endif

/* --------------------------------- */
/* setup identity in a login context */
/* --------------------------------- */

if (debug == 1) {
    printf("Setting Identity...\n");
    fflush(NULL);
}

sec_login_setup_identity(
    PRINCIPAL_NAME,
    sec_login_no_flags,
    &login_context,
    &status);
ERRCHK ( status );

/* ------------------------ */
/* Get Key From KEYTAB File */
/* ------------------------ */

if (debug == 1) {
    printf("Getting Key...\n");
    fflush(NULL);
}

sec_key_mgmt_get_key(
    rpc_c_authn_dce_secret,
    KEYTAB,
    PRINCIPAL_NAME,
    0,
    &server_key,
    &status);
ERRCHK ( status );

/* ---------------------- */
/* Validate the Identity */
/* ---------------------- */

if (debug == 1) {
    printf("Validate Identity...
    ");
    fflush (NULL);
}

identity_valid =
    sec_login_validate_identity(
        login_context,
        server_key,
        &reset_passwd,
        &auth_src,
        &status);
ERRCHK ( status );

/* ---------------- */
/* Set login context */
/* ---------------- */

if (debug == 1) {
    printf("Setting context...
    ");
    fflush (NULL);
}

sec_login_set_context(
    login_context,
    &status);
ERRCHK ( status );

/* ---------------- */
/* Free Key Storage */
/* ---------------- */

if (debug == 1) {
    printf("Freeing key storage...
    ");
    fflush (NULL);
}

sec_key_mgmt_free_key(
    &server_key,
    &status);
ERRCHK ( status );

Figure 126 (Part 4 of 9). Security Sample Server Code (mboxs.c)
/* check login OK and authenticated */
if (identity_valid &&
    (auth_src == sec_login_auth_src_network))
{
    printf("Logged in...
");
    fflush (NULL);
}
else
{
    printf("Failed to login...
");
    fflush (NULL);
    exit(1);
}

/* Register the authentication level which will be used */
if (debug == 1) {
    printf("Registering authentication information ...
");
    fflush (NULL);
}
rpc_server_register_auth_info (  
    PRINCIPAL_NAME,
    rpc_c_authn_default,
    NULL,
    NULL,
    &status
    );
ERRCHK ( status );

/* Register interface/epv associations with rpc runtime. */
if (debug == 1) {
    printf("Registering interface/epv ...
");
    fflush (NULL);
}
rpc_server_register_if ( IF_HANDLE, NULL, NULL, &status );
ERRCHK ( status );
if (debug == 1) {
    printf("Using all protocal sequences ...\n");
    fflush (NULL);
}

rpc_server_use_all_protseqs ( MAX_CONC_CALLS_PROTSEQ, &status );
ERRCHK ( status );

if (debug == 1) {
    printf("Getting binding handle ...\n");
    fflush (NULL);
}

rpc_server_inq_bindings( &bind_vector_p, &status );
ERRCHK ( status );

if (debug == 1) {
    printf("Registering endpoint ...\n");
    fflush (NULL);
}

rpc_ep_register(
    IF_HANDLE,
    bind_vector_p,
    NULL,
    "Message Box server, version 2.0",
    &status
);
ERRCHK ( status );

if (debug == 1) {
    printf("Exporting binding information ...\n");
    fflush (NULL);
}
rpc_ns_binding_export(
    rpc_c_ns_syntax_default,
    ENTRY_NAME,
    IF_HANDLE,
    bind_vector_p,
    NULL,
    &status
);
ERRCHK ( status );

/* ------------------------ */
/* baggage to handle ctrl-C */
/* ------------------------ */

#if !defined(_WINDOWS) && !defined(MVS)
sigemptyset ( &sigset );
sigaddset ( &sigset, SIGINT );
sigaddset ( &sigset, SIGTERM );
if ( pthread_signal_to_cancel_np ( &sigset, &this_thread ) != 0 ) {
    printf ( "pthread_signal_to_cancel_np failed\n"");
    exit (1);
}
#endif

/* ----------------------------------------- */
/* Import the mbox structure from file FNAME */
/* ----------------------------------------- */
mbox_import ( FNAME );

/* ---------------------------- */
/* Listen for service requests. */
/* ---------------------------- */
TRY {

    printf("Server %s listening.\n", ENTRY_NAME);
    fflush(NULL);
    rpc_server_listen ( MAX_CONC_CALLS_TOTAL, &status );
    ERRCHK ( status );
}

/* ---------------------------------------------------- */
/* ctrl-c or shutdown request from an authorised client */
/* ---------------------------------------------------- */

Figure 126 (Part 7 of 9). Security Sample Server Code (mboxs.c)
FINALLY {

    /* --------------------------------------- */
    /* Export the mbox structure to file FNAME */
    /* --------------------------------------- */

    mbox_export ( FNAME );

    /* ---------------------------------------------------- */
    /* Unexport the binding information from the namespace. */
    /* ---------------------------------------------------- */

    if (debug == 1) {
        printf("Unexporting binding information ...\n");
        fflush(NULL);
    }

    rpc_ns_binding_unexport ( 
        rpc_c_ns_syntax_default, 
        ENTRY_NAME, 
        IF_HANDLE, 
        NULL, 
        &status
    );
    ERRCHK ( status );

    /* ------------------------------------- */
    /* Unregister interface from RPC runtime */
    /* ------------------------------------- */

    if (debug == 1) {
        printf("Unregistering interface from runtime ...\n");
        fflush(NULL);
    }

    rpc_server_unregister_if ( IF_HANDLE, NULL, &status );
    ERRCHK( status );

    /* ----------------------------- */
    /* Unregister interface from EPV */
    /* ----------------------------- */

    if (debug == 1) {
        printf("Unregistering interface from EPV ...\n");
        fflush(NULL);
    }

    rpc_ep_unregister ( IF_HANDLE, bind_vector_p, NULL, &status );
    ERRCHK( status );

Figure 126 (Part 8 of 9). Security Sample Server Code (mboxs.c)
/* -------------------------------- */
/* remove thread exception handling */
/* -------------------------------- */

#ifdef IBMOS2
    pthread_dinst_exception_handler();
#endif

/* -------------------------------- */
/* exit */
/* -------------------------------- */

exit ( 0 );
}
ENDTRY;
}
B.2 Manager Module C Source

The following is the full C source code for the manager module of the messagebox application.

```c
/* */
/* */
/* Name: mboxm.c */
/* */
/* Description: Implements the handling procedures for the mbox data */
/* structure in the MessageBox sample application */
/* */
/***********************************************************/
/* */
/* included headers */
/* */
#include <string.h>
#include <stdio.h>
#include "mbox.h"
#include "mboxcom.h"
/* */
/* defined constants */
/* */
#define MAX_USERS 100
#define MBOX_IMPORT_FAILED -14
#define MBOX_EXPORT_FAILED -15
#define EOF_READ 0
#define STR_READ 1
#ifdef _WINDOWS
#define strdup _fstrdup
#define strcmp _fstrcmp
#endif
/* */
/* prototypes */
/* */
/* prototypes for routines imported from the security module */
char *get_principal ( handle_t );
int is_authorized ( handle_t );
```

Figure 127 (Part 1 of 11). Security Sample Server Manager Code (mboxm.c)
/* global variables */
/* ---------------------- */

/* entry is the structure for a single message in the message queue */
struct entry {
    char    *msg;
    char    *sdr;
    struct entry *nxt;
};
typedef struct entry entry_t;

/* mbox is the structure for a message box for a principal */
struct mbox {
    char    *principal;
    struct entry *first;
};
typedef struct mbox mbox_t;

/* allocate empty message boxes for MAX_USERS */
mbox_t mbox[MAX_USERS] = { NULL, NULL };  

/**********************************************************************/
/* */
/* Name: function get_mbox() */
/* */
/* Description: get_mbox returns a pointer to the message box of */
/* principal or a NULL if there is no message box for */
/* this principal. */
/* */
/* **********************************************************************/
mbox_t *get_mbox( const char *principal )
{
    /* local variables */
    mbox_t     *mb = mbox;

    /* find principal in chain */
    while ( mb -> principal ) {
        if ( strcmp ( principal, mb -> principal ) == 0 )
            return mb;
        mb++;
    }
    return NULL;
}

Figure 127 (Part 2 of 11). Security Sample Server Manager Code (mboxm.c)
entry_t *new_msg( mbox_t *mbox )
{
    entry_t *last;
    entry_t *new = (entry_t *)malloc( sizeof ( entry_t ) );
    new -> msg = NULL;
    new -> sdr = NULL;
    new -> nxt = NULL;
    if ( mbox -> first ) {
        last = mbox -> first;
        while ( last -> nxt )
            last = last -> nxt;
        last -> nxt = new;
    } else
        mbox -> first = new;

    return new;
}

idl_long_int mbox_new( handle_t bh, const char *principal )
{
mbox_t *mb = mbox;
unsigned32 status;

if ( ! is_authorized(bh) )
    return ( MBOX_NOT_AUTHORIZED );

if (strcmp(principal, "shutdown") == 0) {
    fprintf(stdout, "Requested to shutdown\n");
    rpc_mgmt_stop_server_listening(NULL, &status);
    ERRCHK(status);
    return ( MBOX_OK );
}

while ( mb -> principal ) {
    if ( strcmp (principal, mb -> principal) == 0 )
        return( MBOX_PRINCIPAL_EXIST );
    mb++;
}

if ( mb - mbox > MAX_USERS - 2 )
    return ( MBOX_ALL_USED );

mb -> principal = strdup( principal );
printf("Creating new Message Box for Principal %s.\n", principal);

Figure 127 (Part 4 of 11). Security Sample Server Manager Code (mboxm.c)
/* --------------------- */
/* initialize next entry */
/* --------------------- */

mb++;
mb -> principal = NULL;
mb -> first = NULL;

return ( MBOX_OK );
}

/*********************************************************************/
/* */
/* Name: function mbox_append() */
/* */
/* Description: mbox_append is advertised by the server to clients. */
/* It appends a new message to the existing message box */
/* of the given principal. */
/* Every authenticated user can call mbox_append. It */
/* tries to get the principal name from the RPC runtime. */
/* */
/*****************************************************************************/

idl_long_int mbox_append ( handle_t bh,
 const char *principal,
 const char *message )
{

/* ------------------------------- */
/* local variables */
/* ------------------------------- */

mbox_t *mb;
entry_t *new;

/* --------------------------------------- */
/* look up the mbox entry of the principal */
/* --------------------------------------- */

if ( ! (mb = get_mbox( principal )) )
    return ( MBOX_PRINCIPAL_NOT_EXIST );

/* --------------------------------------- */
/* append the new message entry to the message list */
/* --------------------------------------- */

new = new_msg( mb );

Figure 127 (Part 5 of 11). Security Sample Server Manager Code (mboxm.c)
/* -------------------------------------------------- */
/* assign the message and the sender principal to the */
/* message entry */
/* -------------------------------------------------- */

new -> msg = strdup( message );
new -> sdr = get_principal( bh );
printf("Message from %s for %s.\n", new->sdr, principal);

return ( MBOX_OK );
}

/**********************************************************************/
/* */
/* Name: function mbox_next() */
/* */
/* Description: mbox_next is advertised by the server to RPC clients. */
/* It returns in the message parameter the oldest unread */
/* message from caller’s mbox structure. */
/* mbox_next uses the get_principal call from the */
/* security module to select users message box. */
/* The returned message is discarded from the queue. */
/* */
/**********************************************************************/

idl_long_int mbox_next ( handle_t bh, char *message )
{
    /* ------------------------------- */
    /* local variables */
    /* ------------------------------- */

    mbox_t    *mb;
    char      *principal;

    /* ------------------------------- */
    /* get caller’s principal name */
    /* ------------------------------- */

    if ( (principal = get_principal(bh)) == NULL )
        return ( MBOX_NOT_AUTHORIZED );

    /* ------------------------------- */
    /* get corresponding mbox */
    /* ------------------------------- */

    if ( ! (mb = get_mbox( principal ) ) )
        return ( MBOX_PRINCIPAL_NOT_EXIST );

    // Further code...

Figure 127 (Part 6 of 11). Security Sample Server Manager Code (mboxm.c)
if ( mb -> first ) {
    /* copy the message into message */
    strcpy( message, mb -> first -> sdr );
    strcat( message, " ");
    strcat( message, mb -> first -> msg );
    printf("%s reads message from %s
", principal, mb->first->sdr);
    /* discard the entry */
    mb -> first = mb -> first -> nxt;
    return ( MBOX_OK );
}

return ( MBOX_NO_MSG );

int mbox_export( const char *fname )
{
    /* local variables */
    entry_t *me;
    FILE *export;
    mbox_t *mb = mbox;

    /* */
if ( (export = fopen(fname, "w")) == NULL ) {
    fprintf(stdout, "Cannot export messages to file %s\n", fname);
    return ( MBOX_EXPORT_FAILED );
}

printf("Exporting unread messages to file %s\n", fname);
while ( mb -> principal ) {
    fprintf(export,"%s:\n", mb -> principal);
    me = mb -> first;
    while ( me ) {
        fprintf(export, "%s: %s\n", me -> sdr, me -> msg);
        me = me -> nxt;
    }
    fputc(\'\n\', export);
    mb++;
}
fclose(export);
return ( MBOX_OK );

int readln (FILE *stream, char *buffer)
{
    /* Name: function readln() */
    /* Description: readln reads a line into buffer from file `stream' */
    /* until a newline character `\n' or the end-of-file */
    /* mark EOF is read. The newline character is replaced */
    /* by `\0'. */
    /* readln discards carriage returns `\r' == 0xd. */
    /* Return Values: */
    /* STR_READ = 1 if a line is read. */
    /* EOF_READ = 0 if EOF is encountered. */
    /* NOTE: The last line is returned together with EOF_READ. */
    /* */
    /* *********************************************************************/
    /* */
    /* Figure 127 (Part 8 of 11). Security Sample Server Manager Code (mboxm.c) */
    /* *********************************************************************/
int i;
char *c = buffer;

while ( 1 ) {
    switch ( i = fgetc(stream) ) {
    case EOF:  
        *c = EOF;  
        return EOF_READ;  
    case 'n':  
        *c = '0';  
        return STR_READ;  
    case 'r':  
        continue;  
    default:  
        *c++ = i;  
        if ( c - buffer > MAX_CHAR )  
            return STR_READ;
        
        continue;  
    }
}

/***************************************************************************/
/* */
/* Name: function mbox_import() */
/* */
/* Description: mbox_import imports the mbox structure from the file */
/* written by mbox_export. */
/* */
/***************************************************************************/
int mbox_import ( const char *fname )
{
    /* ------------------------------- */
    /* local variables */
    /* ------------------------------- */

    FILE *import;
    char buffer[MAX_CHAR];
    char *m;
    entry_t *me;
    mbox_t *mb = mbox;

    
Figure 127 (Part 9 of 11). Security Sample Server Manager Code (mboxm.c)
/* open local file fname for reading */
/* */

if ( (import = fopen(fname,"r")) == NULL ) {
    fprintf(stdout,"Cannot import messages from file %s\n",fname);
    return( MBOX_IMPORT_FAILED );
}

printf("Importing old messages from file %s\n",fname);

/* while file is not empty read lines */
/* NOTE: The last line is ignored because it is always */
/* empty by definition of the file format. */
while ( readln(import, buffer) ) {
    if ( *buffer == \0 ) {
        /* line read is empty -> new principal */
        if ( ++mb - mbox > MAX_USERS - 2 )
            return ( MBOX_ALL_USED );

        mb -> principal = NULL;
        mb -> first = NULL;
        continue;
    }

    /* split line after ':': */
    if ( m = strchr(buffer,':') )
        "m++ = '\0';

/* if principal is not set a new message box is read */
if ( mb -> principal ) {
    /* append message to message list */
    me = new_msg( mb );
    me -> sdr = strdup(buffer);
    me -> msg = strdup(m);
} else

/* ----------------------------------- */
/* line read is empty -> new principal */
/* ----------------------------------- */

Figure 127 (Part 10 of 11). Security Sample Server Manager Code (mboxm.c)
/* --------------------------------------- */
/* set principal to assign new message box */
/* --------------------------------------- */

mb -> principal = strdup(buffer);

fclose( import );
return( MBOX_OK );
B.3 Security Module C Source

The following is the full C source code for the security module of the messagebox application.

```c
#include <string.h>
#include <dce/rpc.h>
#ifdef _WINDOWS
#include <dce/daclf.h>
#define READ
#define WRITE
#else
#include <dce/daclif.h>
#endif
#include "mboxcom.h"
```

Figure 128 (Part 1 of 5). Security Sample Security Code (mboxsec.c)
/* define variables */

/* ACL access checking values */
#define READ 0x00000001
#define WRITE 0x00000002
#define EXECUTE 0x00000004
#define CONTROL 0x00000008
#define INSERT 0x00000010
#define DELETE 0x00000020
#define TEST 0x00000040

#ifdef _WINDOWS
# define strdup _fstrdup
#endif

/**********************************************************************/
/* Name: function get_principal() */
/* Description: Returns a pointer to a string with principal name associated with clients binding handle. */
/* Client must set up name based authorization to send his principal name. */
/**********************************************************************/

char *get_principal ( handle_t bh )
{
    /* local variables */
    rpc_authz_handle_t Credentials;
    unsigned32 authz_svc,
               status;
    char *pname;

    if (debug == 1) {
        printf("Getting clients auth info ...
        ");
        fflush(NULL);
    }

    Figure 128 (Part 2 of 5). Security Sample Security Code (mboxsec.c)
rpc_binding_inq_auth_client(
  bh,     /* binding handle */
  &Credentials,  /* returned privileges */
  NULL,     /* we provide no principal name */
  NULL,     /* no protection level returned */
  NULL,     /* no authn_svc returned */
  &authz_svc,  /* Credential contents indicator */
  &status
);
ERRCHK( status );

/* --------------------------------- */
/* check the contents of credentials */
/* --------------------------------- */

if ( authz_svc != rpc_c_authz_name )
  return NULL;

/* ------------------- */
/* cast type to string */
/* ------------------- */

pname = strdup((char *)Credentials);

/* -------------------------------------- */
/* strip off leading cell name and return */
/* -------------------------------------- */

return (strrchr(pname, '/') + 1);
}

********************************************************************************
/*
/* Name: function is_authorized() */
/* Description: Returns true ( != 0 ) if the principal associated */
/* with client binding handle has control rights granted by the ACL on server name space entry. */
/* Client must set up DCE authorization to send his PAC. */
/*
********************************************************************************

int is_authorized ( handle_t bh )
{

/* -------------------------------- */
/* local variables */
/* -------------------------------- */

Figure 128 (Part 3 of 5). Security Sample Security Code (mboxsec.c)
rpc_authz_handle_t Credentials;
unsigned32 authz_svc,
status;
sec_acl_handle_t acl;
boolean32 accessOK;
unsigned32 num_rtnd,
num_avail;
uuid_t mgrs[10];

/* --------------------------------------------- */
/* get clients auth info, client should send PAC */
/* --------------------------------------------- */

if (debug == 1) {
    printf("Getting clients auth info ...\n");
    fflush(NULL);
}

rpc_binding_inq_auth_client(
    bh, /* binding handle */
    &Credentials, /* returned privileges */
    NULL, /* we provide no principal name */
    NULL, /* no protection level returned */
    NULL, /* no authn_svc returned */
    &authz_svc, /* Credential contents indicator */
    &status
);
ERRCHK( status )

/* -------------------------------- */
/* check the contents of credentials */
/* -------------------------------- */

if ( authz_svc != rpc_c_authz_dce )
    return 0;

/* -------------- */
/* get ACL handle */
/* -------------- */

if (debug == 1) {
    printf("Getting ACL handle ...\n");
    fflush(NULL);
}

sec_acl_bind(
    ENTRY_NAME, /* entry name for acl checking */
    1, /* get handle to entry in namespace */
    &acl, /* handle to acl */
    &status
);
ERRCHK( status );
if (debug == 1) {
    printf("Getting ACL manager ...
    
fflush(NULL);
}

sec_acl_get_manager_types(
    acl,  /* handle to the ACL */
    sec_acl_type_object,  /* acl pts to a CDS object */
    NUM_ELEMS(mgrs),  /* number of items in array */
    &num_rtnd,  /* number of items returned */
    &num_avail,  /* number of items existing */
    mgrs,  /* array base */
    &status
);
ERRCHK( status );

/* deny access if no ACL manager available */
if (num_rtnd == 0)
    return 0;

/* prove ACL granting control right */
/* ---------------- */

if (debug == 1) {
    printf("Testing CONTROL access ...
    
fflush(NULL);
}

accessOK = sec_acl_test_access_on_behalf(
    acl,  /* handle to acl */
    &mgrs[0],  /* ACL mgr for object */
    Credentials,  /* this is PAC from client */
    CONTROL,  /* permissions to check for */
    &status
);
ERRCHK( status );

/* return */
/* ---------------- */

return accessOK;

Figure 128 (Part 5 of 5). Security Sample Security Code (mboxsec.c)
B.4 Client Module C Source

The following is the full C source code for the client module of the messagebox application.

 /**********************************************************************/
 /* */
 /* Module: mboxc.c */
 /* */
 /* Description: Implements the client for the MessageBox application. */
 /* */
 /**********************************************************************/

 /* ------------------------ */
 /* included headers */
 /* ------------------------ */

 #ifdef _WINDOWS
 #include <dce/dcewin.h>
 #endif
 #include <string.h>
 #include "mbox.h"
 #include "mboxcom.h"

Figure 129 (Part 1 of 7). Security Sample Client Code (mboxc.c)
/* defined constants */
/* prototypes */
/* global variables */
/* Main */

int main (int argc, char *argv[])
{
/* local variables */
char     msg[MAX_CHAR] = "";
int      action = MBOX_READ;
int      rc;
handle_t bh;
rpc_ns_handle_t ns_handle;
error_status_t status;

/* local variables */
/* Main */
/* global variables */
/* prototypes */
/* defined constants */

#define NO_ERROR 10
#define NO_OPTION -11
#define NO_PRINCIPAL -12
#ifdef _WINDOWS
#define strdup _fstrdup
#endif
#ifdef IBMOS2
#define USAGE "usage: %s [/c] [principal] [message]"
#define OPTCHAR '/'
#else
#define USAGE "usage: %s [-c] [principal] [message]"
#define OPTCHAR '-'
#endif
#define IF_HANDLE MessageBox_v2_0_c_ifspec

void error( int );

define constants
 prototypes
 global variables
 Main

Figure 129 (Part 2 of 7). Security Sample Client Code (mboxc.c)
/* ---------------------- */
/* get application name */
/* ---------------------- */

prgname = strdup(*argv++);

/* ---------------------------------- */
/* parse the command line for options */
/* ---------------------------------- */

debug = 0;
while ( --argc ) {
    if ( **argv != OPTCHAR )
        break;
    while ( *++(*argv) ) {
        switch ( **argv ) {
            case «c»:
                action = MBOX_CREATE;
                break;
            case «d»:
                debug = 1;
                break;
            default:
                error(NO_OPTION);
                break;
        }
        argv++;
    }
}

/* ---------------------------------------------------------------- */
/* if there are args, assume the first none option is the principal */
/* ---------------------------------------------------------------- */

if ( argc ) {
    principal = *argv++;
    if ( action == MBOX_READ )
        action = MBOX_APPEND;

    /* ------------------------------------------------ */
    /* concat the rest of the command line to a message */
    /* ------------------------------------------------ */

    while ( --argc ) {
        strcat(msg,*argv++);
        strcat(msg,

/* ----------------------------- */
/* check for the principal argument */
/* ----------------------------- */

if ( ( action == MBOX_CREATE ) && ( principal == "" ) )
    error( NO_PRINCIPAL );

/* ----------------------------- */
/* set up for reading binding information */
/* ----------------------------- */

if (debug == 1) {
    printf("beginning binding import ...
");
    fflush(NULL);
}

rpc_ns_binding_import_begin(
    rpc_c_ns_syntax_default,
    ENTRY_NAME,
    IF_HANDLE,
    NULL,
    &ns_handle,
    &status
);
ERRCHK( status );

/* ----------------------------- */
/* import first binding information from name space */
/* ----------------------------- */

if (debug == 1) {
    printf("importing binding ...
");
    fflush(NULL);
}

rpc_ns_binding_import_next(
    ns_handle,
    &bh,
    &status
);
ERRCHK( status );

/* ----------------------------- */
/* stop reading binding information */
/* ----------------------------- */

if (debug == 1) {
    printf("importing done ...
");
    fflush(NULL);
}
rpc_ns_binding_import_done(
    &ns_handle,
    &status
);
ERRCHK( status );

/* ---------------------------------------------------------- */
/* determine which authorization info to send based on action */
/* ---------------------------------------------------------- */

if ( action == MBOX_CREATE ) {
    if (debug == 1) {
        printf("setting auth info to PAC ...
");
        fflush(NULL);
    }
    rpc_binding_set_auth_info(
        bh,
        PRINCIPAL_NAME,
        rpc_c_protect_level_pkt_integ, /* send checksum */
        rpc_c_authn_default,
        NULL, /* use login context */
        rpc_c_authz_dce, /* send PAC */
        &status
    );
}
else {
    if (debug == 1) {
        printf("setting auth info to name ...
");
        fflush(NULL);
    }
    rpc_binding_set_auth_info(
        bh,
        PRINCIPAL_NAME,
        rpc_c_protect_level_pkt_integ, /* send checksum */
        rpc_c_authn_default,
        NULL, /* use login context */
        rpc_c_authz_name, /* send name, not PAC */
        &status
    );
}
ERRCHK( status );

Figure 129 (Part 5 of 7). Security Sample Client Code (mboxc.c)
switch ( action ) {
  case MBOX_CREATE:
    if ( (rc = mbox_new(bh,principal)) != MBOX_OK )
      error( rc );
    break;
  case MBOX_APPEND:
    if ( (rc = mbox_append(bh,principal,msg)) != MBOX_OK )
      error( rc );
    break;
  case MBOX_READ:
    switch ( rc = mbox_next(bh,msg) ) {
      case MBOX_OK:
        printf("%s\n",msg);
        break;
      case MBOX_NO_MSG:
        printf("%s: no more messages\n",prgname);
        break;
      default:
        error( rc );
        break;
    }
  exit ( NO_ERROR );
}

void error (int msg)
{
  switch ( msg ) {
    /* Name: function error() */
    /* Description: error() implements the error printing routine. */
    /* All error output is send to stderr. After the error text is printed error() exits the application with an error return value. */

Figure 129 (Part 6 of 7). Security Sample Client Code (mboxc.c)
case MBOX_PRINCIPAL_EXIST:
    fprintf(stderr, "mbox for principal %s exists already\n", 
        prgname, principal);
    break;

case MBOX_PRINCIPAL_NOT_EXIST:
    fprintf(stderr, "principal %s has no message box\n", 
        prgname, principal);
    break;

case MBOX_NOT_AUTHORIZED:
    fprintf(stderr, "you are not authorized\n", prgname);
    break;

case MBOX_ALL_USED:
    fprintf(stderr, "no more message boxes available\n", prgname);
    break;

case NO_PRINCIPAL:
    fprintf(stderr, "missing principal name\n", prgname);
    break;

case NO_OPTION:
    fprintf(stderr, "unknown option\n", prgname);
    break;

default:
    fprintf(stderr, "something is wrong\n", prgname);

}

fprintf(stderr, USAGE, prgname);
exit( msg );

Figure 129 (Part 7 of 7). Security Sample Client Code (mboxc.c)
B.5 Common Header File

The following is the C source code for the common header file of the messagebox application.

/*
/* Name: header file mboxcom.h
/*
/* Description: Common header file for the Message Box example.
/* Defines the error check pseudo function which is
/* called to check the return status of call to the
/* RPC runtime.
/* Defines also the principal used by the server to
/* authenticate, the name for the entry in the CDS name
/* space and the return values of the remote procedures.
/*
/****************************************************************************/

#ifndef _MBOX_COMMON
#define _MBOX_COMMON

boolean32 debug; /* enable/disable debugging info */

#ifndef _H_STDIO
#include <stdio.h>
#endif

#ifdef _WINDOWS
#include <dce/base.h>
#include <dce/dceerror.h>
#endif

#include <stdio.h>
#include <string.h>
#if defined(MVS)
char *strdup(const char *str);
#endif

#define ERRCHK(x) \
if ( (x) != error_status_ok ) { \
    fprintf(stdout, "%s: %d\n", __FILE__, __LINE__ - 1); \
    dce_error_inq_text(x, dce_error_string, &error_inq_st); \
}

Figure 130 (Part 1 of 2). Security Sample Header File (mboxcom.h)
fprintf(stdout, \
  "cause: %d(x'X') - %s\n", \
  x, x, dce_error_string); \
} \
else { \
  if (debug == 1) { \
    fprintf(stdout, "OK: %s: %d\n", __FILE__, __LINE__ - 1); \
    fflush(NULL); \
  } \
} \

cchar dce_error_string\256"; \
error_status_t error_inq_st; \
#define NUM_ELEMS(a) (sizeof(a)/sizeof(a\0)) \
#define ENTRY_NAME \
"/./servers/MessageBox" \
#define PRINCIPAL_NAME \
"MessageBox" \
/* Return values of remote procedures */ \
#define MBOX_OK 0 \
#define MBOX_NO_MSG 1 \
#define MBOX_READ 2 \
#define MBOX_APPEND 3 \
#define MBOX_CREATE 4 \
#define MBOX_NOT_AUTHORIZED -1 \
#define MBOX_PRINCIPAL_EXIST -2 \
#define MBOX_PRINCIPAL_NOT_EXIST -3 \
#define MBOX_ALL_USED -4 \
#endif \
=endif \

Figure 130 (Part 2 of 2). Security Sample Header File (mboxcom.h)
The following is the C source code for the non-SAA function, `strdup`, that is not part of the AD/Cycle C/370 R2 library. It was used in the code that was ported to MVS from AIX and OS/2 and it is provided to minimize changes to the original code.

```c
char *strdup(const char *str)
{
    char *string1;
    string1 = (char *) malloc(strlen(str)+1);
    strcpy(string1, str);
    return(string1);
}
```

Figure 131. Library File (mboxlib.c)
B.7 Message REXX exec

The following is the REXX source code for the “message” of the messagebox application.

/* REXX */
trace o
parse arg p1
"call dce(mboxc) ’posix(on)/”p1”’"

Figure 132. Message REXX exec
B.8 Batch JCL to Run MessageBox Server

The following is the JCL to run the MessageBox application.

```
//ERICFINS JOB (999,POK),’ERICFIN’,NOTIFY=ERICFIN,
//   CLASS=A,MSGCLASS=H,
//   MSGLEVEL=(1,1),REGION=OM
//********************************************************************
//* JCL TO RUN THE MBOX SERVER                                      *
//********************************************************************
//RUN EXEC PGM=MBOXS,PARM=’POSIX(ON)/’,REGION=OM
//STEPLIB DD DSN=ERICFIN.DCE.LOAD,DISP=SHR
//      DD DSN=CEE.V1R3M0.SCEERUN,DISP=SHR
//      DD DSN=DCE.SEUVEXEC,DISP=SHR
//SYSIN DD *
//SYSERR DD SYSOUT=*  
//SYSPRINT DD SYSOUT=*  
```

*Figure 133. MessageBox Server JCL*
Appendix C. MATHX Source (Well-Known Endpoint)

Sample source for MATHX (well known endpoint) and makefile.

The following source listings, makefile, and printf output, are generated using the basic MATHX example, but run in the OpenEdition, HFS environment. The intention is to provide a sample Makefile so that you can more easily maintain your changes in this environment. We have included the source for MATHX because this example uses partial binding and listens on a well-known endpoint.

C.1 MATHX Client with Well-Known Endpoint

```c
/******************************************************************************/
/* Module : MATHXC */
/* Purpose : Client module for server MATHXS. Gets the values to be */
/* added from the command line and sends them to the remote add */
/* procedure together with the number of values passed and a */
/* pointer to the variable that will contain the result. */
/* Uses explicit binding. */
/******************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dce/rpc.h>
#include [errchk.h]
#include [mathx.h]

#define ENTR[_NAME ]/.:/servers/pok/mathxs] /* Server entry name. */

int main ( int argc, char *argv[] )
{
  long int i, numv, sum;
  value_array_t va;
  rpc_binding_handle_t bh;
  unsigned32 status;
  rpc_ns_handle_t imp_ctxt;
  unsigned_char_t *string_binding = NULL;

  /**************************************************************************/
  /* Check if the user passed the minimum number of parameters. */
  /**************************************************************************/
  if ( argc < 3 )
    {
      printf ( ]Usage : %s <values to be added>
    , argv[0]);
      fflush(NULL);
      exit ( 1 );
    }
```

*Figure 134 (Part 1 of 3). Sample Source for Well-Known Endpoint*
```c
/* Compose */
rpc_string_binding_compose(
    [],
    &ncadg_ip_udp],
    [9.12.13.69],
    [2000],
    [],
    &string_binding,
    &status);
ERRCHK ( status );

/* Generate a binding handle from the string binding in the command line. */
printf(Generate a binding handle
]
];
fflush(NULL);

rpc_binding_from_string_binding (
    string_binding,
    &bh,
    &status );
ERRCHK ( status );

printf(string_binding = %s
], string_binding);
fflush(NULL);

/* Get values from the command line and load them into array va. */
printf ( ]Get values and load into the array...\n] );
fflush(NULL);

for ( i = 0; i < MAX_VALUES && i < argc - 1; i++ )
    va[ i ] = atoi ( argv[ i + 1 ] );
numv = i;
```

*Figure 134 (Part 2 of 3). Sample Source for Well-Known Endpoint*
/***************************************************************************/
/* Call add passing handle, values, number of values and pointer to result.*/
***************************************************************************/

printf ( ]Call add routine, pass handle and values, and point to result...
fflush(NULL);

add ( bh, va, ( long )numv, Σ );

printf ( ]The sum is \%d\n], sum );
fflush(NULL);
}

Figure 134 (Part 3 of 3). Sample Source for Well-Known Endpoint
C.2 MATHX Server with Well-Known Endpoint

```
#include <stdio.h>
#include <stdlib.h>
#include <dce/rpc.h>
#include <pthread.h>
#include <dce/dce_cf.h>
#include <dce/exc_handling.h>
#include [errchk.h]
#include [mathx.h]

#define MAX_CONC_CALLS_PROTSEQ 1 /* Max concurrent calls per protocol. */
#define MAX_CONC_CALLS_TOTAL 2 /* Max concurrent calls total. */
#define ENTR[_NAME _]/.:/servers/pok/mathxs] /* Server entry name. */

int main ( int argc, char *argv[]) {
    rpc_binding_vector_t *bv_p;
    unsigned32 status;
    int i;
    char *string_binding;
    char SERVER = ]MATHXS[;  

    printf([Use supported protocol sequences...\n]);
    fflush(NULL);

    rpc_server_use_protseq_ep( 
        [ncadg_ip_udp],
        MAX_CONC_CALLS_PROTSEQ,
        ]2000],
        &status );
    ERRCHK ( status );
```

Figure 135 (Part 1 of 3). Sample Source for MATHX Server with Well-Known Endpoint
/* Register interface/epv associations with RPC runtime. */

printf("Registering server interface with RPC runtime...\n");
fflush(NULL);

rpc_server_register_if (mathx_v1_0_s_ifspec,
    NULL,
    NULL,
    &status);
ERRCHK ( status );

/* Get the binding handle vector from RPC runtime. */

printf("Get binding handle from RPC runtime...\n");
fflush(NULL);

rpc_server_inq_bindings (&bv_p,
    &status);
ERRCHK ( status );

/* String Binding to String Binding */

printf("( %s )Bindings:\n",SERVER);
fflush(stdout);
for (i=0; i< bv_p->count; i++)
{
    rpc_binding_to_string_binding( bv_p->binding_h[i],
        &string_binding,&status);
    ERRCHK ( status );
    printf("%s\n", (char *)string_binding);
    fflush(stdout);
}

/* Free String */

rpc_string_free( &string_binding,&status);
ERRCHK ( status );

Figure 135 (Part 2 of 3). Sample Source for MATHX Server with Well-Known Endpoint
/** Listen for service requests. */

```
TR[
{
    printf ( ]Server %s listening...\n], ENTR[.NAME );
    fflush(NULL);

    rpc_server_listen (
        MAX_CONC_CALLS_TOTAL,
        &status );
    ERRCHK ( status );
}
CATCH_ALL
{
    printf(] CATCH ALL \n]);
}
ENDTR[;
}
```

*Figure 135 (Part 3 of 3). Sample Source for MATHX Server with Well-Known Endpoint*
C.3 Well-Known Endpoint OUTPUT (printf statements)

************************************************************************
** Started the mathx_server in background (mathx_server &)
************************************************************************

# mathx_server &
[1] 1114130
# Use supported protocol sequences...
Registering server interface with RPC runtime...
Get binding handle from RPC runtime...
(MATHXS )Bindings:
Server /./servers/pok/mathxs listening...

************************************************************************
** Started the mathx_Client in the foreground
************************************************************************

# mathx_client 100 100
Generate a binding handle
Get values and load into the array...
Call add routine, pass handle and values, and point to result...
Add has been called with numv = 2
Value_a[ 0 ] = 100
Value_a[ 1 ] = 100
Zero the accumulator
Add the values and place results in total...
The sum is 200

Figure 136. OUTPUT Sstatements for Well-Known Eexample
C.4 Makefile for MATHX OE Example

IF = mathx
IDL = /bin/idl
IDL_FLAGS = -I/share/include -v -version -keep all -cc_opt ]$(CFLAGS)
CFLAGS = -DMVS -D_DCE_THREADS -D_OPEN_SYS -I/share/include
MYPATH = /u/bmount/dce
MYCFLAGS = -V -W ]0,NOSHOWINC,NOCHECKOUT,NOLIST,NOOFFSET] -v
LIBS = -l dce
FROMIDL = $(IF).h $(IF)_cstub.c $(IF)_sstub.c
COBJ = mathxc.o $(IF)_cstub.o
SOBJ = mathxs.o $(IF)_sstub.o mathxm.o
all: $(IF)_client $(IF)_server
$(IF)_client: $(COBJ)
  c89 -o $(IF)_client $(COBJ) $(LIBS)
$(IF)_server: $(SOBJ)
  c89 -o $(IF)_server $(SOBJ) $(LIBS)
$(IF)_cstub.o : $(IF)_cstub.c $(IF).h
  c89 -c $(CFLAGS) $(MYCFLAGS) -I$(MYPATH) $*.c
$(IF)_sstub.o : $(IF)_sstub.c $(IF).h
  c89 -c $(CFLAGS) $(MYCFLAGS) -I$(MYPATH) $*.c
mathxc.o : mathxc.c $(IF).h
  c89 -c $(CFLAGS) $(MYCFLAGS) -I$(MYPATH) $*.c
mathxs.o : mathxs.c $(IF).h
  c89 -c $(CFLAGS) $(MYCFLAGS) -I$(MYPATH) $*.c
mathxm.o : mathxm.c $(IF).h
  c89 -c $(CFLAGS) $(MYCFLAGS) -I$(MYPATH) $*.c
$(FROMIDL): $(IDL) $(IF).idl $(IF).acf
  $(IDL) $(IF).idl $(IDL_FLAGS)
clean:
  rm -f $(FROMIDL) *.o

Figure 137. Sample Makefile for OE Example.
D.1 DB2

This section contains the source for the DB2 programs.

D.1.1 IDL

```
[uuid(19F22E70-7B56-1E4F-AE33-C9C2D4FF006A),
version(1.0),
pointer_default(ptr)
]
interface DB2SQL
{
  typedef struct
  {
    char C_FIRST [ 17 ];
    char C_MIDDLE [ 3 ];
    char C_LAST [ 17 ];
    char C_STREET_1 [ 21 ];
    char C_STREET_2 [ 21 ];
    char C_CITY [ 21 ];
    char C_STATE [ 3 ];
    char C_ZIP [ 10 ];
    char C_PHONE [ 17 ];
  } CUSTOMER;

  typedef struct
  {
    long int size;
    [ size_is(size) ] CUSTOMER *pCustomers [ * ];
  } CUSLST;

  void get_customersC
  ( [ in,string ] char c_last [ 17 ],
    [ out ] long int *sql_code,
    [ out,string ] char result_msg [ 101 ],
    [ out ] CUSLST **ppCusLst
  );
}
```

*Figure 138. DB2 IDL*
D.1.2 Header

/* Generated by IDL compiler version DCE 1.0.0-1 */
#ifndef DB2SQL_v1_0_included
#define DB2SQL_v1_0_included
#include <dce/idlbase.h>
#include <dce/dcerpcmsg.h>
#include <dce/rpc.h>

#ifdef __cplusplus
extern "C"
{
#endif
#include "dce/nbase.h"
typedef struct {
    idl_char C_FIRST [ 17 ] ;
    idl_char C_MIDDLE [ 3 ] ;
    idl_char C_LAST [ 17 ] ;
    idl_char C_STREET_1 [ 21 ] ;
    idl_char C_STREET_2 [ 21 ] ;
    idl_char C_CITY [ 21 ] ;
    idl_char C_STATE [ 3 ] ;
    idl_char C_ZIP [ 10 ] ;
    idl_char C_PHONE [ 17 ] ;
} CUSTOMER;
typedef struct {
    idl_long_int size;
    CUSTOMER *pCustomers [ 1 ] ;
} CUSLST;
extern void get_customersC(
#ifdef IDL_PROTOTYPES
    /* [ in ] */ handle_t IDL_handle,
    /* [ in ] */ idl_char c_last [ 17 ] ,
    /* [ out ] */ idl_long_int *sql_code,
    /* [ out ] */ idl_char result_msg [ 101 ] ,
    /* [ out ] */ CUSLST **ppCusLst,
    /* [ out ] */ error_status_t *rpc_comm_status,
    /* [ out ] */ error_status_t *rpc_fault_status
#else
#endif
    }
);

Figure 139 (Part 1 of 2). DB2 Header
typedef struct DB2SQL_v1_0_epv_t {
  void (*get_customersC)(
    #ifdef IDL_PROTOTYPES
    /* [ in ] */ handle_t IDL_handle,
    /* [ in ] */ idl_char c_last [ 17 ] ,
    /* [ out ] */ idl_long_int *sql_code,
    /* [ out ] */ idl_char result_msg [ 101 ] ,
    /* [ out ] */ CUSLST **ppCusLst,
    /* [ out ] */ error_status_t *rpc_comm_status,
    /* [ out ] */ error_status_t *rpc_fault_status
    #endif
  );
} DB2SQL_v1_0_epv_t;

extern rpc_if_handle_t DB2SQL_v1_0_c_ifspec;
extern rpc_if_handle_t DB2SQL_v1_0_s_ifspec;

#ifdef __cplusplus
}
#endif
#endif

Figure 139 (Part 2 of 2). DB2 Header
D.1.3 Client Source Code

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "db2sql.h"
#define NOPAC
#define SIZE 4097
#define NO 0
#define YES 1
#define FALSE 0
#define TRUE 1
#define SEP1 '#'
#define SEP2 '$'
#define NUL '\0'
#define DCE_LOGIN "MYCLIENT"
#define PASSWD "VANAER"
#define SERVER_PRINCIPAL "MYSERVER"
#define SERVER_NAME "./servers/DB2SQLSV"
#define IFSPEC "DB2SQL_v1_0_c_ifspec"
#define IP "9.12.13.69"
#define PORT "1090"
#define PROT "ncadg_ip_udp"
/*--------------- D C E --------------------------------*/
#include <dce/rpc.h>
#include <dce/sec_login.h>
#include <dce/dce_error.h>
#include <pthread.h>
/*-----------------------------*/
/* external static variables used for DCE */
/*-----------------------------*/
/* int bind_ok; */
/*-----------------------------*/
/* pass following arguments */
/* 0) Binding Handle (dummy) */
/* 1) last name char16 in */
/* 2) pointer to sqlcode out */
/* 3) pointer to message out */
/* 4) pointer to ptr to conformant */
/* of pointers to db2rows */
/* 5) pointer to error commstatus */
/* 6) pointer to error faultstatus */
/*-----------------------------*/
```

Figure 140 (Part 1 of 3). DB2 Client
int main(int ac, char **av)
{
    short i;
    char lastname[17];
    CUSLST *pCusLst;
    char message[201];
    long sqlcode;
    rpc_binding_handle_t bh;
    rpc_if_handle_t ifhand = IFSPEC;
    error_status_t err1;
    error_status_t err2;
    error_status_t st;
    char CLIENT[9];
    char prot[ ] = PROT;
    char ip[ ] = IP;
    char port[ ] = PORT;
    char *dceopt = "EP";
    strcpy(CLIENT, "DB2SQLCL");
    printf("===> Starting CLIENT %s 
", CLIENT);
    fflush(stdout);
    /* bind_ok = FALSE; */
    dce_serv(dceopt,
            CLIENT, prot, ip, port,
            ifhand,
            &bh,
            DCE_LOGIN,
            SERVER_PRINCIPAL,
            SERVER_NAME,
            PASSWD);
    /*-----------------------------------------------*/
    strcpy(lastname, "H%");
    pCusLst = NULL;
    sqlcode = 0;
    /*-----------------------------------------------*/
    get_customersC
    (bh, lastname, &sqlcode, message, &pCusLst, &err1, &err2);
    /*-----------------------------------------------*/
    if (dce_errt(err1, err2) != 0) {
        return(-1);
    } else {
        if (sqlcode >= 0) {
            /*-----------------------------------------------*/
            printf("sqlcode = %d Message = %s 
", sqlcode, message);
            printf("&pCusLst = %08xl\n", (long int) &pCusLst);
            printf("pCusLst = %08xl\n", (long int) pCusLst);
            printf("pCusLst->size = %d\n",
                   pCusLst->size);
        }
    }
}

Figure 140 (Part 2 of 3). DB2 Client
for (i=0; i < pCusLst->size; i++)
{
    printf("\n*** New Entry ***\n");
    printf("First: %s\n", pCusLst->pCustomers [ i ] ->C_FIRST);
    printf("Middle: %s\n", pCusLst->pCustomers [ i ] ->C_MIDDLE);
    printf("Last: %s\n", pCusLst->pCustomers [ i ] ->C_LAST);
    printf("Street: %s\n", pCusLst->pCustomers [ i ] ->C_STREET_1);
    printf("\n", pCusLst->pCustomers [ i ] ->C_STREET_2);
    printf("City: %s\n", pCusLst->pCustomers [ i ] ->C_CITY);
    printf("State: %s\n", pCusLst->pCustomers [ i ] ->C_STATE);
    printf("Phone: %s\n", pCusLst->pCustomers [ i ] ->C_PHONE);
}

/*------------------------------------------------------*/

/*-----------------------------------------------*/
} else {
    printf("sqlcode = %d Message = %s \n",sqlcode,message);
    return;
}
}

/*------------------------------------------------------*/

Figure 140 (Part 3 of 3). DB2 Client
D.1.4 Server Source Code

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define NOIMS
#define EP
#define NORPCD
#define NOSECCDS
#define NUL \0 /* binary X00 */
#define IFSPEC DB2SQL_v1_0_s_ifspec
#define IFEPVT DB2SQL_v1_0_epv_t
#define IFMGR DB2SQL_V1_0_manager_epv
#ifdef SECCDS
#include <dce/sec_login.h>
#endif
#include <pthread.h>
#include <dce/rpc.h>
#include <dce/dce_cf.h>
#include <dce/exc_handling.h>
#include "db2sql.h"
char DB2SSID[5];
int main(int ac, char *av[])
{
    rpc_binding_vector_p_t bvec;
    error_status_t st, est;
    char *string_binding;
    char err_str[1024];
    #ifdef SECCDS
    char dce_login[9];
    char password[9];
    char hostname[9];
    sec_login_handle_t lcontext;
    sec_login_auth_src_t auth_src;
    sec_passwd_rec_t pwrec;
    boolean32 reset_passwd;
    #endif
    #ifdef IMS
    char request[5];
    char imsuser[9];
    #endif
    int numthrds, numqueue;
    int rc, it, i, ims;
    char SERVER[9];
    strcpy(SERVER, DB2SQLSV);
    printf("==>
```
/*----------------------------------------------*/
/* 1 parameter is passed to the ServerCode */
/* DB2 Subsystem ID 4char */
/*----------------------------------------------*/

if (ac > 1) {
    /* put DB2SSID in External var for access from managers */
    strncpy(DB2SSID, av[1], 4);
    DB2SSID[4] = NUL;
    printf("Managers %s can connect to DB2 %s\n", SERVER, DB2SSID);
} else {
    printf("Managers %s cannot connect to DB2\n", SERVER);
    return;
}

/* ------------------------------------------------*/
/* Setting the protocol to be used in this program */
/* ------------------------------------------------*/
numthrds = 2;
numqueue = 2;

#ifdef EP
    /* register with DCE/RUNTIME protocol + EP */
    rpc_server_use_protseq_ep("ncadg_ip_udp", numqueue, 1090, &st);
#else
    /* register with DCE/RUNTIME protocol */
    rpc_server_use_protseq("ncadg_ip_udp", numqueue, &st);
#endif
if (st != error_status_ok) {
    dce_error_inq_text(st, err_str, &est);
    printf("( %s )--Cannot use protocol sequence - %s\n", SERVER, err_str);
} else {
    printf("( %s )Use protocol sequence OK - \n", SERVER);
}
fflush(stdout);

/* ------------------------------------*/
/* Register interface with RPC runtime */
/* ------------------------------------*/
rpc_server_register_if(DB2SQL_v1_0_s_ifspec, NULL, NULL, &st);
/*(rpc_mgr_epv_t) &IFMGR ,&st);*/
if (st != error_status_ok) {
    dce_error_inq_text(st, err_str, &est);
    printf("( %s )--Cannot register Interface - %s\n", SERVER, err_str);
} else {
    printf("( %s )Register interface OK - \n", SERVER);
}
fflush(stdout);

Figure 141 (Part 2 of 6). DB2 Server
/* Get binding handles from the runtime */
rpc_server_inq_bindings(&bvec, &st);

if (st != error_status_ok) {
    dce_error_inq_text(st, err_str, &est);
    printf("( %s -->Cannot inq bindings - %s\n", SERVER, err_str);
    fflush(stdout);
}

/* Display the bindings on the console */
printf("( %s )Bindings:\n", SERVER);
fflush(stdout);
for (i=0; i< bvec->count; i++)
{
    rpc_binding_to_string_binding(bvec->binding_h [ i ],
        &string_binding,&st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, err_str, &est);
        printf("( %s -->Cannot convert bindings - %s\n", SERVER, err_str);
        fflush(stdout);
    }
    printf("%s
", (char *)string_binding);
    fflush(stdout);
    rpc_string_free(&string_binding,&st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, err_str, &est);
        printf("( %s -->Cannot free mem - %s\n", SERVER, err_str);
        fflush(stdout);
    }
}
#endif
/* register dynamically EP with endpoint Mapper */
rpc_ep_register
#endif SECCDS
(DB2SQL_v1_0_s_ifspec, bvec, NULL,"SEC-CDS",&st);
#else
(DB2SQL_v1_0_s_ifspec, bvec, NULL,NULL , &st);
#endif
if (st != error_status_ok) {
    dce_error_inq_text(st, err_str, &est);
    printf("( %s -->Cannot register EP - %s\n", SERVER, err_str);
} else {
    printf("( %s )Register EP OK - \n\n",SERVER);
}
fflush(stdout);
#endif

Figure 141 (Part 3 of 6). DB2 Server
/* Establish Identity with security server */
/* call goes to registry service (on Security server */
/* returns TGT (for priviledge services) and ConvKey */

printf(( %s )Establishing Identity with security server...
       \n",SERVER);
fflush(stdout);
sec_login_setup_identity(dce_login,sec_login_no_flags,&lcontext,&st);
    if (st != error_status_ok)
    {
        dce_error_inq_text(st,err_str,&est);
        printf(( %s )--->Cannot setup login-identity %s\n",SERVER,err_str);
        fflush(stdout);
        exit(1);
    }
    pwrec.key.tagged_union.plain = "VANAER";
pwrec.key.key_type = sec_passwd_plain;
pwrec.pepper = NULL;
pwrec.version_number = sec_passwd_c_version_none;

/* Validation of Identity */
/* call goes to LOCAL security runtime */
/* runtime decrypts envelope with PASSWORD */

printf(( %s )Validation of identity..... \n",SERVER);
fflush(stdout);
sec_login_validate_identity(lcontext, &pwrec, &reset_passwd, &auth_src,&st);
    if (st != error_status_ok)
    {
        dce_error_inq_text(st,err_str,&est);
        printf(( %s )--->Cannot validate login-identity %s\n",SERVER,err_str);
        fflush(stdout);
        exit(1);
    }

Figure 141 (Part 4 of 6). DB2 Server
/*--------------------------*/
/* Set login context */
/*- sets network credentials (in context) */
/*- this context will be assumed by processes by */
/* e.g. users, clients */
/*--------------------------*/
printf("%s Set login context........ \n", SERVER);
fflush(stdout);
sec_login_set_context(lcontext, &st);
if (st != error_status_ok)
{
    dce_error_inq_text(st, err_str, &est);
    printf("%s -->Cannot set login-context %d %s\n", SERVER, st, err_str);
    fflush(stdout);
    exit(1);
}

/*-------------------------*/
/* Register authorization info */
/*- registers authorization info with runtime */
/*-------------------------*/
printf("%s Register Authorization info..... \n", SERVER);
fflush(stdout);
/* Registering server authentication */
/* rpc_c_authn_dce_secret (shared secret) */
rpc_server_register_auth_info(SERVER,
rpc_c_authn_none, NULL, NULL, &st);
if (st != error_status_ok)
{
    dce_error_inq_text(st, err_str, &est);
    printf("%s -->Cannot register authorization- %s\n", SERVER, err_str);
    exit(1);
}
printf("%s Identity is established........ \n", SERVER);
fflush(stdout);

Figure 141 (Part 5 of 6). DB2 Server
/* ----------------------------------------- */
/* DIRECTORY CALLS */
/* ----------------------------------------- */
/* Export binding to namespace */
/* ----------------------------------------- */
printf("( %s )Export binding to namespace .... \n",SERVER);
fflush(stdout);
rpc_ns_binding_export(rpc_c_ns_syntax_default,
(unsigned_char_t *)servers/DB2SQL
, DB2SQL_v1_0_s_ifspec,
bvec, (uuid_vector_t *)NULL, &st);

if (st != error_status_ok)
{
dce_error_inq_text(st,err_str,&est);
printf("( %s ) Cannot export binding - %s\n",SERVER,err_str);
exit(1);
} else {
printf("( %s ) Export binding to CDS OK - %s\n",SERVER,err_str);
}
fflush(stdout);
#endif

/* At this point the server waits for clients */
TRY
{
printf("( %s ) listening...\n",SERVER);
fflush(stdout);
rpc_server_listen(numthrds, &st);
}
CATCH_ALL
{
printf("( %s ) Unregistering interface SERVER \n",SERVER);
fflush(stdout);
rpc_server_unregister_if( DB2SQL_v1_0_s_ifspec, NULL,&st);
#endif
#endif
#ifdef SECCDS
printf("( %s ) Unregistering from namespace \n",SERVER);
fflush(stdout);
rpc_ns_binding_unexport(rpc_c_ns_syntax_default,hostname,
DB2SQL_v1_0_s_ifspec, NULL,&st);
#endif
/*------------------------------*/
ENDTRY;
/
/*-----------------------END SERVER2----------------------------*/
D.1.5 Manager Source Code

```c
#include <stdio.h>
#include <pthread.h>
#include <db2sql.h>

#pragma linkage(dsnali, OS)
#pragma linkage(GOFETCH, OS)

/* void gcc(idl_char *c_last,
   idl_long_int *sql_code,
   idl_char *result_msg,
   CUSLST **ppCusLst); */

void get_customersC(handle_t IDL_handle,
   idl_char *c_last,
   idl_long_int *sql_code,
   idl_char *result_msg,
   CUSLST **ppCusLst,
   error_status_t *rpc_comm_status,
   error_status_t *rpc_fault_status)
{
  long int fnret;
  char opnfunc[13] = "OPEN"; /* OPEN */
  char clsfunc[13] = "CLOSE"; /* CLOSE */
  char plan[9] = "DB2SQL"; /* plan name */
  char term_opt[5] = "SYNC"; /* term option */
  long int return_code;
  long int reason_code;

  extern char DB2SSID[];

  fnret = dsnali( /* OPEN */
      opnfunc, /* OPEN */
      DB2SSID, /* subsystem */
      plan, /* plan name */
      &return_code, /* return code */
      &reason_code ); /* reason code */

  printf("Open request fnret = %d\n", fnret);
  fflush(NULL);
  if (fnret != 0)
  {
    sprintf(result_msg,
      "*** UNABLE TO OPEN DB/2 FOR RETURN: %ld REASON: %ld",
      return_code,
      reason_code);
    *rpc_fault_status = 1;
    return;
  }
}
```

Figure 142 (Part 1 of 2). DB2 Manager
GOFETCH("GCC", c_last,
          sql_code,
          result_msg,
          ppCusLst);

/* Close the plans connection with DB/2. */

fnret = dsnali( clsfunc,
              /* CLOSE */
              term_opt,
              /* term option */
              &return_code,
              /* return code */
              &reason_code); /* reason code */

if (fnret != 0)
{
   sprintf(result_msg,
            "*** UNABLE TO CLOSE DB/2 FOR RETURN: %ld REASON: %ld",
            return_code,
            reason_code);
   *rpc_fault_status = 1;
   return;
}

printf("Exit from get_customerC...");

Figure 142 (Part 2 of 2). DB2 Manager
D.1.6 Function Source Code

#include <stdio.h>
#include <db2sql.h>
#pragma linkage(dsnali, OS)
#pragma linkage(DSNHLI, OS)
typedef struct { short SQLPLLEN;
short SQLFLAGS;
short SQLCTYPE;
char SQLPROGN [ 8 ] ;
short SQLTIMES [ 4 ] ;
short SQLSECTN;
char *SQLCODEP;
char *SQLVPARM;
char *SQLAPARM;
short SQLSTNUM;
short SQLSTNUM;
short SQLSType; } SQLPLIST;
typedef struct { short SQLTYPE;
short SQLLEN;
char *SQLADDR;
char *SQLIND; } SQLELTS;
typedef SQLELTS *SQLELTS_PTR;
char SQLTEMP [ 19 ];
void gcc(idl_char *c_last,
idl_long_int *sql_code,
idl_char *result_msg,
CUSLST **ppCusLst){
int i;
/***$$
EXEC SQL INCLUDE SQLCA
$$$***/
#ifndef SQLCODE
struct sqlca{ unsigned char sqlcaid [ 8 ] ;
long sqlcabc;
long sqlcode;
short sqlerrml;
unsigned char sqlerrmc [ 70 ];
unsigned char sqlerrp[ 8 ];
long sqlerrd [ 6 ];
unsigned char sqlwarn [ 11 ];
unsigned char sqlstate [ 5 ]; };
#define SQLCODE sqlca.sqlcode
#define SQLWARNO sqlca.sqlwarn[ 0 ]
#define SQLWARN1 sqlca.sqlwarn[ 1 ]
#define SQLWARN2 sqlca.sqlwarn[ 2 ]
#define SQLWARN3 sqlca.sqlwarn[ 3 ]
#define SQLWARN4 sqlca.sqlwarn[ 4 ]
#define SQLWARN5 sqlca.sqlwarn[ 5 ]
#define SQLWARN6 sqlca.sqlwarn[ 6 ]
#define SQLWARN7 sqlca.sqlwarn[ 7 ]
#define SQLWARN8 sqlca.sqlwarn[ 8 ]
#define SQLWARN9 sqlca.sqlwarn[ 9 ]
#define SQLWARN0 sqlca.sqlwarn[ 10 ]
#define SQLSTATE sqlca.sqlstate
#endif

Figure 143 (Part 1 of 7). DB2 Function Code
struct sqlca sqlca;
/****$
EXEC SQL INCLUDE CUSTOMER
$***/
/***************************************************************************/
/* DCLGEN TABLE(CUSTOMER) */
/* LIBRARY(SWANSON.DCE.DCLGEN(CUSTOMER)) */
/* ACTION(REPLACE) */
/* LANGUAGE(C) */
/* STRUCTURE(customer) */
/* APOST */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS */
/***************************************************************************/
/****$
EXEC SQL DECLARE DAHLE.CUSTOMER TABLE
   ( C_ID CHAR(4) NOT NULL,
     C_D_ID CHAR(2) NOT NULL,
     C_W_ID CHAR(4) NOT NULL,
     C_FIRST CHAR(16) NOT NULL,
     C_MIDDLE CHAR(2) NOT NULL,
     C_LAST CHAR(16) NOT NULL,
     C_STREET_1 CHAR(20) NOT NULL,
     C_STREET_2 CHAR(20) NOT NULL,
     C_CITY CHAR(20) NOT NULL,
     C_STATE CHAR(2) NOT NULL,
     C_ZIP CHAR(9) NOT NULL,
     C_PHONE CHAR(16) NOT NULL,
     C_SINCE TIMESTAMP NOT NULL,
     C_CREDIT CHAR(2) NOT NULL,
     C_CREDIT_LIM DECIMAL(7,2) NOT NULL,
     C_DISCOUNT DECIMAL(4,4) NOT NULL,
     C_BALANCE DECIMAL(10,2) NOT NULL,
     C_YTD_PAYMENT DECIMAL(12,2) NOT NULL,
     C_PAYMENT_CNT DECIMAL(4,0) NOT NULL,
     C_DELIVERY_CNT DECIMAL(4,0) NOT NULL,
     C_DATA VARCHAR(500) NOT NULL
   )
$***/
/****$
EXEC SQL BEGIN DECLARE SECTION
$***/
char FIRST [ 17 ] ;
char MIDDLE [ 3 ] ;
char LAST [ 18 ] ;
char last [ 18 ] ;
char STREET_1 [ 21 ] ;
char STREET_2 [ 21 ] ;
char CITY [ 21 ] ;
char STATE [ 3 ] ;
char ZIP [ 10 ] ;
char PHONE [ 17 ] ;

Figure 143 (Part 2 of 7). DB2 Function Code
EXEC SQL END DECLARE SECTION

*ppCusLst = rpc_ss_allocate(sizeof(CUSTOMER) +
(99 * sizeof(CUSLST *)));
if (*ppCusLst == NULL)
{
    printf("*** Memory Allocation Failed ***\n");
    exit(0);
}
(*ppCusLst)->size = 0;
for (i=0; i < 10; i++)
    (*ppCusLst)->pCustomers [ i ] = NULL;

EXEC SQL DECLARE
C1 CURSOR FOR
SELECT
    C_FIRST,
    C_MIDDLE,
    C_LAST,
    C_STREET_1,
    C_STREET_2,
    C_CITY,
    C_STATE,
    C_ZIP,
    C_PHONE
FROM
    DAHLE.CUSTOMER
WHERE
    C_LAST LIKE :last

Figure 143 (Part 3 of 7). DB2 Function Code
EXEC SQL OPEN C1

SQLPLIST SQLPLIST3 =
{40, -32768, 50, "DB2SQLF", 0, 0, 0,
  1, 0, 0, 0, 106, 3};
SQLELTS_PTR SQLELTS_PTR3;
struct
  { long SQLPVARS;
    char SQLPVELT [ (sizeof(SQLELTS) * 1) ];
  } SQLPVARS3;
SQLELTS_PTR3 = (SQLELTS *) &SQLPVARS3.SQLPVELT;
SQLELTS_PTR3->SQLTYPE = 460;
SQLELTS_PTR3->SQLLEN = 18;
SQLELTS_PTR3->SQLADDR = (char *) &last;
SQLPVARS3.SQLPVARS = 16;
SQLPLIST3.SQLVPARM = (char *) &SQLPVARS3.SQLPVARS;
SQLPLIST3.SQLCODEP = (char *) &sqlca;
SQLPLIST3.SQLTIMES [ 0 ] = 0x153E;
SQLPLIST3.SQLTIMES [ 1 ] = 0x0B6A;
SQLPLIST3.SQLTIMES [ 2 ] = 0x14CC;
SQLPLIST3.SQLTIMES [ 3 ] = 0xBB20;
DSNHLI ( (unsigned int * ) &SQLPLIST3);

printf("SQLCODE = %ld\n", SQLCODE);
if (SQLCODE != 0)
{
  sprintf(result_msg,
    " *** UNABLE TO OPEN CURSOR SQLCODE: %ld",
    SQLCODE);
  return;
}

while (SQLCODE == 0)
{

EXEC SQL FETCH C1 INTO
  :FIRST,
  :MIDDLE,
  :LAST,
  :STREET_1,
  :STREET_2,
  :CITY,
  :STATE,
  :ZIP,
  :PHONE

Figure 143 (Part 4 of 7). DB2 Function Code
{ SQLPLIST SQLPLIST4 =
{ 40, 2048, 30, "DB2SQLF", 0, 0, 0, 1, 0, 0, 0, 118, 4};
SQLELTS_PTR SQLELTS_PTR4;
struct
{ long SQLAVARS;
    char SQLAVELT [ (sizeof(SQLELTS) * 9) ];
} SQLAVARS4;
SQLELTS_PTR4 = (SQLELTS *) &SQLAVARS4.SQLAVELT;
SQLELTS_PTR4->SQLTYPE = 460;
SQLELTS_PTR4->SQLLEN = 17;
SQLELTS_PTR4->SQLADDR = (char *) & (FIRST);
SQLELTS_PTR4->SQLIND = NULL;
SQLELTS_PTR4 = SQLELTS_PTR4 + 1;
SQLELTS_PTR4->SQLTYPE = 460;
SQLELTS_PTR4->SQLLEN = 3;
SQLELTS_PTR4->SQLADDR = (char *) & (MIDDLE);
SQLELTS_PTR4->SQLIND = NULL;
SQLELTS_PTR4 = SQLELTS_PTR4 + 1;
SQLELTS_PTR4->SQLTYPE = 460;
SQLELTS_PTR4->SQLLEN = 18;
SQLELTS_PTR4->SQLADDR = (char *) & (LAST);
SQLELTS_PTR4->SQLIND = NULL;
SQLELTS_PTR4 = SQLELTS_PTR4 + 1;
SQLELTS_PTR4->SQLTYPE = 460;
SQLELTS_PTR4->SQLLEN = 21;
SQLELTS_PTR4->SQLADDR = (char *) & (STREET_1);
SQLELTS_PTR4->SQLIND = NULL;
SQLELTS_PTR4 = SQLELTS_PTR4 + 1;
SQLELTS_PTR4->SQLTYPE = 460;
SQLELTS_PTR4->SQLLEN = 21;
SQLELTS_PTR4->SQLADDR = (char *) & (STREET_2);
SQLELTS_PTR4->SQLIND = NULL;
SQLELTS_PTR4 = SQLELTS_PTR4 + 1;
SQLELTS_PTR4->SQLTYPE = 460;
SQLELTS_PTR4->SQLLEN = 21;
SQLELTS_PTR4->SQLADDR = (char *) & (CITY);
SQLELTS_PTR4->SQLIND = NULL;
SQLELTS_PTR4 = SQLELTS_PTR4 + 1;
SQLELTS_PTR4->SQLTYPE = 460;
SQLELTS_PTR4->SQLLEN = 3;
SQLELTS_PTR4->SQLADDR = (char *) & (STATE);
SQLELTS_PTR4->SQLIND = NULL;
SQLELTS_PTR4 = SQLELTS_PTR4 + 1;
SQLELTS_PTR4->SQLTYPE = 460;
}

Figure 143 (Part 5 of 7). DB2 Function Code
SQLELTS_PTR4->SQLLEN = 10;
SQLELTS_PTR4->SQLADDR = (char *) &ZIP;
SQLELTS_PTR4->SQLIND = NULL;
SQLELTS_PTR4 = SQLELTS_PTR4 + 1;
SQLELTS_PTR4->SQLTYPE = 460;
SQLELTS_PTR4->SQLLEN = 17;
SQLELTS_PTR4->SQLADDR = (char *) &PHONE;
SQLELTS_PTR4->SQLIND = NULL;
SQLAVARS4.SQLAVARS = 112;
SQLPLIST4.SQLAPARM = (char *) &SQLAVARS4.SQLAVARS;
SQLPLIST4.SQLCODEP = (char *) &sqlca;
SQLPLIST4.SQLTIMES [ 0 ] = 0x153E;
SQLPLIST4.SQLTIMES [ 1 ] = 0x0B6A;
SQLPLIST4.SQLTIMES [ 2 ] = 0x14CC;
SQLPLIST4.SQLTIMES [ 3 ] = 0xBB20;
DSNHLI ( (unsigned int *) &SQLPLIST4);
}

*sql_code = SQLCODE;
printf("SQLCODE = %ld\n", SQLCODE);
sprintf(result_msg,
"*** DB2 FETCH COMPLETED WITH SQLCODE: %ld", SQLCODE);
if (SQLCODE == 0)
{ (*ppCusLst)->pCustomers [ (*ppCusLst)->size ] =
  rpc_ss_allocate(sizeof(CUSTOMER));
  strcpy ( (*ppCusLst)->pCustomers [ (*ppCusLst)->size ] ->C_FIRST, FIRST);
  strcpy ( (*ppCusLst)->pCustomers [ (*ppCusLst)->size ] ->C_MIDDLE, MIDDLE);
  strcpy ( (*ppCusLst)->pCustomers [ (*ppCusLst)->size ] ->C_LAST, LAST);
  strcpy ( (*ppCusLst)->pCustomers [ (*ppCusLst)->size ] ->C_STREET_1, STREET_1);
  strcpy ( (*ppCusLst)->pCustomers [ (*ppCusLst)->size ] ->C_STREET_2, STREET_2);
  strcpy ( (*ppCusLst)->pCustomers [ (*ppCusLst)->size ] ->C_CITY, CITY);
  strcpy ( (*ppCusLst)->pCustomers [ (*ppCusLst)->size ] ->C_STATE, STATE);
  strcpy ( (*ppCusLst)->pCustomers [ (*ppCusLst)->size ] ->C_ZIP, ZIP);
  strcpy ( (*ppCusLst)->pCustomers [ (*ppCusLst)->size ] ->C_PHONE, PHONE);
  (*ppCusLst)->size = (*ppCusLst)->size + 1;
}

Figure 143 (Part 6 of 7). DB2 Function Code
EXEC SQL CLOSE C1

{ SQLPLIST SQLPLIST5 = {40, 0, 45, "DB2SQLF ", 0, 0, 0, 0, 1, 0, 0, 0, 160, 5}; SQLLETS_PTR SQLLETS_PTR5; SQLPLIST5.SQLCODEP = (char *) &sqlca; SQLPLIST5.SQLTIMES [ 0 ] = 0x153E; SQLPLIST5.SQLTIMES [ 1 ] = 0x0B6A; SQLPLIST5.SQLTIMES [ 2 ] = 0x14CC; SQLPLIST5.SQLTIMES [ 3 ] = 0xBB20; DSNHLI ((unsigned int *) &SQLPLIST5); }

if (SQLCODE != 0)
{
    *sql_code = SQLCODE;
    sprintf(result_msg,
            "*** UNABLE TO CLOSE CURSOR SQLCODE: %ld",
            SQLCODE);
    return;
}

Figure 143 (Part 7 of 7). DB2 Function Code
D.2 DL/I

Sample source for DL/I Server

D.2.1 IDL

```
[  
  uuid(a9daeea9-f014-14a2-9955-00008c2ec018),  
  version(1.0),  
  pointer_default(ptr)  
]  
interface IMSDLI  
{
  typedef struct
  {
    char part_number [ 18 ] ;  
    char desc [ 16 ] ;  
  } PARTROOT;  
  typedef PARTROOT *PPARTROOT;

typedef struct  
{
  float ss_unit_price;  
  float ss_in_stock;  
} STOKSTAT;  
  typedef STOKSTAT *PSTOKSTAT;

typedef struct  
{
  PARTROOT partroot;  
  long int size;  
  [ size_is(size) ] PSTOKSTAT pStokStat [ * ] ;  
} PARTSTOK;  
  typedef PARTSTOK *PPARTSTOK;

void get_partC  
(  
  [ in,string ] char part_number [ 17 ] ,  
  [ out,string ] char result_msg [ 101 ] ,  
  [ out ] PARTSTOK **ppPartStok  
);  
}
```

Figure 144. IDL
D.2.2 Header

/* Generated by IDL compiler version DCE 1.0.0-1 */
#ifndef IMSDLI_v1_0_included
#define IMSDLI_v1_0_included
#include <dce/idlbase.h>
#include <dce/dcerpcmsg.h>
#include <dce/rpc.h>
#ifdef __cplusplus
extern "C" {
#endif
#include "dce/nbase.h"
typedef struct {
    idl_char part_number [ 18 ] ;
    idl_char desc [ 16 ] ;
} PARTROOT;
typedef PARTROOT *PPARTROOT;
typedef struct {
    idl_short_float ss_unit_price;
    idl_short_float ss_in_stock;
} STOKSTAT;
typedef STOKSTAT *PSTOKSTAT;
typedef struct {
    PARTROOT partroot;
    idl_long_int size;
    PSTOKSTAT pStokStat [ 1 ] ;
} PARTSTOK;
typedef PARTSTOK *PPARTSTOK;
#ifdef IDL_PROTOTYPES
extern void get_partC(
#endif
#ifdef __cplusplus
extern "C" {
#endif
/* [ in ] */ handle_t IDL_handle,
/* [ in ] */ idl_char part_number [ 17 ] ,
/* [ out ] */ idl_char result_msg [ 101 ] ,
/* [ out ] */ PARTSTOK **ppPartStok,
/* [ out ] */ error_status_t *rpc_comm_status,
/* [ out ] */ error_status_t *rpc_fault_status
#endif
#endif
);

Figure 145 (Part 1 of 2). DL/I Header
typedef struct IMSDLI_v1_0_epv_t {
    void (*get_partC)(
        #ifdef IDL_PROTOTYPES
        /* [ in ] */ handle_t IDL_handle,
        /* [ in ] */ idl_char part_number [ 17 ] ,
        /* [ out ] */ idl_char result_msg [ 101 ] ,
        /* [ out ] */ PARTSTOK **ppPartStok,
        /* [ out ] */ error_status_t *rpc_comm_status,
        /* [ out ] */ error_status_t *rpc_fault_status
        #endif
    );
} IMSDLI_v1_0_epv_t;

extern rpc_if_handle_t IMSDLI_v1_0_c_ifspec;
extern rpc_if_handle_t IMSDLI_v1_0_s_ifspec;

#ifdef __cplusplus
}
#endif
#endif

Figure 145 (Part 2 of 2). DL/I Header
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "imsdli.h"
#define NOPAC
#define SIZE 4097
#define NO 0
#define YES 1
#define FALSE 0
#define TRUE 1
#define SEP1 ‘#’
#define SEP2 ‘$’
#define NUL ‘\0’
#define DCE_LOGIN "MYCLIENT"
#define PASSWD "VANAER"
#define SERVER_PRINCIPAL "MYSERVER"
#define SERVER_NAME "/:/servers/IMSDLISV"
#define IFSPEC IMSDLI_v1_0_c_ifspec
#define IP "9.12.13.69"
#define PORT "1095"
#define PROT "ncadg_ip_udp"
/*------------------- D C E -----------------------------*/
#include <dce/rpc.h>
#include <dce/sec_login.h>
#include <dce/dce_error.h>
#include <thread.h>

Figure 146 (Part 1 of 3). DL/I Client
/*-----------------------------------------------*/
/* external static variables used for DCE */
/*-----------------------------------------------*/
int bind_ok;

/*------------------------------------------------------*/
/* pass following arguments */
/* 0) Binding Handle (dummy) */
/* 1) last name char16 in */
/* 2) pointer to sqlcode out */
/* 3) pointer to message out */
/* 4) pointer to ptr to conformant */
/* of pointers to db2rows */
/* 5) pointer to error commstatus */
/* 6) pointer to error faultstatus */
/* -----------------------------------------------*/

int main(int ac, char **av)
{
    short i;

    handle_t IDL_handle;
    idl_char part_number [ 18 ];
    idl_char result_msg [ 101 ];
    PPARTSTOK pPartStok = NULL;
    char lastname [ 17 ];
    char message [ 201 ];
    rpc_binding_handle_t bh;
    rpc_if_handle_t ifhand = IFSPEC;
    error_status_t err1;
    error_status_t err2;
    error_status_t st;
    char CLIENT [ 9 ];
    char prot [ ] = PROT;
    char ip [ ] = IP;
    char port [ ] = PORT;
    char *dceopt = "EP";
    strcpy(CLIENT,"IMSDLICL");
    printf("Starting CLIENT %s 
", CLIENT);
    fflush(stdout);
    bind_ok = FALSE;
    dce_serv(dceopt,
             CLIENT,prot,ip,port,
             ifhand,
             &bh,
             DCE_LOGIN,
             SERVER_PRINCIPAL,
             SERVER_NAME,
             PASSWD);

    Figure 146 (Part 2 of 3). DL/I Client
strcpy(part_number, "023003806");
printf("Calling get_partC. \n");
get_partC(bh,
    part_number,
    result_msg,
    &pPartStok,
    &err1,
    &err2);

if (dce_errt(err1, err2) != 0)
{
    return(-1);
}
else
{
    printf("get_partC result_msg = %s\n", result_msg);
    if (pPartStok != NULL)
    {
        printf("get_partC pPartStok = %08X\n", pPartStok);
        printf("get_partC pPartStok->partroot.part_number = %s\n", (char *) pPartStok->partroot.part_number);
        printf("get_partC pPartStok->partroot.desc = %s\n", (char *) pPartStok->partroot.desc);
        printf("get_partC pPartStok->size = %d\n", pPartStok->size);
        for (i = 0; i < pPartStok->size; i++)
        {
            printf("get_partC pPartStok->pStokStat [ %d ] ->ss_unit_price = %6.3f\n", i, pPartStok->pStokStat [ i ] ->ss_unit_price);
            printf("get_partC pPartStok->pStokStat [ %d ] ->ss_in_stock = %6.1f\n", i, pPartStok->pStokStat [ i ] ->ss_in_stock);
        }
    }
}

Figure 146 (Part 3 of 3). DL/I Client
/* #pragma runopts(execops,nostae,nospie) */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define IMS
#define EP
#define NORPCD
#define NOSECCDS
#define NUL  \0 /* binary X00 */
#define OK 1
#define IFSPEC IMSDLI_v1_0_s_ifspec
#define IFEPVT IMSDLI_v1_0_epv_t
#define IFMGR IMSDLI_V1_0_manager_epv
#ifdef SECCDS
#include <dce/sec_login.h>
#endif
#include <pthread.h>
#include <dce/rpc.h>
#include <dce/dce_cf.h>
#include <dce/exc_handling.h>
/* #include <pthread_exc.h> */
#include imsdl.h
#ifdef IMS
#pragma linkage(CTLINIT,OS)
#pragma linkage(CTLTERM,OS)
#endif
/* globalref IFEPVT IFMGR; */
char DB2SSID [ 5 ] ;

int main(int ac, char *av [ ] )
{
  rpc_binding_vector_p t bvec;
  error_status_t st, est;
  char *string_binding;
  char err_str [ 1024 ] ;
  #ifdef SECCDS
  char dce_login [ 9 ] ;
  char password [ 9 ] ;
  char hostname [ 9 ] ;
  sec_login_handle_t lcontext;
  sec_login_auth_src_t auth_src;
  sec_passwd_rec_t pwrec;
  boolean32 reset_passwd;
  #endif

  Figure 147 (Part 1 of 7). DL/I Server
```c
#ifdef IMS
char request [ 5 ];
char imsuser [ 9 ];
char IMSSUF [ 3 ];
#endif

int numthrds;
int numqueue;
int rc;
int it;
int i;
int ims;
char SERVER [ 9 ];
strcpy(SERVER,"IMSDLISV");
printf("===> Starting SERVER %s\n",SERVER);
#ifdef SECCDS
fflush(stdout);
strcpy(dce_login,"MYSERVER");
strcpy(password,"VANAER");
#endif
rc = 0;
if ( ac > 1 ) {
    if ( ac == 3 ) {
        strncpy(DB2SSID,av [ 1 ] ,4);
        DB2SSID [ 4 ] = NUL;
        printf("Managers %s can connect to DB2 %s\n",SERVER,DB2SSID);
        strncpy(IMSSUF,av [ 2 ] ,2);
        IMSSUF [ 2 ] = NUL;
        printf("Server %s will initialize IMS with member %s\n",SERVER,IMSSUF);
        ims = 0;
    } else {
        /* put DB2SSID in External var for access from managers */
        strncpy(DB2SSID,av [ 1 ] ,4);
        DB2SSID [ 4 ] = NUL;
        printf("Managers %s can connect to DB2 %s\n",SERVER,DB2SSID);
        printf("Server %s will not connect to IMS \n",SERVER);
        ims = -1;
    }
} else {
    printf(" Server %s can not connect to DB2 nor IMS \n",SERVER);
    ims = -1;
}
/* ------------------------------------------------*/
/* Setting the protocol to be used in this program */
/* ------------------------------------------------*/
```

*Figure 147 (Part 2 of 7). DL/I Server*
```c
#ifdef IMS
if ( ims == 0) {
    rc = 0;
    /*------------------------------*/
    /*----Connect to DBCTL ---------*/
    /*------------------------------*/
    numthrds = 5;
    strcpy(imsuser,SERVER);
    printf(" go to CTLINIT for %s\n");
    fflush(stdout);
    /* do also internally tests for completion of INIT */
    rc = CTLINIT(IMSSUF,numthrds,imsuser);
    printf(" after CTLINIT rc %d \n", rc );
    fflush(stdout);
    /*------------------------------*/
    if ( rc > 0 ) {
        printf("Server ended no connection with IMS \n", rc );
        fflush(stdout);
        return(rc);
    }
    /* -------SERVER CONNECTED TO IMS -----------------*/
    ims = OK;
}#endif
numthrds = 2;
numqueue = 2;
#endif
/* register with DCE/RUNTIME protocol + EP */
rpc_server_use_protseq_ep(ncadg_ip_udp,numqueue ,"1095",&st);
#else
/* register with DCE/RUNTIME protocol */
rpc_server_use_protseq(ncadg_ip_udp,numqueue,&st);
#endif
if (st != error_status_ok) {
    dce_error_inq_text(st,err_str,&est);
    printf("Use protocol sequence OK - \n",SERVER);
}
fflush(stdout);
/* ------------------------------------*/
/* Register interface with RPC runtime*/
/* ------------------------------------*/
rpc_server_register_if(IMSDLI_v1_0_s_ifspec, NULL,
    NULL,&st);
/*(rpc_mgr_epv_t) &IFMGR ,&st);*/
if (st != error_status_ok) {
    dce_error_inq_text(st,err_str,&est);
    printf("Register interface OK - \n",SERVER);
}
fflush(stdout);
```

Figure 147 (Part 3 of 7). DL/I Server
/* Get binding handles from the runtime */
rpc_server_inq_bindings(&bvec, &st);

if (st != error_status_ok) {
    dce_error_inq_text(st, err_str, &est);
    printf(“%s --> Cannot inq bindings - %s
”, SERVER, err_str);
    fflush(stdout);
}

/* Display the bindings on the console */
printf("%s
", SERVER);
for (i=0; i < bvec->count; i++)
{
    rpc_binding_to_string_binding(bvec->binding_h[i], &string_binding, &st);
    if (st != error_status_ok)
    {
        dce_error_inq_text(st, err_str, &est);
        printf("%s --> Cannot convert bindings - %s
", SERVER, err_str);
        fflush(stdout);
    }
    printf("%s
", (char*)string_binding);
    fflush(stdout);
    rpc_string_free(&string_binding, &st);
    if (st != error_status_ok)
    {
        dce_error_inq_text(st, err_str, &est);
        printf("%s --> Cannot free mem - %s
", SERVER, err_str);
        fflush(stdout);
    }
}

Figure 147 (Part 4 of 7). DL/I Server
ifdef RPCD
/* -------------------------------------------- */
/* register dynamically EP with endpoint Mapper */
/* -------------------------------------------- */

rpc_ep_register
ifdef SECCDS
( IMSDLI_v1_0_s_ifspec, bvec, NULL,"SEC-CDS", &st);
#else
( IMSDLI_v1_0_s_ifspec, bvec, NULL, NULL, &st);
#endif
if (st != error_status_ok) {
dce_error_inq_text(st,err_str,&est);
printf(( %s )-->Cannot register EP - %s

,SERVER,err_str);
} else {
printf(( %s )Register EP OK - \n",SERVER);
}
fflush(stdout);
#endif

ifdef SECCDS
/--------------------*/
/ SECURITY CALLS */
/--------------------*/
/*/ Establish Identity with security server */
/*/ call goes to registry service (on Security server */
/*/ returns TGT (for priviledge services) and ConvKey */
/--------------------*/
printf(( %s )Establishing Identity with security server... \n",SERVER);
fflush(stdout);
sec_login_setup_identity(dce_login,sec_login_no_flags,&lcontext,&st);
if (st != error_status_ok) {
dce_error_inq_text(st,err_str,&est);
printf(( %s )-->Cannot setup login-identity %s

,SERVER,err_str);
flush(stdout);
exit(1);
}
pwrec.key.tagged_union.plain = "VANAER";
pwrec.key.key_type = sec_passwd_plain;
pwrec.pepper = NULL;
pwrec.version_number = sec_passwd_c_version_none;

Figure 147 (Part 5 of 7). DL/I Server
/* Validation of Identity */
/*- call goes to LOCAL security runtime */
/*- runtime decrypts envelope with PASSWORD */
/*----------------------------------------------------*/

printf("Validation of identity..... \
",SERVER);
fflush(stdout);
sec_login_validate_identity(lcontext, &pwrec, &reset_passwd,
 &auth_src,&st);
if (st != error_status_ok)
{
    dce_error_inq_text(st,err_str,&est);
    printf("--->Cannot validate login-identity %s\n",SERVER,err_str)
    fflush(stdout);
    exit(1);
}
/*----------------------------------------------------*/
/* Set login context */
/*- sets network credentials (in context) */
/*- this context will be assumed by processes by */
/* e.g. users,clients*/
/*----------------------------------------------------*/
printf("Set login context.......... \
",SERVER);
fflush(stdout);
sec_login_set_context(lcontext, &st);
if (st != error_status_ok)
{
    dce_error_inq_text(st,err_str,&est);
    printf("--->Cannot set login-context %d %s\n",SERVER,st,err_str)
    fflush(stdout);
    exit(1);
}
/*----------------------------------------------------*/
/* Register authorization info */
/*- registers authorization info with runtime */
/*----------------------------------------------------*/
printf("Register Authorization info..... \
",SERVER);
fflush(stdout);
/* Registering server authentication */
/* rpc_c_authn_dce_secret (shared secret) */
rpc_server_register_auth_info(SERVER,
rpc_c_authn_none,NULL,NULL,&st);
if (st != error_status_ok)
{
    dce_error_inq_text(st,err_str,&est);
    printf("--->Cannot register authorization- %s\n",SERVER,err_str)
    exit(1);
}
printf("Identity is established......... \
",SERVER);
fflush(stdout);

Figure 147 (Part 6 of 7). DL/I Server
/*-----------------------------------------*/
/* DIRECTORY CALLS */
/*-----------------------------------------*/
/*----------------------------------------------------*/
/* Export binding to namespace */
/*----------------------------------------------------*/
printf(
†( %s )Export binding to namespace .... \n",SERVER);
fflush(stdout);
rpc_ns_binding_export(rpc_c_ns_syntax_default,
(unsigned_char_t *)"/.
.servers/IMSDLI",
IMSDLI_v1_0_s_ifspec,
bvec, (uuid_vector_t *)NULL, &st);
if (st != error_status_ok)
{ dce_error_inq_text(st,err_str,&est);
printf(
†( %s )->Cannot export binding - %s\n",SERVER,err_str);
exit(1);
}
printf("( %s )Export binding to CDS OK - %s\n",SERVER,err_str);
}
fflush(stdout);
#endif
/* At this point the server waits for clients */
TRY
{ printf("( %s )listening...\n",SERVER);
fflush(stdout);
rpc_server_listen(numthrds, &st);
}
CATCH_ALL
{ if (ims == OK) {
printf("( %s )go to CTLTERM \n");
fflush(stdout);
rc = CTLTERM ();
printf("( %s )after CTLTERM rc %d \n", rc );
fflush(stdout);
}
printf("( %s )Unregistering interface SERVER \n",SERVER);
fflush(stdout);
rpc_server_unregister_if( IMSDLI_v1_0_s_ifspec, NULL,&st);
#endif
#ifdef RPCD
printf("( %s )Unregistering endpoint \n",SERVER);
fflush(stdout);
rpc_ep_unregister( IMSDLI_v1_0_s_ifspec, bvec,NULL,&st);
#endif
#ifdef SECCDS
printf("( %s )Unregistering from namespace \n",SERVER);
fflush(stdout);
rpc_ns_binding_unexport(rpc_c_ns_syntax_default,hostname,
IMSDLI_v1_0_s_ifspec, NULL,&st);
#endif
}
ENDTRY;
/

Figure 147 (Part 7 of 7). DL/I Server
D.2.5 Manager Source Code

```c
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <imsdli.h>

#pragma linkage(DFSSCHED,OS)
#pragma linkage(DFSTRMCM,OS)
#pragma linkage(DFSTRMAB,OS)
#pragma linkage(GOFETCH, OS)

/* void GPC( char *part_number,
            idl_char *result_msg,
            PPARTSTOK *ppPartStok);
*/
float zoned_to_float( int length,
                      char *source,
                      int decimal);

void get_partC( handle_t IDL_handle,
                idl_char *part_number,
                idl_char *result_msg,
                PPARTSTOK *ppPartStok,
                error_status_t *rpc_comm_status,
                error_status_t *rpc_fault_status)
{
    int rc;

    printf("\n");
    printf("Entry to get_partC. . .\n");
    fflush(NULL);
    rc = DFSSCHED("DFSSAM08");
    printf("DFSSCHED rc = %d\n", rc);
    fflush(NULL);
    GOFETCH("GPC . . part_number,
              result_msg,
              ppPartStok");

    rc = DFSTRMCM();
    printf("DFSTRMCM rc = %d\n", rc);
    fflush(NULL);
    printf("Exit from get_partC. . .\n");
}
```

Figure 148. DL/I Manager
D.2.6 Function Source Code

```c
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <ims.h>
#include <imsdli.h>

void GPC( char *part_number,
         idl_char *result_msg,
         PPARTSTOK *ppPartStok);

float zoned_to_float( int length,
                      char *source,
                      int decimal);

void GPC(char *part_number,
         idl_char *result_msg,
         PPARTSTOK *ppPartStok)
{
    char buffer[80] ;
    int i ;
    struct {
        char ssa_segment[8];
        char ssa_asterick[1];
        char ssa_command[2];
        char ssa_start[1];
        char ssa_key[8];
        char ssa_logical[2];
        char part_number[17];
        char ssa_end[1];
        char ssa_null[1];
    } ssa_partroot;

    struct {
        char ssa_segment[8];
        char ssa_end[1];
        char ssa_null[1];
    } ssa_stockstat;

    struct {
        char PART_NO[17];
        char FILLER1[9];
        char DESC[15];
        char FILLER2[119];
    } PartRoot;

    struct {
        char FILLER1[20];
        char SS_UNIT_PRICE[9]; /* PIC 9(6)V999. 9 */
        char FILLER3[84];
        char SS_IN_STOCK[8]; /* PIC S9(7)V9. 8 */
    } StockStatus;
}
```

Figure 149 (Part 1 of 4). DL/I Function Code
IO_PCB_TYPE *io_ptr = NULL;
PCB_STRUCT_8_TYPE *di21part_ptr = NULL;
char *GU = "GU"; /* Function codes used */
char *GPSB = "GPSB";
char *GNP = "GNP";
char *GN = "GN";
int three = 3, four = 4, rc;
printf("Entry to GPC. . .\n");
fflush(NULL);
strcpy(result_msg, "*** OK ***");

Figure 149 (Part 2 of 4). DL/I Function Code

Appendix D. IMS and DB2 Examples 303
/*********************************************************/
rc = ctdli(three,GPSB,&io_ptr,&di21part_ptr);
printf("ctdli GPSB rc = %d \n": rc);
fflush(NULL);
if (rc != 0)
{
    return;
}
memset(&PartRoot, \0, sizeof(PartRoot));
rc = ctdli(four,GU,di21part_ptr,&PartRoot,&ssa_partroot);
printf("ctdli GU rc = %d \n": rc);
fflush(NULL);
if (rc != 0)
{
    return;
}

*ppPartStok = (PPARTSTOK) rpc_ss_allocate( sizeof(PARTSTOK) +
    sizeof(PSTOKSTAT) * 9);
printf("*ppPartStok = %ld\n": (int) *ppPartStok);
memset(&( (*ppPartStok)->partroot), \0, sizeof(PARTROOT));
memcpy( (*ppPartStok)->partroot.part_number,
    PartRoot.PART_NO,
    sizeof(PartRoot.PART_NO));
memcpy( (*ppPartStok)->partroot.desc,
    PartRoot.DESC,
    sizeof(PartRoot.DESC));

(*ppPartStok)->size = 0;
for (i = 0; i < 10; i++)
    (*ppPartStok)->pStokStat[ i ] = NULL;
do
{
    memset(&StockStatus, \0, sizeof(StockStatus));
    rc = ctdli(four,GNP,di21part_ptr,&StockStatus,&ssa_stockstat);
    printf("ctdli GNP rc = %d \n": rc);
    fflush(NULL);
    if (rc == 0)
    {
            (PSTOKSTAT) rpc_ss_allocate(sizeof(STOKSTAT));
        (*ppPartStok)->pStokStat[ (*ppPartStok)->size ]->ss_unit_price =
            zoned_to_float( sizeof(StockStatus.SS_UNIT_PRICE),
                StockStatus.SS_UNIT_PRICE,
                3);
            zoned_to_float( sizeof(StockStatus.SS_IN_STOCK),
                StockStatus.SS_IN_STOCK,
                1);
        (*ppPartStok)->size = (*ppPartStok)->size + 1;
    }
}
while ( (rc == 0) || (rc == 'GK') || (rc == 'GA') )
{
    printf("Exit from GPC. . .\n");
    fflush(NULL);
}

float zoned_to_float(int length, char *source, int decimal)
{
    char buffer [ 80 ];
    float result;
    printf("Entry zoned to float. . .\n");
    memset(buffer, '\0', sizeof(buffer));
    memcpy(buffer, source, length);
    printf("buffer = %s\n", buffer);
    buffer [ strlen(buffer) - 1 ] = buffer [ strlen(buffer) - 1 ] | 0xF0;
    printf("fixed buffer = %s\n", buffer);
    result = (float) atoi(buffer);
    printf("result = %6.3f\n", result);
    result = result / pow(10, decimal);
    printf("zoned to float returning = %6.3f\n", result);
    printf("Exit from zoned to float. . .\n");
    return result;
}

Figure 149 (Part 4 of 4). DL/I Function Code
Appendix E. IMS Dependent Region as DCE Client Source

E.1 C Source

E.1.1 C Main

```c
#pragma runopts(env(IMS),plist(IMS),posix(on),stack(12000))
#pragma nomargins
#include <ims.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/*------------------------------------------------------------*/
/* BMP to access DCE services */
/*------------------------------------------------------------*/

main() {

#define ioPcb (IO_PCB_TYPE *)(_pcblist [ 0 ])
#define dbPcb (_pcblist [ 1 ]) /* won’t need this for now */

int rc = 0;
int c1 = 20;
int c2 = 60;
int result = 0;

printf("Entered in BMP MATHXIMS \n");
fflush(stdout);

printf("Adding %d %d \n",c1,c2);
fflush(stdout);

rc = MATHXIC( c1, c2, &result);

printf("Back in BMP MATHXIMS \n");
fflush(stdout);

printf("The result is %d \n",result);
fflush(stdout);

return(0);
}
```

Figure 150. BMP Main Source

© Copyright IBM Corp. 1994
E.1.2 C Subroutine for BMP Source

#include <stdio.h>
#include <stdlib.h>
#include <dce/rpc.h>
#include "errchk.h"
#include "mathx.h"

int MATHXIC ( int c1, int c2, int *presult)
{
    long int i, numv, sum;
    value_array_t va;
    rpc_binding_handle_t bh;
    unsigned32 status;
    unsigned_char_t *string_binding = NULL;

    printf ("In MATHXICC ...
    fflush(NULL);

    rpc_string_binding_compose("","ncadg_ip_udp",
                               "9.12.13.69",
                               "2000",
                               &string_binding,
                               &status);
    ERRCHK ( status );

Figure 151 (Part 1 of 2). C Subroutine for BMP Source
/***************************************************/
/* Generate a bhandle from the string binding */
/***********************************************************/

printf("Generate a binding handle\n");
fflush(NULL);

rpc_binding_from_string_binding ( 
    string_binding, 
    &bh, 
    &status );
ERRCHK ( status );

printf("string_binding = %s\n", string_binding);
fflush(NULL);

printf ( "Values to add %d %d ...
", c1 , c2 );
fflush(NULL);

va [ 0 ] = c1;
va [ 1 ] = c2;
umv = 2;

/***********************************************************/
/* Call add passing handle,values,number of values,ptr to result.*/
/***********************************************************/

printf ( "Call add, pass handle and values, point to result...
" );
fflush(NULL);

add ( bh, va, ( long )numv, &sum );

printf ( "The sum is %d\n", sum );
fflush(NULL);

*presult = sum;

return(0);
}

Figure 151 (Part 2 of 2). C Subroutine for BMP Source
E.2 PLI Source

E.2.1 PLI Main

MATHXB: PROC(IO_PTR,DB1_PTR) OPTIONS(MAIN);
/* DCL PLIXOPT CHAR(100) VARYING
   INIT(POSIX(ON)) STATIC EXTERNAL; */
DCL MATHXIC ENTRY;
DCL IO_PTR PTR,
   DB1_PTR PTR;
DCL 1 IO_PCB BASED(IO_PTR),
    2 LTERM CHAR(8),
    2 RESERVED CHAR(2),
    2 STATUS CHAR(2),
    2 CUR_DATE FIXED DEC(7,0),
    2 CUR_TIME FIXED DEC(7,0),
    2 MSG_SEQ FIXED BIN(31,0),
    2 MOD_NAME CHAR(8),
    2 USERID CHAR(8),
    2 GRPNAME CHAR(8);

DCL 1 DB1_PCB BASED(DB1_PTR),
    2 DATA_BASE CHAR(8),
    2 SEG_LEVEL CHAR(2),
    2 STATUS CHAR(2),
    2 PROCOPT CHAR(4),
    2 RESERVED CHAR(4),
    2 SEG_NAME CHAR(8),
    2 KEY_FB_LEN FIXED BIN(31,0),
    2 NUM_SENSEG FIXED BIN(31,0),
    2 KEY_FB_AREA CHAR(1);

DCL THREE FIXED BIN(31,0) INIT(3),
    FOUR FIXED BIN(31,0) INIT(4),
    FIVE FIXED BIN(31,0) INIT(5);
DCL GPSB CHAR(4) INIT('GPSB');
DCL GU CHAR(4) INIT('GU ');
DCL IOZONE CHAR(100);
DCL 1 SSA1,
    2 SEGN CHAR(8) INIT('PARTROOT'),
    2 LH CHAR(1) INIT(''),
    2 FLD CHAR(8) INIT('PARTKEY'),
    2 OPER CHAR(2) INIT('=>'),
    2 KEY CHAR(17) INIT('AAAAAAAAAAAAAAAAA'),
    2 RH CHAR(1) INIT('');
DCL C1 FIXED BIN(31,0);
DCL C2 FIXED BIN(31,0);
DCL RESULT FIXED BIN(31,0);
DCL RESULT_P PTR;
DCL CRESULT CHAR(4),
PRESULT PICZZZ9 DEF(CRESULT);

/* IMS BMP ACTING AS A CLIENT FOR DCE */

DISPLAY('BMP CLIENT GOT CONTROL');

C1 = 20;
C2 = 40;
RESULT = 0;
PUT SKIP LIST (' BEFORE ' . C1, C2, RESULT);

RESULT_P = ADDR(RESULT);
CALL MATHXIC(C1, C2, RESULT_P);
RC = PLIRETV();

PUT SKIP LIST (' AFTER ' . C1, C2, RESULT);

DISPLAY('BMP BACK IN CONTROL');

PRESULT = RESULT;
DISPLAY('RESULT = ' || CRESULT);

DISPLAY('RETURN TO STUB');

END MATHXB;

Figure 152 (Part 2 of 2). BMP Main Source
E.2.2 PLI Subroutine for BMP Source

#include <stdio.h>
#include <stdlib.h>
#include <dce/rpc.h>
#include "errchk.h"
#include "mathx.h"

#pragma runopts(POSIX(ON))
#pragma linkage(MATHXIC,PLI)

int MATHXIC ( int c1, int c2, int **ppresult)
{
    long int i, numv, sum;
    value_array_t va;
    rpc_binding_handle_t bh;
    unsigned32 status;
    unsigned_char_t *string_binding = NULL;
    printf ("In MATHXICP ...
    " );
    fflush(NULL);
    printf ("Values to add %d %d ...
    " , c1, c2 );
    fflush(NULL);

    /* Compose */
    rpc_string_binding_compose(
        "","ncadg_ip_udp",
        "9.12.13.69",
        "2000",
        &string_binding,
        &status);
    ERRCHK ( status );

Figure 153 (Part 1 of 2). PLI Subroutine for BMP Source
/*******************************/  
/* Generate a bhandle from the string binding */  
/*******************************/  

printf(“Generate a binding handle\n”);
fflush(NULL);

rpc_binding_from_string_binding (  
    string_binding,  
    &bh,  
    &status );
ERRCHK ( status );

printf(“string_binding = %s\n", string_binding);
fflush(NULL);

va [ 0 ] = c1;
va [ 1 ] = c2;
numv = 2;

/*******************************/  
/* Call add passing handle,values,number of values,ptr to result.*/  
/*******************************/  

printf (“Call add, pass handle and values, point to result...\n”);
fflush(NULL);

add ( bh, va, ( long )numv, &sum );

printf (“The sum is %d\n", sum);
fflush(NULL);

**ppresult = sum;
return(0);
}

Figure 153 (Part 2 of 2). PLI Subroutine for BMP Source
IDENTIFICATION DIVISION.
    PROGRAM-ID. MATHXIMS.
    AUTHOR. S REISER.
    INSTALLATION POK.
    DATE-WRITTEN AUG 1994.

*=================================================================
* THIS IS A COBOL PGM WHO WILL USE A "C"
*=================================================================

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
    FILE-CONTROL.
* ASSOCIATE FUNCTIONS WITH IO TYPES
DATA DIVISION.
FILE SECTION.
* DEFINE THE FILE TYPES
WORKING-STORAGE SECTION.
  01 RESULTP PIC S9(9) COMP.
  01 C1 PIC S9(9) COMP VALUE 20.
  01 C2 PIC S9(9) COMP VALUE 40.

PROCEDURE DIVISION.
    DISPLAY "ENTERED MATHXIC1 - CLIENT STARTUP".
    PERFORM 100-INITIALIZE-PARAGRAPH.
    PERFORM 100-MAIN-PARAGRAPH.
    DISPLAY "PROGRAM MATHXIC1 - NORMAL END".
    STOP RUN.
100-INITIALIZE-PARAGRAPH.
    DISPLAY "IN-->100-INITIALIZE-PARAGRAPH".
100-MAIN-PARAGRAPH.
    DISPLAY "IN 100-MAIN-PARAGRAPH".
    PERFORM 200-CALL-CLIENT.
    PERFORM 200-LIST-INFORMATION.
200-CALL-CLIENT.
    DISPLAY "IN-->200-CALL-CLIENT".
    DISPLAY "BEFORE-->MATHCICC".
    DISPLAY C1.
    DISPLAY C2.
    CALL "MATHXICB" USING BY CONTENT C1 C2 REFERENCE RESULTP.
    DISPLAY "AFTER <--MATHCICC".
    DISPLAY C1 C2.
    DISPLAY RESULTP.
200-LIST-INFORMATION.

Figure 154. BMP Main Source
E.3.2 COBOL Subroutine for BMP Source

```c
#include <stdio.h>
#include <stdlib.h>
#include <dce/rpc.h>
#include "errchk.h"
#include "mathx.h"

int MATHXICB ( int c1, int c2, int *ppresult)
{
    long int i, numv, sum;
    value_array_t va;
    rpc_binding_handle_t bh;
    unsigned32 status;
    unsigned_char_t *string_binding = NULL;
    printf ("In MATHXICB ...
    fflush(NULL);
    printf ("Values to add %i %i ...
    c1, c2 );
    fflush(NULL);
    /********************************************************/
    rpc_string_binding_compose(
        "ncadg_ip_udp",
        "9.12.13.69",
        "2000",
        &string_binding,
        &status);
    ERRCHK ( status );
}```

Figure 155 (Part 1 of 2). COBOL Subroutine for BMP Source
/***************************************************************************
/* Generate a bhandle from the string binding  */
***************************************************************************
printf("Generate a binding handle\n");
fflush(NULL);
rpc_binding_from_string_binding (  
    string_binding,  
    &bh,  
    &status );
ERRCHK ( status );
printf("string_binding = %s\n", string_binding);
fflush(NULL);
va [ 0 ] = c1;
va [ 1 ] = c2;
umv = 2;
***************************************************************************
/* Call add passing handle,values,number of values,ptr to result.*/
***************************************************************************
printf ( "Call add, pass handle and values, point to result...\n");
fflush(NULL);
add ( bh, va, ( long )numv, &sum );
printf ( "The sum is %i\n", sum );
fflush(NULL);
*ppresult = sum;
return(0);
}
Appendix F. Dynamic SQL Server

This appendix contains the source for the dynamic SQL server. It contains the IDL; generated header file; the client, server, and manager code.

F.1 IDL

```idl
[uuid(c3de6a7a-fe97-11cc-8e01-0800095415c2),
version(1.0),
pointer_default(ptr)
]
interface SQLDSV
{
 /* Constant definitions for Column Data Types */
 const unsigned long MAXELEMENTS = 65000;
 /* largest defined type */
 const unsigned short CDT_max = 13;

 const unsigned short CDT_undefined = 0;
 const unsigned short CDT_integer = 1;
 const unsigned short CDT_string = 2;
 const unsigned short CDT_boolean = 3;
 const unsigned short CDT_double = 4;
 const unsigned short CDT_rowid = 5;
 const unsigned short CDT_date = 6;
 const unsigned short CDT_raw = 7;
 const unsigned short CDT_time = 8;
 const unsigned short CDT_tmstmp = 9;
 const unsigned short CDT_smallint = 10;
 const unsigned short CDT_float = 11;
 const unsigned short CDT_packed = 12;

 /* Constant definitions for Indicator variables */
 const short CDI_overflow = -2;
 const short CDI_nullvalue = -1;
 const short CDI_ok = 0;

 /* Constant definitions for flags */
 const unsigned short CDF_moreData = (1<<0);
 const unsigned short CDF_shipLabels = (1<<1);
 const unsigned short CDF_shipWidth = (1<<2);
 const unsigned short CDF_shipIndicators = (1<<3);
```

Figure 156 (Part 1 of 4). Dynamic SQL Server IDL
typedef [ ptr,string ] char *string;

typedef struct ValueListExtHeader {
    struct ValueListExtHeader *next;
    char blkType [ 4 ];
    unsigned short blkversion;
    unsigned short blksize;
} ValueListExtHeader;

typedef struct StringColType {
    unsigned short nrows;
    [ ptr,size_is(nrows) ] short *indicator;
    [ ptr,size_is(nrows) ] string *data;
} StringColType;

typedef struct CharColType {
    unsigned short nrows;
    [ ptr,size_is(nrows) ] short *indicator;
    unsigned short nbytes;
    [ ptr,size_is(nbytes) ] char *data;
} CharColType;

typedef struct ByteColType {
    unsigned short nrows;
    [ ptr,size_is(nrows) ] short *indicator;
    unsigned short nbytes;
    [ ptr,size_is(nbytes) ] byte *data;
} ByteColType;

typedef struct IntegerColType {
    unsigned short nrows;
    [ ptr,size_is(nrows) ] short *indicator;
    [ ptr,size_is(nrows) ] long *data;
} IntegerColType;

typedef struct SmallintColType {
    unsigned short nrows;
    [ ptr,size_is(nrows) ] short *indicator;
    [ ptr,size_is(nrows) ] short *data;
} SmallintColType;

typedef struct DateColType {
    unsigned short nrows;
    [ ptr,size_is(nrows) ] short *indicator;
    unsigned short nbytes;
    [ ptr,size_is(nbytes) ] char *data;
} DateColType;

typedef struct TimeColType {
    unsigned short nrows;
    [ ptr,size_is(nrows) ] short *indicator;
    unsigned short nbytes;
    [ ptr,size_is(nbytes) ] char *data;
} TimeColType;

Figure 156 (Part 2 of 4). Dynamic SQL Server IDL
typedef struct TmstmpColType {
    unsigned short nrows;
    [ ptr, size_is(nrows) ] short *indicator;
    unsigned short nbytes;
    [ ptr, size_is(nbytes) ] char *data;
} TmstmpColType;

typedef struct BooleanColType {
    unsigned short nrows;
    [ ptr, size_is(nrows) ] short *indicator;
    [ ptr, size_is(nrows) ] boolean *data;
} BooleanColType;

typedef struct DoubleColType {
    unsigned short nrows;
    [ ptr, size_is(nrows) ] short *indicator;
    [ ptr, size_is(nrows) ] double *data;
} DoubleColType;

typedef struct FloatColType {
    unsigned short nrows;
    [ ptr, size_is(nrows) ] short *indicator;
    [ ptr, size_is(nrows) ] float *data;
} FloatColType;

typedef struct RawData {
    unsigned short nbytes;
    [ ptr, size_is(nbytes) ] byte *bytes;
} RawData;

typedef struct RawColType {
    unsigned short nrows;
    [ ptr, size_is(nrows) ] short *indicator;
    [ ptr, size_is(nrows) ] struct RawData *data;
} RawColType;

Figure 156 (Part 3 of 4). Dynamic SQL Server IDL
typedef /* [ switch_type(unsigned short) ] */
    union ColDataType switch (unsigned short colDataType) ColDataUnion {
    case CDT_undefined: ;
    case CDT_integer: struct IntegerColType integerCol;
    case CDT_string: struct StringColType stringCol;
    case CDT_boolean: struct BooleanColType booleanCol;
    case CDT_double: struct DoubleColType doubleCol;
    case CDT_rowid: struct CharColType rowidCol;
    case CDT_date: struct DateColType dateCol;
    case CDT_raw: struct RawColType rawCol;
    case CDT_char: struct CharColType charCol;
    case CDT_time: struct TimeColType timeCol;
    case CDT_tmstmp: struct TmstmpColType tmstmpCol;
    case CDT_smallint: struct SmallintColType smallintCol;
    case CDT_float: struct FloatColType floatCol;
    case CDT_packed: struct ByteColType packedCol;
    default: ;
} ColDataType;

typedef struct ValueList {
    struct ValueListExtHeader *header;
    unsigned short ncols;
    unsigned short nrows;
    unsigned short nrowsProcessed;
    unsigned short flags;

    unsigned short *colWidth;
    string *colLabel;
    ColDataType *colData;
} ValueList;

typedef struct ValueList *VLHandle;

long int SQL_SRV_EXEC ( {
    handle_t h,
    string dbplname,
    string sql_stmt,
    ValueList *result,
    long int *pcond,
    char output [ 201 ],
    char operation [ 21 ]
});
/* Generated by IDL compiler version DCE 1.0.0-1 */
#ifndef SQLDSV_v1_0_included
#define SQLDSV_v1_0_included
#endif
#ifndef IDL_DOUBLE_USED
#define IDL_DOUBLE_USED
#endif
#include <dce/idlbase.h>
#include <dce/dcerpcmsg.h>
#include <dce/rpc.h>

#ifdef __cplusplus
extern "C"
{
#endif
#include "dce/nbase.h"

#define MAXELEMENTS (65000)
#define CDT_max (13)
#define CDT_undefined (0)
#define CDT_integer (1)
#define CDT_string (2)
#define CDT_boolean (3)
#define CDT_double (4)
#define CDT_rowid (5)
#define CDT_date (6)
#define CDT_raw (7)
#define CDT_char (8)
#define CDT_time (9)
#define CDT_tmstmp (10)
#define CDT_smallint (11)
#define CDT_float (12)
#define CDT_packed (13)
#define CDI_overflow (-2)
#define CDI_nullvalue (-1)
#define CDI_ok (0)
#define CDF_moreData (1)
#define CDF_shipLabels (2)
#define CDF_shipWidth (4)
#define CDF_shipIndicators (8)

Figure 157 (Part 1 of 5). Dynamic SQL Server Header
typedef idl_char *string;

typedef struct ValueListExtHeader {
    struct ValueListExtHeader *next;
    idl_char blkType [ 4 ] ;
    idl_ushort_int blkversion;
    idl_ushort_int blksize;
} ValueListExtHeader;

typedef struct StringColType {
    idl_ushort_int nrows;
    idl_short_int *indicator;
    string *data;
} StringColType;

typedef struct CharColType {
    idl_ushort_int nrows;
    idl_short_int *indicator;
    idl_ushort_int nbytes;
    idl_char *data;
} CharColType;

typedef struct ByteColType {
    idl_ushort_int nrows;
    idl_short_int *indicator;
    idl_ushort_int nbytes;
    idl_byte *data;
} ByteColType;

typedef struct IntegerColType {
    idl_ushort_int nrows;
    idl_short_int *indicator;
    idl_long_int *data;
} IntegerColType;

typedef struct SmallintColType {
    idl_ushort_int nrows;
    idl_short_int *indicator;
    idl_short_int *data;
} SmallintColType;

typedef struct DateColType {
    idl_ushort_int nrows;
    idl_short_int *indicator;
    idl_ushort_int nbytes;
    idl_char *data;
} DateColType;

typedef struct TimeColType {
    idl_ushort_int nrows;
    idl_short_int *indicator;
    idl_ushort_int nbytes;
    idl_char *data;
} TimeColType;

Figure 157 (Part 2 of 5). Dynamic SQL Server Header
typedef struct TmstmpColType {
  idl_ushort_int nrows;
  idl_short_int *indicator;
  idl_ushort_int nbytes;
  idl_char *data;
} TmstmpColType;

typedef struct BooleanColType {
  idl_ushort_int nrows;
  idl_short_int *indicator;
  idl_boolean *data;
} BooleanColType;

typedef struct DoubleColType {
  idl_ushort_int nrows;
  idl_short_int *indicator;
  idl_long_float *data;
} DoubleColType;

typedef struct FloatColType {
  idl_ushort_int nrows;
  idl_short_int *indicator;
  idl_short_float *data;
} FloatColType;

typedef struct RawData {
  idl_ushort_int nbytes;
  idl_byte *bytes;
} RawData;

typedef struct RawColType {
  idl_ushort_int nrows;
  idl_short_int *indicator;
  struct RawData *data;
} RawColType;

Figure 157 (Part 3 of 5). Dynamic SQL Server Header
typedef struct ColDataType {
    idl_ushort_int colDataType;
    union {
        /* case(s): 0 */
        /* Empty arm */
        /* case(s): 1 */
        struct IntegerColType integerCol;
        /* case(s): 2 */
        struct StringColType stringCol;
        /* case(s): 3 */
        struct BooleanColType booleanCol;
        /* case(s): 4 */
        struct DoubleColType doubleCol;
        /* case(s): 5 */
        struct CharColType rowidCol;
        /* case(s): 6 */
        struct DateColType dateCol;
        /* case(s): 7 */
        struct RawColType rawCol;
        /* case(s): 8 */
        struct CharColType charCol;
        /* case(s): 9 */
        struct TimeColType timeCol;
        /* case(s): 10 */
        struct TmstmpColType tmstmpCol;
        /* case(s): 11 */
        struct SmallintColType smallintCol;
        /* case(s): 12 */
        struct FloatColType floatCol;
        /* case(s): 13 */
        struct ByteColType packedCol;
        /* case(s): default */
        /* Empty arm */
    } ColDataUnion;
} ColDataType;

typedef struct ValueList {
    ValueListExtHeader *header;
    idl_ushort_int ncols;
    idl_ushort_int nrows;
    idl_ushort_int nrowsProcessed;
    idl_ushort_int flags;
    idl_ushort_int *colWidth;
    string *colLabel;
    ColDataType *colData;
} ValueList;

typedef struct ValueList *VLHandle;

Figure 157 (Part 4 of 5). Dynamic SQL Server Header
externidl_long_int SQL_SRV_EXEC(
#ifdefIDL_PROTOTYPES
    /* [ in ] */ handle_t h,
    /* [ in ] */ string dbplname,
    /* [ in ] */ string sql_stmt,
    /* [ in, out ] */ ValueList *result,
    /* [ out ] */ idl_long_int *pcond,
    /* [ out ] */ idl_char output [ 201 ],
    /* [ out ] */ idl_char operation [ 21 ],
    /* [ out ] */ error_status_t *c_sts,
    /* [ out ] */ error_status_t *f_sts
#endif
);

typedef struct SQLDSV_v1_0_epv_t {
    idl_long_int (*SQL_SRV_EXEC)(
#ifdefIDL_PROTOTYPES
        /* [ in ] */ handle_t h,
        /* [ in ] */ string dbplname,
        /* [ in ] */ string sql_stmt,
        /* [ in, out ] */ ValueList *result,
        /* [ out ] */ idl_long_int *pcond,
        /* [ out ] */ idl_char output [ 201 ],
        /* [ out ] */ idl_char operation [ 21 ],
        /* [ out ] */ error_status_t *c_sts,
        /* [ out ] */ error_status_t *f_sts
#endif
    );
} SQLDSV_v1_0_epv_t;

extern rpc_if_handle_t SQLDSV_v1_0_c_ifspec;
extern rpc_if_handle_t SQLDSV_v1_0_s_ifspec;

#ifdef__cplusplus
}
#endif
#endif

Figure 157 (Part 5 of 5). Dynamic SQL Server Header
F.3 Client Source Code

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "SQLDSV.h"

#define DCE
#define NOCDSEC
#define SIZE 4097
#define NO 0
#define YES 1
#define SEP1 '\''
#define SEP2 'S'
#define NUL '\0'
#define DCE_LOGIN "MYCLIENT"
#define PASSWD "VANAER"
#define SERVER_PRINCIPAL "MYSERVER"
#define SERVER_NAME "/.:/servers/SQLDSVSV"
#define IFSPEC SQLDSV_v1_0_c_ifspec
#define IP "9.12.13.69"
#define PORT "1070"
#define PROT "ncadg_ip_udp"

/*---------- defines for SQL ------------------------------------*/
#define T_DATE 384 /* length 10 character string */
#define TN_DATE 385 /* length 10 + NULLS character string */
#define T_TIME 388 /* length 8 character string */
#define TN_TIME 389 /* length 8 + NULLS character string */
#define T_TMSTMP 392 /* length 26 character string */
#define TN_TMSTMP 393 /* length 26 + NULLS character string */
#define T_VCHAR 448
#define TN_VCHAR 449
#define T_CHAR 452
#define TN_CHAR 453
#define T_LVCHAR 456
#define TN_LVCHAR 457
#define T_GVCHAR 464
#define TN_GVCHAR 465
#define T_GCHAR 468
#define TN_GCHAR 469
#define T_GLCHAR 472
#define TN_GLCHAR 473
#define T_FLOAT 480 /* length 4 of 8 */
#define TN_FLOAT 481 /* length 4 of 8 + NULLS */
#define T_DEC 484 /* length 1/2/3/4/5/6/7/8 */
#define TN_DEC 485 /* length 1/2/3/4/5/6/7/8 + NULLS */
#define T_INT 496 /* length 4 */
#define TN_INT 497 /* length 4 + NULLS */
#define T_SMINT 500 /* length 2 */
#define TN_SMINT 501 /* length 2 + NULLS */
```

Figure 158 (Part 1 of 7). Dynamic SQL Server Client
/*-----------------------------------------------*/
#define SQLDA_FIX 16
#define SQLDA_VAR 44
#define BLANKS ""
#define FLOATLEN 4
#define DOUBLELEN 8
/*-----------------------------------------------*/
#define FALSE 0
#define TRUE 1
#define NO 0
#define YES 1
#define NUL \0
/*-----------------------------------------------*/
#ifdef DCE
/*------------------------------------------------------*/
/* external static variables used for DCE */
/*------------------------------------------------------*/
int bind_ok;
/*------------------------------------------------------*/
 ifdef DCE
/*------------------------------------------------------*/
/* pass following arguments */
/* 0) Binding Handle (dummy) */
/* 1) pointer to DBPlan_name */
/* 2) pointer to SQL statement */
/* 3) pointer to cond from DCE */
/* 4) pointer to output from DCE */
/* 5) pointer to operation from DCE */
/* 6) pointer to ValueList from DCE */
/* 7) pointer to error commstatus */
/* 8) pointer to error faultstatus */
/*------------------------------------------------------*/
Figure 158 (Part 2 of 7). Dynamic SQL Server Client
int main(int ac, char **av)
{
    char dbplname[9];  /* from terminal in */
    char input[101];  /* from terminal in */
    long cond;  /* to terminal out */
    char output[SIZE];  /* to terminal out */
    char operation[21];  /* to terminal out */
    unsigned long int flags;  /* to terminal out */
    FILE *stream;
    id1_boolean boolrc;
    ValueList my_valuelist;
    rpc_binding_handle_t bh;
    rpc_if_handle_t ifhand = IFSPEC;
    error_status_t err1;
    error_status_t err2;
    long st;
    long rc;
    long i, j, l;
    long npos;
    long ncols;
    char ch;
    string new_s_ptr;
    string cdt_ptr;
    ColDataType *coldata_ptr;
    string *collabel_ptr;
    char *collabel;
    void *data_ptr;
    short *ind_ptr;
    short lensql;
    long coltype;
    unsigned short colwidth;
    long my_int;
    short my_smi;
    float my_flo;
    double my_dbl;
    VLHandle my_VLHandle;
    char CLIENT[9];
    char prot[ ] = PROT;
    char ip[ ] = IP;
    char port[ ] = PORT;
    char *dceopt = "EP";
    char *o = &output[0];
    char *op = &operation[0];
    strcpy(CLIENT,"SQLDSVCL");
    printf("==>	Starting CLIENT %s
",CLIENT);
    fflush(stdout);
    strcpy(dbplname,"SQLDSVMA");

    Figure 158 (Part 3 of 7). Dynamic SQL Server Client
/*-------------------------------------*/
/* read SQL Statement */
/*-------------------------------------*/
if ((stream = fopen("dd:SQLIN","r")) != NULL) {
    for ( i = 0; (i < (sizeof(input) - 1) && (ch = fgetc(stream)) != ']' && (ch != '\n')); i++) {
        input [ i ] = ch;
    }
    input [ i ] = '\0';
    fclose(stream);
} else {
    printf("==> %s Can NOT open SQLIN \n",CLIENT);
    fflush(stdout);
    return(0);
}
strcpy(output,"");
strcpy(operation,""):
/* flags = CDF_shipLabels + CDF_shipIndicators; */
flags = 0;
printf("Flags = %d Input = %s \n",flags,input);
my_valuelist.header = NULL;
my_valuelist.ncols = 0;
my_valuelist.nrows = 60;
my_valuelist.nrowsProcessed = 0;
my_valuelist.flags = flags;
my_valuelist.colWidth = NULL;
my_valuelist.colLabel = NULL;
my_valuelist.colData = NULL;
my_VLHandle = &my_valuelist;
/*-------------------------------------*/
bind_ok = FALSE;
dce_serv(dceopt,
        CLIENT,prot,ip,port,
        ifhand,
        &hh,
        DCE_LOGIN,
        SERVER_PRINCIPAL,
        SERVER_NAME,
        PASSWD);
*/-------------------------------------*/

Figure 158 (Part 4 of 7). Dynamic SQL Server Client
rc = SQL_SRV_EXEC ( bh, dbplname, input,
    my_VLHandle, &cond, o, op, &err1,&err2);
/*-----------------------------------------------*/

if (dce_errt(err1, err2) != 0) {
    return(-1);
} else {
    if ((cond == 0) || (cond == 100)) {
        printf("%s analyzing value list \n", CLIENT);
        if ((void *)my_valuelist.header != NULL) {
            printf("- Valuelist Header \n");
            printf("blkType %c%c%c%c version %d size %d \n", my_valuelist.header->blkType[0], my_valuelist.header->blkType[1], my_valuelist.header->blkType[2], my_valuelist.header->blkType[3], my_valuelist.header->blkversion, my_valuelist.header->blksize);
            fflush(stdout);
        } /* endif flags */
        printf("Requested %d rows processed %d rows \n", my_valuelist.nrows, my_valuelist.nrowsProcessed);
        printf("Containing %d cols flags are %d \n", my_valuelist.ncols, my_valuelist.flags);
        if ( flags & CDF_shipLabels ) {
            for ( i = 0; i < my_valuelist.ncols; i++) {
                collabel = my_valuelist.colLabel[ i ];
                printf("%s", collabel);
            } /* endif flags */
            printf("\n");
            fflush(stdout);
            for ( j = 0; j < my_valuelist.nrowsProcessed; j++) {
                #ifdef DB2
                /* T_DEC has been converted to */
                /* T_INT T_SMINT T_FLOAT */
                #endif
                for ( i = 0; i < my_valuelist.ncols; i++) {
                    coldata_ptr = my_valuelist.colData + i;
                    colwidth = my_valuelist.colWidth[ i ];
                    coltype = coldata_ptr->colDataType;
                    ind_ptr = coldata_ptr->ColDataUnion.rawCol.indicator;
                    if ( flags & CDF_shipIndicators ) {
                        if (ind_ptr != NULL )
                            printf("(%d)", ind_ptr->ColDataUnion.rawCol.indicator[ j ]); else printf("() ");
                }
        }}
} /* endif flags */

Figure 158 (Part 5 of 7). Dynamic SQL Server Client
while (1) {

    switch (coltype) {
    case CDT_integer:
        my_int =
            coldata_ptr->ColDataUnion.integerCol.data [ j ];
        printf("%d", my_int);
        break;

    case CDT_smallint:
        my_smi =
            coldata_ptr->ColDataUnion.smallintCol.data [ j ];
        printf("%d", my_smi);
        break;

    case CDT_double:
        my_dbl =
            coldata_ptr->ColDataUnion.doubleCol.data [ j ];
        printf("%f", my_dbl);
        break;

    case CDT_float:
        my_flo =
            coldata_ptr->ColDataUnion.floatCol.data [ j ];
        printf("%f", my_flo);
        break;

    case CDT_string:
        data_ptr =
            coldata_ptr->ColDataUnion.stringCol.data [ j ];
        l = strlen(data_ptr);
        printf("%s", data_ptr);
        break;

    case CDT_char:
        lensql = colwidth;
        data_ptr =
            coldata_ptr->ColDataUnion.charCol.data +
            ( j * lensql);
        l = strlen(data_ptr);
        printf("%s", data_ptr);
        break;

    case CDT_date:
        npos = j * colwidth;
        cdt_ptr =
            (char *)coldata_ptr->ColDataUnion.dateCol.data;
        cdt_ptr = cdt_ptr + npos;
        printf("%s", cdt_ptr);
        break;
    }

Figure 158 (Part 6 of 7). Dynamic SQL Server Client
case CDT_time:
    npos = j * colwidth;
    cdt_ptr =
        (char *)coldata_ptr->ColDataUnion.timeCol.data;
    cdt_ptr = cdt_ptr + npos;
    printf("%s ", cdt_ptr);
    break;

case CDT_tmstmp:
    npos = j * colwidth;
    cdt_ptr = (char *)
        coldata_ptr->ColDataUnion.tmstmpCol.data;
    cdt_ptr = cdt_ptr + npos;
    printf("%s ", cdt_ptr);
    break;

case CDT_packed:
    printf("CDT_type %d not yet processed \n", coltype);
    break;

default:
    printf("invalid CDT_type %d detected \n", coltype);
    break;
    } /* end SWITCH */
    break;
} /* end WHILE */
} /* end FOR 2 */
    printf("\n");
    flush(stdout);
    } /* end FOR 1 */
} /* end else cond */
    printf("( %s )cond = %d %s \n", CLIENT, cond, output);
}
    exit(0);
} /* end else err1 err2 */
} /* end else cond */

Figure 158 (Part 7 of 7). Dynamic SQL Server Client
F.4 Server Source Code

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define NOIMS
#define EP
#define NORPCD
#define NOSECCDS
#define NUL \0 /* binary X00 */
#define IFSPEC SQLDSV_v1_0_s_ifspec
#define IFEPVT SQLDSV_v1_0_epv_t
#define IFMGR SQLDSV_V1_0_manager_epv
#ifdef SECCDS
#include <dce/sec_login.h>
#endif
#include <pthread.h>
#include <dce/rpc.h>
#include <dce/dce_cf.h>
#include <dce/exc_handling.h>
/* #include <pthread_exc.h> */
#include SQLDSV.h
/* globalref IFEPVT IFMGR; */

char DB2SSID [ 5 ];
int main(int ac, char *av [ ] )
{
    rpc_binding_vector_p_t bvec;
    error_status_t st, est;
    char *string_binding;
    char err_str [ 1024 ] ;
    #ifdef SECCDS
    char dce_login [ 9 ] ;
    char password [ 9 ] ;
    char hostname [ 9 ] ;
    sec_login_handle_t lcontext;
    sec_login_auth_src_t auth_src;
    sec_passwd_rec_t pwrec;
    boolean32 reset_passwd;
    #endif
```

Figure 159 (Part 1 of 8). Dynamic SQL Server
#ifdef IMS
    char request [ 5 ];
    char imsuser [ 9 ];
#endif
int numthrds;
int numqueue;
int rc;
int it;
int i;
int ims;
char SERVER [ 9 ];
strcpy(SERVER, "SQLDSVSV");
printf("=> Starting SERVER %s
", SERVER);
#endif
rc = 0;
/*----------------------------------------------*/
/* 1 parameter is passed to the SerVerCode */
/* DB2 Subsystem ID 4char */
/*----------------------------------------------*/
/*---------------------------------------------*/
if ( ac > 1 ) {
    /* put DB2SSID in External var for access from managers */
    strncpy(DB2SSID, av[1], 4);
    DB2SSID[4] = NUL;
    printf("Managers %s can connect to DB2 %s
", SERVER, DB2SSID);
} else {
    printf("Managers %s cannot connect to DB2
", SERVER);
    return;
}
/* ------------------------------------------------*/
/* Setting the protocol to be used in this program */
/* ------------------------------------------------*/
numthrds = 2;
numqueue = 2;
#endif
#ifdef EP
    /* register with DCE/RUNTIME protocol + EP */
    rpc_server_use_protseq_ep("ncadg_ip_udp", numqueue, "1070", &st);
#endif
/* register with DCE/RUNTIME protocol */
rpc_server_use_protseq("ncadg_ip_udp", numqueue, &st);
#endif
Figure 159 (Part 2 of 8). Dynamic SQL Server
if (st != error_status_ok) {
    dce_error_inq_text(st, err_str, &est);
    printf("( %s )--->Cannot use protocol sequence - %s\n", SERVER, err_str);
} else {
    printf("( %s )Use protocol sequence OK - \n", SERVER);
}
fflush(stdout);

/* ------------------------------------*/
/* Register interface with RPC runtime */
/* ------------------------------------*/
rpc_server_register_if(SQLDSV_v1_0_s_ifspec, NULL, NULL, &st);
    /*(rpc_mgr_epv_t) &IFMGR ,&st);*/
if (st != error_status_ok) {
    dce_error_inq_text(st, err_str, &est);
    printf("( %s )--->Cannot register Interface - %s\n", SERVER, err_str);
} else {
    printf("( %s )Register interface OK - \n", SERVER);
}
fflush(stdout);

Figure 159 (Part 3 of 8). Dynamic SQL Server
/* Get binding handles from the runtime */
rpc_server_inq_bindings(&bvec, &st);

if (st != error_status_ok) {
    dce_error_inq_text(st,err_str,&est);
    printf("%s -->Cannot inq bindings - %s\n",SERVER,err_str);
    fflush(stdout);
}

/* Display the bindings on the console */

printf("%sBindings:\n",SERVER);
fflush(stdout);
for (i=0; i< bvec->count; i++)
{
    rpc_binding_to_string_binding(bvec->binding_h[i],&string binding,&st);
    if (st != error_status_ok) {
        dce_error_inq_text(st,err_str,&est);
        printf("%s -->Cannot convert bindings - %s\n",SERVER,err_str);
        fflush(stdout);
    }
    printf("%s\n", (char *)string binding);
    fflush(stdout);
    rpc_string_free(&string binding,&st);
    if (st != error_status_ok) {
        dce_error_inq_text(st,err_str,&est);
        printf("%s -->Cannot free mem - %s\n",SERVER,err_str);
        fflush(stdout);
    }
}
```c
#ifdef RPCD
    /* -------------------------------------------- */
    /* register dynamically EP with endpoint Mapper */
    /* -------------------------------------------- */

    rpc_ep_register

    #ifdef SECCDS
        ( SQLDSV_v1_0_s_ifspec, bvec, NULL,"SEC-CDS",&st);
    #else
        ( SQLDSV_v1_0_s_ifspec, bvec, NULL,NULL , &st);
    #endif

    if (st != error_status_ok) {
        dce_error_inq_text(st,err_str,&est);
        printf( "%s
\n\n\n%*s -->Cannot register EP - %s
\n\n\n%*s SERVER,err_str);
    } else {
        printf("%s Register EP OK - \n\n\n%*s SERVER);
    }
    fflush(stdout);
#endif

#elifdef SECCDS
    /*-----------------------------------------*/
    /* SECURITY CALLS */
    /*-----------------------------------------*/
    /*-----------------------------------------*/
    /* Establish Identity with security server */
    /*- call goes to registry service (on Security server */
    /*- returns TGT (for priviledge services) and ConvKey */
    /*-----------------------------------------*/

    printf("%s Establishing Identity with security server...
\n\n\n%*s SERVER);kHzflush(stdout);

    sec_login_setup_identity(dce_login,sec_login_no_flags,&lcontext,&st);
    if (st != error_status_ok)
    {
        dce_error_inq_text(st,err_str,&est);
        printf("%s -->Cannot setup login-identity %s
\n\n\n%*s SERVER,err_str);
        fflush(stdout);
        exit(1);
    }

    pwrec.key.tagged_union.plain = "VANAER";
    pwrec.key.key_type = sec_passwd_plain;
    pwrec.pepper = NULL;
    pwrec.version_number = sec_passwd_c_version_none;
```

Figure 159 (Part 5 of 8). Dynamic SQL Server
/* Validation of Identity */
/*- call goes to LOCAL security runtime */
/*- runtime decrypts envelope with PASSWORD */

printf("Validation of identity..... \n",SERVER);
fflush(stdout);
sec_login_validate_identity(lcontext, &pwrec, &reset_passwd,
&auth_src,&st);
if (st != error_status_ok)
{
    dce_error_inq_text(st,err_str,&est);
    printf("-->Cannot validate login-identity %s\n",SERVER,err_str)
    fflush(stdout);
    exit(1);
}

/* Set login context */
/*- sets network credentials (in context) */
/*- this context will be assumed by processes by */
/* e.g. users, clients */

printf("Set login context......... \n",SERVER);
fflush(stdout);
sec_login_set_context(lcontext, &st);
if (st != error_status_ok)
{
    dce_error_inq_text(st,err_str,&est);
    printf("-->Cannot set login-context %d %s\n",SERVER,st,err_str)
    fflush(stdout);
    exit(1);
}

Figure 159 (Part 6 of 8). Dynamic SQL Server
/* Register authorization info */
/* registers authorization info with runtime */
/*-----------------------------------------*/
printf("( %s )Register Authorization info..... \n",SERVER);
fflush(stdout);
/* Registering server authentication */
/* rpc_c_authn_dce_secret (shared secret) */
rpc_server_register_auth_info(SERVER,
rpc_c_authn_none,NULL,NULL,&st);
if (st != error_status_ok)
{
    dce_error_inq_text(st,err_str,&est);
    printf("( %s )--Cannot register authorization- %s\n",SERVER,err_str)
    exit(1);
}

printf("( %s )Identity is established......... \n",SERVER);
fflush(stdout);

/*-----------------------------------------*/
/* DIRECTORY CALLS */
/*-----------------------------------------*/
/*----------------------------------------------------*/
/* Export binding to namespace */
/*----------------------------------------------------*/
printf("( %s )Export binding to namespace .... \n",SERVER);
fflush(stdout);
	rpc_ns_binding_export(rpc_c_ns_syntax_default,
	(unsigned_char_t *)&:/servers/SQLDSV
	SQLDSV_v1_0_s_ifspec,
bvec,(uuid_vector_t *)NULL, &st);
if (st != error_status_ok)
{
    dce_error_inq_text(st,err_str,&est);
    printf("( %s )--Cannot export binding - %s\n",SERVER,err_str);
    exit(1);
}
else {
    printf("( %s )Export binding to CDS OK - %s\n",SERVER,err_str);
}
fflush(stdout);
#endif

Figure 159 (Part 7 of 8). Dynamic SQL Server
/* At this point the server waits for clients */
TRY
{
    printf("( %s )listening...
",SERVER);
    fflush(stdout);
    rpc_server_listen(numthrds, &st);
}
CATCH_ALL
{
    printf("( %s )Unregistering interface SERVER 
",SERVER);
    fflush(stdout);
    rpc_server_unregister_if(SQLDSV_v1_0_s_ifspec, NULL,&st);
#ifdef RPCD
    printf("( %s )Unregistering endpoint 
",SERVER);
    fflush(stdout);
    rpc_ep_unregister(SQLDSV_v1_0_s_ifspec, bvec,NULL,&st);
#endif
#ifdef SECCDS
    printf("( %s )Unregistering from namespace 
",SERVER);
    fflush(stdout);
    rpc_ns_binding_unexport(rpc_c_ns_syntax_default,hostname, SQLDSV_v1_0_s_ifspec, NULL,&st);
#endif
/*-------------------------------*/
}
ENDTRY;
/*-------------------------------END SERVER2-------------------------------*/

Figure 159 (Part 8 of 8). Dynamic SQL Server
F.5 Manager Source Code

/* #include "myenv.h" */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#pragma linkage(DSNALI,OS)
#pragma linkage (DSNHLI,OS)
typedef struct
{ short SQLPLLEN;
  short SQLFLAGS;
  short SQLCTYPE;
  char SQLPROGN [ 8 ];
  short SQLTIMES [ 4 ];
  short SQLSECTN;
  char *SQLCODEP;
  char *SQLPARM;
  char *SQLAPARM;
  short SQLSTNUM;
  short SQLSTYPE;
} SQLPLIST;

typedef struct
{ short SQLTYPE;
  short SQLLEN;
  char *SQLADDR;
  char *SQLIND;
} SQLELTS;

typedef SQLELTS *SQLELTS_PTR;
char SQLTEMP [ 19 ];
/***$$
EXEC SQL INCLUDE SQLDA
$$$**/*/
#ifndef SQLDASIZE
struct sqlda
{ unsigned char sqldaid [ 8 ];
  long sqldabc;
  short sqln;
  short sqld;
  struct sqlvar
  { short sqltype;
    short sqllen;
    unsigned char *sqldata;
    short *sqlind;
    struct sqlname
    { short length;
      unsigned char data [ 30 ];
    } sqlname;
  } sqlvar [ 1 ];
}
#define SQLDASIZE(n) \    ( sizeof(struct sqlda) + ((n)-1) * sizeof(struct sqlvar) )
#endif

Figure 160 (Part 1 of 53). Dynamic SQL Server Manager
#define DCE
#define DEBUG1
#define DEBUG2
#ifdef DCE
#include <dce/rpc.h>
#include <dce/dce_error.h>
#endif
#include "SQLDSV.h"

/*---------defines for SQL ------------------------------------*/
#define T_DATE 384 /* length 10 character string */
#define TN_DATE 385 /* length 10 + NULLS character string */
#define T_TIME 388 /* length 8 character string */
#define TN_TIME 389 /* length 8 + NULLS character string */
#define T_TMSTMP 392 /* length 26 character string */
#define TN_TMSTMP 393 /* length 26 + NULLS character string */
#define T_VCHAR 448 /* varchar */
#define TN_VCHAR 449 /* varchar with NULL */
#define T_CHAR 452 /* char */
#define TN_CHAR 453 /* char with NULL */
#define T_LVCHAR 456 /* long varchar */
#define TN_LVCHAR 457 /* long varchar with NULL */
#define T_GVCHAR 464
#define TN_GVCHAR 465
#define T_GCHAR 468
#define TN_GCHAR 469
#define T_GLCHAR 472
#define TN_GLCHAR 473
#define T_FLOAT 480 /* length 4 of 8 */
#define TN_FLOAT 481 /* length 4 of 8 + NULLS */
#define T_DEC 484 /* length 1/2/3/4/5/6/7/8 */
#define TN_DEC 485 /* length 1/2/3/4/5/6/7/8 + NULLS */
#define T_INT 496 /* length 4 */
#define TN_INT 497 /* length 4 + NULLS */
#define T_SMINT 500 /* length 2 */
#define TN_SMINT 501 /* length 2 + NULLS */

Figure 160 (Part 2 of 53). Dynamic SQL Server Manager
/*-----------------------------------------------*/
/*-----------------------------------------------*/
#define SQLDA_FIX 16
#define SQLDA_VAR 44
#define BLANKS " "
#define FLOATLEN 4
#define DOUBLELEN 8
/*-----------------------------------------------*/
#define NO 0
#define YES 1
#define SEP1 
#define SEP2 
#define NUL 
#define RC_OK 0
#define RC_WARN 4
#define RC_ERROR 8
#define RC_SEVERE 12
#define RC_END_CSR 100
#define RC_ARITH_WEN 802
#define ZERO 0
#define NDYNAM 200
#define BUFFSIZE 4096
/*-----------------------------------------------*/
#define RC_INVALID -9999
#define RC_BUFF_OFLOW -9998
#define RC_TYPE_UNSUP -9997
#define RC_PROG_LIMITS -9996
/*-----------------------------------------------*/
typedef struct sqlda {
    unsigned char sqldaid [ 8 ];
    long sqldabc;
    short sqln;
    short sqld;
    struct sqlvar {
        short sqltype;
        short sqllen;
        unsigned char *sqldata;
        short *sqlind;
        struct sqlname {
            short length;
            unsigned char data [ 30 ];
        } sqlname;
    } sqlvar [ 10 ];
} SQLDA;
typedef struct sqlda SQLDA;

Figure 160 (Part 3 of 53). Dynamic SQL Server Manager
extern char DB2SSID [ 5 ] ;
long int SQL_SRV_EXEC ( handle_t bh,
    char *dbplname,
    char *sql_stmt,
    VLHandle  result,
    long int *pcond,
    char output [ 101 ],
    char operation [ 21 ],
    error_status_t *c_sts,
    error_status_t *f_sts)
{

    typedef struct ValueList *VLHandle;

    long retcode;
    long reascode;
    int rc;
    char function [ 9 ] ;
    char sprintf_buf [ 100 ] ;

    /*-------------------CAF variables -----------------------------*/
    char sync [ 5 ] = "SYNC";
    char abort [ 5 ] = "ABRT";
    long fnret;
    char openfn [ 13 ] = "OPEN ";
    char closefn [ 13 ] = "CLOSE ";
    char ssid [ 5 ] ;
    char planname [ 9 ] ;

    /*-------------------CAF variables -----------------------------*/
    *pcond = 0;
    printf("-> (SQLDSVMA) ==> New TRANSACTION <= 
      ");
    printf("-> (SQLDSVMA) ==> %s
      ", sql_stmt);
    fflush(stdout);
    printf("-> (SQLDSVMA) ==> requested %d rows flags = %d 
      ",
      result->nrows, result->flags);

Figure 160 (Part 4 of 53). Dynamic SQL Server Manager
/*-----------------------------------------------*/
/* Connect to DB2 via CAF attachment */
/*-----------------------------------------------*/
strcpy(planname, dbplname);
strcpy(ssid, DB2SSID);
reascode = 0;
retcode = 0;
printf("-> (SQLDSVMA) connecting via CAF plan %s to DB2 %s 
", planname, ssid);
fflush(stdout);
fnret = DSNALI(&openfn[0], &ssid[0], &planname[0],
             &retcode, &reascode);
printf("OPEN->DSNALI fnret %d retcode %9d reascode %08x %9d 
", fnret, retcode, reascode, reascode);
fflush(stdout);
if ((retcode == 0) & (fnret == 0)) {
    printf("CONNECT OK 
");
    fflush(stdout);
    rc = EXECSQL(sql_stmt, result, pcond, output, operation);
    /*-----------------------------------------------*/
#else DB2
    DSNALI(&closefn[0], &sync[0],
            &retcode, &reascode);
    printf("CLOS->SQLDSVMA %9d %08x %9d 
", retcode, reascode, reascode);
    fflush(stdout);
#endif
} else {
    strcpy(output,"SQLDSVMA");
    strcat(output," Plan OPEN error retc = ");
    sprintf(sprintf_buf,"%+07i",retcode);
    strcat(output,sprintf_buf,7);
    strcat(output," reasc = ");
    sprintf(sprintf_buf,"%+07i",reascode);
    strcat(output,sprintf_buf,7);
    printf("-> (SQLDSVMA) Na open PLAN in else 
");
    rc = fnret;
    *pcond = 8;
} return (rc);
*/

Figure 160 (Part 5 of 53). Dynamic SQL Server Manager
/*----------------------------------------------*/
/*----------------EXEC_SQL EXEC_SQL------------------------*/
/*----------------------------------------------*/

int EXECSQL(
    char *sql_stmt,
    VLHandle result,
    long int *pcond,
    char *output,
    char *operation)
{
    typedef struct ValueList {
        [ ptr,ignore ] ValueListExtHeader *header;
        unsigned short ncols;
        unsigned short nrows;
        unsigned short nrowsProcessed;
        unsigned short flags;
        [ ptr,size_is(ncols) ] unsigned short *colwidth;
        [ ptr,size_is(ncols) ] string *colLabel;
        [ ptr,size_is(ncols) ] ColDataType *coldata;
    } ValueList;

    typedef struct ValueListExtHeader {
        struct ValueListExtHeader *next;
        char blkType [ 4 ];
        unsigned short blkversion;
        unsigned short blksize;
    } ValueListExtHeader;

    EXEC SQL INCLUDE SQLCA

    ifndef SQLCODE
    struct sqlca
    {
        unsigned char sqlcaid [ 8 ];
        long sqlcabc;
        long sqlcode;
        short sqlerrml;
        unsigned char sqlerrmc [ 70 ];
        unsigned char sqlerrp [ 8 ];
        long sqlerrd [ 6 ];
        unsigned char sqlwarn [ 11 ];
        unsigned char sqlstate [ 5 ];
    }
    endif

    Figure 160 (Part 6 of 53). Dynamic SQL Server Manager
#define SQLCODE sqlca.sqlcode
#define SQLWARN0 sqlca.sqlwarn[0]
#define SQLWARN1 sqlca.sqlwarn[1]
#define SQLWARN2 sqlca.sqlwarn[2]
#define SQLWARN3 sqlca.sqlwarn[3]
#define SQLWARN4 sqlca.sqlwarn[4]
#define SQLWARN5 sqlca.sqlwarn[5]
#define SQLWARN6 sqlca.sqlwarn[6]
#define SQLWARN7 sqlca.sqlwarn[7]
#define SQLWARN8 sqlca.sqlwarn[8]
#define SQLWARN9 sqlca.sqlwarn[9]
#define SQLWARNA sqlca.sqlwarn[10]
#define SQLSTATE sqlca.sqlstate

struct sqlca sqlca;

/***$$ EXEC SQL INCLUDE SQLDA $$$***/
#ifndef SQLDASIZE
struct sqlda
{
  unsigned char sqldaid[8];
  long sqldabc;
  short sqln;
  short sqld;
  struct sqlvar
  {
    short sqltype;
    short sqllen;
    unsigned char *sqldata;
    short *sqlind;
    struct sqlname
    {
      short length;
      unsigned char data[30];
    } sqlname;
  } sqlvar[1];
};
#define SQLDASIZE(n)\( sizeof(struct sqlda) + ((n)-1) * sizeof(struct sqlvar) \)
#endif

struct sqlda *sqlda_ptr;
int size, nsieze, err, buff_size, sqlda_size;
short int recnt;
int i, j, k, curopen;
long reascode;
int open = 0;
short int sflags, ga_voort, ga_fetch, ga_buff;
long int rc;
char segment[21];
char function[9];
char sprintf_buf[100];
char my_sqlstate[6];
char *buff_ptr;

Figure 160 (Part 7 of 53). Dynamic SQL Server Manager
/*--------------------------------------*/
/* verify input for valid SQL statement */

strcpy(function,""");
sflags = result->flags;
/*--------------------------------------*/

strcpy(operation,"VERSQL");
rc = VerSql ( sql_stmt);

if (rc != 0 ) {
/* SQL STATEMENT is NOT VALID */
*pcond = 8;
strcpy(output,"SQL Statement is NOT VALID");
printf("\%s \%s \n",sql_stmt,output);
fflush(stdout);
return;
}

Figure 160 (Part 8 of 53). Dynamic SQL Server Manager
#ifdef DEBUG1
    printf("SQL_Statement Valid \n");
    fflush(stdout);
#endif
strcpy (stmt.buf, sql_stmt);
stmt.len = strlen(stmt.buf);

/*--------------------------------------*/
/* allocate SQLDA */
strcpy(operation,"ALLOCATALOGDA");
sqlda_ptr = NULL;
sqlda_size = 0;
rc = AllocSQLDA (&sqlda_ptr,&sqlda_size);
if (rc != 0 ) {
    *pcond = 8;
    strcpy(output,"CANNOT ALLOCATE SQLDA");
    printf("-> %s \n",output);
    fflush(stdout);
    return;
}
#endif DEBUG1
printf("SQLDA allocated \n");
flush(stdout);
#endif

/*--------------------------------------*/
/* allocate Buffer */
strcpy(operation,"ALLOCBUFFER");
buff_size = 0;
buff_ptr = NULL;
rc = AllocBuffer (&buff_ptr,&buff_size);
if (rc != 0 ) {
    *pcond = 8;
    strcpy(output,"CANNOT ALLOCATE BUFFER");
    printf("-> %s \n",output);
    flush(stdout);
    return;
}
#endif DEBUG1
printf("BUFFER allocated \n");
flush(stdout);
#endif
/* EXEC SQL WHENEVER SQLERROR GOTO SQLERR; */

/***$ *
 EXEC SQL WHENEVER SQLWARNING CONTINUE
 $$***/

/***$ *
 EXEC SQL WHENEVER NOT FOUND CONTINUE
 $$***/

/* prepare input statement */
strcpy(function,"PREPARE");
#ifdef DEBUG1
 printf("PREPARE->SGMDNSQL %s\n". stmt.buf);
 fflush(stdout);
#endif

/***$ *
 EXEC SQL PREPARE STATEMENT FROM :stmt
 $$***/
{
 SQLPLIST SQLLIST3 =
 {40, 8192, 35, "SQLDSVMA", 0, 0, 0, 0, 1, 0, 0, 316, 14};
 SQLELTS_PTR SQLELTS_PTR3;
 SQLLIST3.SQLVPARM = (char *) &stmt;
 SQLLIST3.SQLCODEP = (char *) &sqlca;
 SQLLIST3.SQLTIMES [ 0 ] = 0x153E;
 SQLLIST3.SQLTIMES [ 1 ] = 0x37B2;
 SQLLIST3.SQLTIMES [ 2 ] = 0x00C2;
 SQLLIST3.SQLTIMES [ 3 ] = 0x19E8;
 DSNHLI ( (unsigned int *) &SQLLIST3);
}

Figure 160 (Part 10 of 53). Dynamic SQL Server Manager
if ((SQLCODE > RC_OK) || (SQLCODE < RC_OK)) {
    printf("Prepare SQLCODE %d \n", SQLCODE);
    sqlCAP(SQLCODE, function, output);
    strcpy(operation, function);
    *pcond = 8;
    return;
}

/* describe input statement */
strcpy(function, "DESCRIBE");
printf("DESCRIBE->SGMDNSQL \n");
fflush(stdout);

/***$$
  EXEC SQL DESCRIBE STATEMENT INTO :sqlda_ptr
$$***/
{
    SQLPLIST SQLPLIST4 =
{40, 1024, 40, "SQLDSVMA", 0, 0, 0 ,0,
  1, 0, 0, 0, 333, 17};
    SQLELTS_PTR SQLELTS_PTR4;
    SQLPLIST4.SQLAPARM = (char *) sqlda_ptr;
    SQLPLIST4.SQLCODEP = (char *) &sqlca;
    SQLPLIST4.SQLTIMES [ 0 ] = 0x153E;
    SQLPLIST4.SQLTIMES [ 1 ] = 0x37B2;
    SQLPLIST4.SQLTIMES [ 2 ] = 0x00C2;
    SQLPLIST4.SQLTIMES [ 3 ] = 0x19E8;
    DSNHLI ( (unsigned int * ) &SQLPLIST4);
}

if ((SQLCODE > RC_OK) || (SQLCODE < RC_OK)) {
    sqlCAP(SQLCODE, function, output);
    strcpy(operation, function);
    *pcond = 8;
    return;
}

ga_voort = YES;
#ifdef DEBUG1
    printf("WHILE 1->SGMDNSQL \n");
#endif

Figure 160 (Part 11 of 53). Dynamic SQL Server Manager
while (ga_voort) {
    /*--------------------------------------------------------------*/
    /* look for value in SQLD in order to process stmt correctly */
    if (sqlda_ptr->sqld == ZERO) { /* if Z */
    /* statement is not a SELECT statement */
        strcpy(function, "EXECUTE");
    
    #ifdef DEBUG1
        printf("EXECUTE->SGMDNSQL 
");
        fflush(stdout);
    #endif
    
    /***
    EXEC SQL EXECUTE STATEMENT
    *
    ***/
    {  
        SQLPLIST SQLPLIST5 = 
            {40, 0, 30, "SQLDSVMA", 0, 0, 0 ,0,
             1, 0, 0, 0, 357, 15};
        SQLELTS_PTR SQLELTS_PTR5;
        SQLPLIST5.SQLCODEP = (char *) &sqlca;
        SQLPLIST5.SQLTIMES[0] = 0x153E;
        SQLPLIST5.SQLTIMES[1] = 0x37B2;
        SQLPLIST5.SQLTIMES[2] = 0x00C2;
        SQLPLIST5.SQLTIMES[3] = 0x19E8;
        DSNHLI ( (unsigned int *) &SQLPLIST5);
    }

    if ((SQLCODE > RC_OK) || (SQLCODE < 0)) {
        sqlCAP(SQLCODE, function, output);
        strcpy(operation, function);
        *pcond = 8;
    } else *pcond = 0;
    ga_voort = NO;
    }

    /* statement was a SELECT statement and is described */
    else if (sqlda_ptr->sqld <= sqlda_ptr->sqln) {
    /* SQLDA zone was big enough for description */
    
    #ifdef DEBUG1
        printf("SetupOut->SGMDNSQL \n");
        fflush(stdout);
    #endif
}

Figure 160 (Part 12 of 53). Dynamic SQL Server Manager
ga_buff = YES;
while (ga_buff) {
    rc = SetupOut
        (sqlda_ptr, buff_ptr ,buff_size, result, sflags);

    if ( rc == 8 ) { /* should enlarge Buffer */
        rc = AllocBuffer (&buff_ptr,&buff_size);
    } else ga_buff = NO;
}

#ifdef DEBUG1
printf("DECLARE->SGMDNSQL 
");
fflush(stdout);
#endif
strcpy(function,"DECLARE");

/***$$
EXEC SQL DECLARE C1 CURSOR FOR STATEMENT
$$$$**/

if ((SQLCODE > RC_OK) || (SQLCODE < 0)) {
    sqlCAP(SQLCODE, function, output);
    strcpy(operation, function);
    *pcond = 8;
    return(8);
}

#ifdef DEBUG1
printf("OPEN ->SGMDNSQL 
");
fflush(stdout);
#endif
strcpy(function,"OPEN");
EXEC SQL OPEN C1

{ SQLPLIST SQLPLIST7 =
  {40, 0, 50, "SQLDSVMA", 0, 0, 0, 0, 1, 0, 0, 0, 403, 3};
  SQLELTS_PTR SQLELTS_PTR7;
  SQLPLIST7.SQLCODEP = (char *) &sqlca;
  SQLPLIST7.SQLTIMES [ 0 ] = 0x153E;
  SQLPLIST7.SQLTIMES [ 1 ] = 0x37B2;
  SQLPLIST7.SQLTIMES [ 2 ] = 0x00C2;
  SQLPLIST7.SQLTIMES [ 3 ] = 0x19E8;
  DSNHLI ( (unsigned int * ) &SQLPLIST7);
}

if ((SQLCODE > RC_OK) || (SQLCODE < 0)) {
  sqlCAP(SQLCODE, function, output);
  strcpy(operation, function);
  *pcond = 8;
  return;
} else curopen = YES;

strcpy(function,"FETCH");

/* FETCH all rows until SQLCODE not is NULL */
do {
  #ifdef DEBUG1
    printf("FETCH ->SGMDNSQL \n");
    fflush(stdout);
  #endif

Figure 160 (Part 14 of 53). Dynamic SQL Server Manager
EXEC SQL FETCH C1 USING DESCRIPTOR :*sqlda_ptr

```c
SQLPLIST SQLPLIST8 =
{40, 1024, 30, "SQLDSVMA", 0, 0, 0,
  1, 0, 0, 0, 422, 4};
SQLELTS_PTR SQLELTS_PTR8;
SQLPLIST8.SQLAPARM = (char *) sqlda_ptr;
SQLPLIST8.SQLCODEP = (char *) &sqlca;
SQLPLIST8.SQLTIMES[0] = 0x153E;
SQLPLIST8.SQLTIMES[1] = 0x37B2;
SQLPLIST8.SQLTIMES[2] = 0x00C2;
SQLPLIST8.SQLTIMES[3] = 0x19E8;
DSNHLI ((unsigned int *) &SQLPLIST8);
}
if (SQLCODE == ZERO) {
++recnt;
#ifdef DEBUG1
  printf("ConRow ->SGMDNSQL %d \n", recnt);
  fflush(stdout);
#endif
  rc = ConRow( sqlda_ptr, buff_ptr,
               result, recnt,sflags);
}
if (recnt >= result->nrows) ga_fetch = NO;
} while ((SQLCODE == ZERO) & (ga_fetch == YES));
gavvoort = NO;
if ( SQLCODE == 100 ) *pcond = 0;
else *pcond = SQLCODE;
result->nrowsProcessed = recnt;
strcpy(function,"CLOSE");
```

Figure 160 (Part 15 of 53). Dynamic SQL Server Manager
EXEC SQL CLOSE C1

{$$$**$
SQLPLIST SQLPLIST9 =
{40, 0, 45, “SQLDSVMA”. 0, 0, 0, 0,
1, 0, 0, 0, 440, 5};
SQLELTS_PTR SQLELTS_PTR9;
SQLPLIST9.SQLCODEP = (char *) &sqlca;
SQLPLIST9.SQLTIMES [ 0 ] = 0x153E;
SQLPLIST9.SQLTIMES [ 1 ] = 0x37B2;
SQLPLIST9.SQLTIMES [ 2 ] = 0x00C2;
SQLPLIST9.SQLTIMES [ 3 ] = 0x19E8;
DSNHLI ( (unsigned int *) &SQLPLIST9);
}

/* complete valuelist */
rc = ComVal( result);
}

/* SQLDA zone should be enlarged */
else {
rc = AllocSQLDA (&sqlda_ptr,&sqlda_size);
if (rc != 0 ) {
*pcond = 8;
strcpy(output,“CAN NOT ALLOCATE BUFFER”);
printf(“-> %s \n”,output);
fflush(stdout);
return;
}
}

g_a_voort = YES;
} /* endif Z */
} /* end DO While ga_voort */
#endif DEBUG1
printf(“-> (SGMDNSQL) Leaving \n”);
fflush(stdout);
#endif
rc = 0;
return(rc);
} /* FINAL BRACKET OF EXEC_SQL */
int AllocSQLDA ( SQLDA **sqlda_ptr_p, int *sqlda_size_ptr)
{
    SQLDA *old_sqlda_ptr;
    SQLDA *new_sqlda_ptr;
    int retc, size, old_size;
    retc = 0;
    if (*sqlda_ptr_p == NULL) {
        old_size = 0;
        size = SQLDA_FIX + (NDYNAM * SQLDA_VAR);
        #ifdef DCE
        #ifdef DEBUG1
        printf("\n->(SGMDNSQL)allocating %d for DCE SQLDA\n", size);
        fflush(stdout);
        #endif
        *sqlda_ptr_p = (SQLDA *)rpc_ss_allocate((unsigned)size);
        #else
        #ifdef DEBUG1
        printf("\n->(SGMDNSQL)allocating %d for non DCE SQLDA\n", size);
        fflush(stdout);
        #endif
        *sqlda_ptr_p = (SQLDA *)malloc((unsigned)size);
        #endif
        new_sqlda_ptr = *sqlda_ptr_p;
        new_sqlda_ptr->sqln = NDYNAM;
    } else {
        old_sqlda_ptr = *sqlda_ptr_p;
        old_size = old_sqlda_ptr->sqln;
        size = *sqlda_size_ptr + (NDYNAM * SQLDA_VAR);
        #ifdef DCE
        #ifdef DEBUG1
        printf("\n->(SGMDNSQL)deleting old SQLDA for DCE \n");
        #endif
        rpc_ss_free(old_sqlda_ptr);
        #ifdef DEBUG1
        printf("\n->(SGMDNSQL)allocating %d for DCE SQLDA\n", size);
        fflush(stdout);
        #endif
        *sqlda_ptr_p = (SQLDA *)rpc_ss_allocate((unsigned)size);
        #else
        #ifdef DEBUG1
        printf("\n->(SGMDNSQL)deleting old SQLDA for non DCE \n");
        #endif
        free(old_sqlda_ptr);
        #ifdef DEBUG1
        printf("\n->(SGMDNSQL)allocating %d for non DCE SQLDA\n", size);
        fflush(stdout);
        #endif
        *sqlda_ptr_p = (SQLDA *)malloc((unsigned)size);
        #endif
        new_sqlda_ptr = *sqlda_ptr_p;
        new_sqlda_ptr->sqln = old_size + NDYNAM;
    }
    *sqlda_size_ptr = size;
    return(retc); }

Figure 160 (Part 17 of 53). Dynamic SQL Server Manager
/**************************************************************************/
int AllocBuffer( char **buff_ptr, int *buff_size_ptr)
{
    char  *old_buff_ptr;
    int    retc;
    int    size;
    retc = 0;
    if (*buff_ptr == NULL) {
        size = BUFFSIZE;
#ifndef DCE
#endif
#ifdef DEBUG1
    printf((SGMDNSQL)allocating %d for DCE Buffer
            ,size);
    fflush(stdout);
#else
#endif
    *buff_ptr = (char *)rpc_ss_allocate((unsigned)size);
#endif
#else
#ifndef DCE
#endif
#ifdef DEBUG1
    printf((SGMDNSQL)allocating %d for non DCE Buffer
            ,size);
    fflush(stdout);
#else
#endif
    *buff_ptr = (char *)malloc((unsigned)size);
#endif
}
else {
    old_buff_ptr = *buff_ptr;
    size = *buff_size_ptr + BUFFSIZE;
#ifndef DCE
#endif
#ifdef DEBUG1
    printf((SGMDNSQL)allocating %d for DCE Buffer
            ,size);
    fflush(stdout);
#else
#endif
    *buff_ptr = (char *)rpc_ss_allocate((unsigned)size);
#endif
} else {
    old_buff_ptr = *buff_ptr;
    size = *buff_size_ptr + BUFFSIZE;
#ifndef DCE
#endif
#ifdef DEBUG1
    printf((SGMDNSQL)deleting old Buffer for DCE \n”);
#else
#endif
    rpc_ss_free(old_buff_ptr);
#ifndef DCE
#endif
#ifdef DEBUG1
    printf((SGMDNSQL)allocating %d for DCE Buffer\n ”,size);
    fflush(stdout);
#else
#endif
    *buff_ptr = (char *)rpc_ss_allocate((unsigned)size);
#endif
#else
#ifndef DCE
#endif
#ifdef DEBUG1
    printf((SGMDNSQL)deleting old Buffer for non DCE \n”);
#else
#endif
    free(old_buff_ptr);
#ifndef DCE
#endif
#ifdef DEBUG1
    printf((SGMDNSQL)allocating %d for non DCE Buffer\n ”,size);
    fflush(stdout);
#else
#endif
    *buff_ptr = (char *)malloc((unsigned)size);
#endif
}/*-------------------------------*/
Figure 160 (Part 18 of 53). Dynamic SQL Server Manager
/*-----------------------------------------------*/
/*/SETUPOUT SETUPOUT SETUPOUT------------------*/
/*/-----------------------------------------------*/
int SetupOut
 (SQLDA *sqlda_ptr, char *buffer_ptr, short int buff_size,
  VHandle result, short int flags)
{
  int retc;
  short int nulls;
  short int lensql, new_lensql;
  short int old_type;
  short i, j, k;
  int x;
  short ncols;
  short nrows;
  short nbytes;
  short scale, precision;
  short set_sql;
  short dataoff;
  char columnname [ 31 ] ;
  short labellen;
  char *data_ptr;
  char *hdata_ptr;
  ColDataType *coldata_ptr;
  /*-----------------------------*/
  result->flags = flags;
  columnname [ 30 ] = NUL;
  retc = 0;
  dataoff = 0;
  data_ptr = buffer_ptr;
  result->ncols = sqlda_ptr->sqlid;
  ncols = sqlda_ptr->sqlid;
  nrows = result->nrows;

Figure 160 (Part 19 of 53). Dynamic SQL Server Manager
/*-----------------------------*/
/* allocate Valuelist        */
/*-----------------------------*/
#endif DEBUG1
    printf("in SetupOut->SGMDNSQL nrows = %d ncols = %d flags = %d\n", 
nrows, ncols, flags );
    fflush(stdout);
#endif DEBUG1
    result->colWidth =
#endif DCE
    (idl_ushort_int *)rpc_ss_allocate(sizeof(idl_ushort_int) * ncols);
#else
    (idl_ushort_int *)malloc(sizeof(idl_ushort_int) * ncols);
#endif
#endif DEBUG1
    printf("Allocating colWidth for %d columns size %d\n", 
    ncols, 
    (sizeof(idl_ushort_int) * ncols));
    fflush(stdout);
#endif DEBUG1
    result->colLabel =
#endif DCE
    (string *)rpc_ss_allocate(sizeof(string) * ncols);
#else
    (string *)malloc(sizeof(string) * ncols);
#endif

Figure 160 (Part 20 of 53). Dynamic SQL Server Manager
```c
#ifdef DEBUG1
    printf("Allocating colLabel_ptr for %d columns size %d\n", ncols, (sizeof(string) * ncols));
    fflush(stdout);
#endif
else {
    result->colLabel = NULL;
} /* endif SHIPLABELS */
/* allocate for ColDatatypes */
/*------------------------------*/
result->colData =
#ifdef DCE
    (ColDataType *)rpc_ss_allocate(sizeof(ColDataType) * ncols);
#else
    (ColDataType *)malloc(sizeof(ColDataType) * ncols);
#endif
#ifdef DEBUG1
    printf("Allocating colDataType for %d columns size %d\n", ncols, (sizeof(ColDataType) * ncols));
    printf("All allocation Done \n");
    fflush(stdout);
#endif
for ( j = 0; j < ncols ; j++ ) {
    coldata_ptr = result->colData + j;
    /* printf("Print colprt2 %d \n",coldata_ptr );
    fflush(stdout); */
    lensql = sqlda_ptr->sqlvar [ j ] .sqllen;
    set_sql = 1;
    #ifdef DEBUG2
        printf("Print J= %d length = %d\n",j,lensql);
        fflush(stdout);
    #endif
    labellen = sqlda_ptr->sqlvar [ j ] .sqlname.length;
    memcpy(columname, sqlda_ptr->sqlvar [ j ] .sqlname.data, labellen);
    columname [ labellen ] = NUL;
    #ifdef DEBUG2
        printf("Column %d Name %s Type %d Length %d \n", j, columname, sqlda_ptr->sqlvar [ j ] .sqltype, lensql);
        fflush(stdout);
    #endif
    old_type = sqlda_ptr->sqlvar [ j ] .sqltype;
    new_lensql = lensql;
    old_type = sqlda_ptr->sqlvar [ j ] .sqltype;
    new_lensql = lensql;
```

Figure 160 (Part 21 of 53). Dynamic SQL Server Manager
while (1) {
    switch(sqlda_ptr->sqlvar[j].sqltype) {
    case T_INT:
        nulls = 0;
        coldata_ptr->colDataType = CDT_integer;
        coldata_ptr->ColDataUnion.integerCol.nrows = 0;
        coldata_ptr->ColDataUnion.integerCol.data =
#ifdef DCE
        (idl_long_int *)rpc_ss_allocate(sizeof(idl_long_int) * nrows);
#else
        (idl_long_int *)malloc(sizeof(idl_long_int) * nrows);
#endif
        coldata_ptr->ColDataUnion.integerCol.indicator = NULL;
        break;
    case T_SMINT:
        nulls = 0;
        coldata_ptr->colDataType = CDT_smallint;
        coldata_ptr->ColDataUnion.smallintCol.nrows = 0;
        coldata_ptr->ColDataUnion.smallintCol.data =
#ifdef DCE
        (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
#else
        (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
#endif
        coldata_ptr->ColDataUnion.smallintCol.indicator = NULL;
        break;
    case T_FLOAT:
        /* distinguish between FLOAT and DOUBLE */
        nulls = 0;
        fflush(stdout);
        if (lensql > FLOATLEN) {
            coldata_ptr->colDataType = CDT_double;
            coldata_ptr->ColDataUnion.doubleCol.nrows = 0;
            coldata_ptr->ColDataUnion.doubleCol.data =
#ifdef DCE
            (idl_long_float *)rpc_ss_allocate(sizeof(idl_long_float) * nrows);
#else
            (idl_long_float *)malloc(sizeof(idl_long_float) * nrows);
#endif
            coldata_ptr->ColDataUnion.doubleCol.indicator = NULL;
        } else {
            coldata_ptr->colDataType = CDT_float;
            coldata_ptr->ColDataUnion.floatCol.nrows = 0;
            coldata_ptr->ColDataUnion.floatCol.data =
#ifdef DCE
            (idl_short_float *)rpc_ss_allocate(sizeof(idl_short_float) * nrows);
#else
            (idl_short_float *)malloc(sizeof(idl_short_float) * nrows);
#endif
            coldata_ptr->ColDataUnion.floatCol.indicator = NULL;
        } break;
    }
}

Figure 160 (Part 22 of 53). Dynamic SQL Server Manager
case T_CHAR:
    nulls = 0;
    new_lensql = lensql + 1;
    if (lensql > 8 ) { /* string strategy */
        coldata_ptr->colDataType = CDT_string;
        coldata_ptr->ColDataUnion.stringCol.nrows = 0;
        coldata_ptr->ColDataUnion.stringCol.data =
        #ifdef DCE
          (string *)rpc_ss_allocate(sizeof(string) * nrows);
        #else
          (string *)malloc(sizeof(string) * nrows);
        #endif
        #ifdef DEBUG2
          printf("Setup allocated %d storage for T_STRING \n",
                 (sizeof(string) * nrows));
        #endif
        coldata_ptr->ColDataUnion.stringCol.indicator = NULL;
    } else { /* char strategy */
        nbytes = nrows * new_lensql;
        coldata_ptr->colDataType = CDT_char;
        coldata_ptr->ColDataUnion.charCol.nrows = 0;
        coldata_ptr->ColDataUnion.charCol.nbytes = 0;
        coldata_ptr->ColDataUnion.charCol.data =
        #ifdef DCE
          (char *)rpc_ss_allocate(sizeof(char) * nbytes);
          /* robo */
        #else
          (char *)malloc(sizeof(char) * nbytes);
        #endif
        coldata_ptr->ColDataUnion.charCol.indicator = NULL;
        #ifdef DEBUG2
          printf("Setup allocated %d storage for T_CHAR \n",
                 (sizeof(char) * nbytes));
        #endif
    }
    break;

case T_DEC:
    nulls = 0;
    scale = lensql / 256;
    precision = lensql - ( scale * 256);

Figure 160 (Part 23 of 53). Dynamic SQL Server Manager
```c
#ifdef DB2
#ifdef DEBUG2
printf("T_DEC/DB2 precision = %d scale = %d \n", precision, scale);
fflush(stdout);
#endif
if (( precision < 5 ) & (scale == 0)) {
#ifdef DEBUG2
printf("T_DEC small int \n");
fflush(stdout);
#endif
/* use small integer */
new_lensql= 2;
sqlda_ptr->sqlvar[j].sqltype = T_SMINT;
coldata_ptr->colDataType = CDT_smallint;
coldata_ptr->ColDataUnion.smallintCol.nrows = 0;
coldata_ptr->ColDataUnion.smallintCol.data =
#endif
#endif
/* use large integer */
new_lensql= 4;
sqlda_ptr->sqlvar[j].sqltype = T_INT;
coldata_ptr->colDataType = CDT_integer;
coldata_ptr->ColDataUnion.integerCol.nrows = 0;
coldata_ptr->ColDataUnion.integerCol.data =
#endif
#endif
#endif
```

Figure 160 (Part 24 of 53). Dynamic SQL Server Manager
else if (precision < 7) {
    /* use float */
    new_lensql = 4;
    sqlda_ptr->sqlvar[j].sqltype = T_FLOAT;
    coldata_ptr->colDataType = CDT_float;
    coldata_ptr->ColDataUnion.floatCol.nrows = 0;
    coldata_ptr->ColDataUnion.floatCol.data =
#ifdef DCE
    (idl_short_float *)rpc_ss_allocate(sizeof(idl_short_float) * nrows);
#else
    (idl_short_float *)malloc(sizeof(idl_short_float) * nrows);
#endif
    coldata_ptr->ColDataUnion.floatCol.indicator = NULL;
} else {
    /* use double */
    new_lensql = 8;
    sqlda_ptr->sqlvar[j].sqltype = T_FLOAT;
    coldata_ptr->colDataType = CDT_double;
    coldata_ptr->ColDataUnion.doubleCol.nrows = 0;
    coldata_ptr->ColDataUnion.doubleCol.data =
#ifdef DCE
    (idl_long_float *)rpc_ss_allocate(sizeof(idl_long_float) * nrows);
#else
    (idl_long_float *)malloc(sizeof(idl_long_float) * nrows);
#endif
    coldata_ptr->ColDataUnion.doubleCol.indicator = NULL;
}
else {
    sqlda_ptr->sqlvar[j].sqllen = new_lensql;
#endif
#ifdef DEBUG2
    printf("T_DEC/DB2/2 precision = %d scale = %d \n", precision, scale);
    fflush(stdout);
#endif
    new_lensql = precision + 1;
nbytes = nrows * new_lensql;
    coldata_ptr->colDataType = CDT_packed;
    coldata_ptr->ColDataUnion.packedCol.nrows = 0;
    coldata_ptr->ColDataUnion.packedCol.nbytes = 0;
    coldata_ptr->ColDataUnion.packedCol.data =
#ifdef DCE
    (char *)rpc_ss_allocate(sizeof(char) * nbytes);
#else
    (char *)malloc(sizeof(char) * nbytes);
#endif
    coldata_ptr->ColDataUnion.packedCol.indicator = NULL;
#ifendif
break;

Figure 160 (Part 25 of 53). Dynamic SQL Server Manager
case T_VCHAR:
    /* VCHAR field 1 <= 254 */
    nulls = 0;
    new_lensql = lensql + 3; /* 2 bytes for varchar length */
    coldata_ptr->colDataType = CDT_string;
    coldata_ptr->ColDataUnion.stringCol.nrows = 0;
    coldata_ptr->ColDataUnion.stringCol.data =
    #ifdef DCE
        (string *)rpc_ss_allocate(sizeof(string) * nrows);
    #else
        (string *)malloc(sizeof(string) * nrows);
    #endif
    #ifdef DEBUG2
        printf("Setup allocated %d storage for T_VCHAR \n",
               (sizeof(string) * nrows));
    #endif
    coldata_ptr->ColDataUnion.stringCol.indicator = NULL;
    break;

case T_LVCHAR:
    /* LVCHAR field > 254 */
    nulls = 0;
    new_lensql = lensql + 3; /* 2 bytes for varchar length */
    coldata_ptr->colDataType = CDT_string;
    coldata_ptr->ColDataUnion.stringCol.nrows = 0;
    coldata_ptr->ColDataUnion.stringCol.data =
    #ifdef DCE
        (string *)rpc_ss_allocate(sizeof(string) * nrows);
    #else
        (string *)malloc(sizeof(string) * nrows);
    #endif
    #ifdef DEBUG2
        printf("Setup allocated %d storage for T_LVCHAR \n",
               (sizeof(string) * nrows));
    #endif
    coldata_ptr->ColDataUnion.stringCol.indicator = NULL;

Figure 160 (Part 26 of 53). Dynamic SQL Server Manager
/* allocate storage for long varchar to hold data */

hdata_ptr =

#ifdef DCE
  (char *)rpc_ss_allocate(new_lensql);
#else
  (char *)malloc(new_lensql);
#endif

sqlda_ptr->sqlvar [ j ].sqlind = NULL;
sqlda_ptr->sqlvar [ j ].sqldata = (hdata_ptr + nulls);
set_sql = 0;
break;
case T_QVCHAR:
  break;
case T_GCHAR:
  break;
case T_DATE:
  nulls = 0;
  new_lensql = lensql + 1;
  nbytes = nrows * new_lensql;
coldata_ptr->colDataType = CDT_date;
coldata_ptr->ColDataUnion.dateCol.nrows = 0;
coldata_ptr->ColDataUnion.dateCol.nbytes = 0;
coldata_ptr->ColDataUnion.dateCol.data =

#ifdef DCE
  (char *)rpc_ss_allocate(nbytes);
#else
  (char *)malloc(nbytes);
#endif

coldata_ptr->ColDataUnion.dateCol.indicator = NULL;
break;
case T_TIME:
  nulls = 0;
  new_lensql = lensql + 1;
  nbytes = nrows * new_lensql;
coldata_ptr->colDataType = CDT_time;
coldata_ptr->ColDataUnion.timeCol.nrows = 0;
coldata_ptr->ColDataUnion.timeCol.nbytes = 0;
coldata_ptr->ColDataUnion.timeCol.data =

#ifdef DCE
  (char *)rpc_ss_allocate(nbytes);
#else
  (char *)malloc(nbytes);
#endif

Figure 160 (Part 27 of 53). Dynamic SQL Server Manager
coldata_ptr->ColDataUnion.timeCol.indicator = NULL;
/* sqlda_ptr->sqlvar [ j ] .sqlen = lensql */
break;
case T_TMSTMP:
    nulls = 0;
    new_lensql = lensql + 1;
    nbytes = nrows * new_lensql;
    coldata_ptr->colDataType = CDT_tmstmp;
    coldata_ptr->ColDataUnion.tmstmpCol.nrows = 0;
    coldata_ptr->ColDataUnion.tmstmpCol.nbytes = 0;
    coldata_ptr->ColDataUnion.tmstmpCol.data =
#endif DCE
    (char *)rpc_ss_allocate(nbytes);
#else
    (char *)malloc(nbytes);
#endif

coldata_ptr->ColDataUnion.tmstmpCol.indicator = NULL;
/* sqlda_ptr->sqlvar [ j ] .sqlen = lensql */
break;

/*----------------------------*/
/* FIELDS with NULL indicator */
/*----------------------------*/
case TN_INT:
    nulls = 2;
    coldata_ptr->colDataType = CDT_integer;
    coldata_ptr->ColDataUnion.integerCol.nrows = 0;
    coldata_ptr->ColDataUnion.integerCol.data =
#endif DCE
    (idl_long_int *)rpc_ss_allocate(sizeof(idl_long_int) * nrows);
#else
    (idl_long_int *)malloc(sizeof(idl_long_int) * nrows);
#endif

Figure 160 (Part 28 of 53). Dynamic SQL Server Manager
if (flags & CDF_shipIndicators)
    coldata_ptr->ColDataUnion.integerCol.indicator =
#ifdef DCE
    (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
#else
    (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
#endif
else
    coldata_ptr->ColDataUnion.integerCol.indicator = NULL;
break;
case TN_SMINT:
    nulls = 2;
coldata_ptr->colDataType = CDT_smallint;
coldata_ptr->ColDataUnion.smallintCol.nrows = 0;
coldata_ptr->ColDataUnion.smallintCol.data =
#ifdef DCE
    (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
#else
    (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
#endif
if (flags & CDF_shipIndicators)
    coldata_ptr->ColDataUnion.smallintCol.indicator =
#ifdef DCE
    (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
#else
    (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
#endif
else
    coldata_ptr->ColDataUnion.smallintCol.indicator = NULL;
break;
case TN_FLOAT:
    nulls = 2;
if (lensql > FLOATLEN) {
    coldata_ptr->colDataType = CDT_double;
coldata_ptr->ColDataUnion.doubleCol.nrows = 0;
coldata_ptr->ColDataUnion.doubleCol.data =
#ifdef DCE
    (idl_long_float *)rpc_ss_allocate(sizeof(idl_long_float) * nrows);
#else
    (idl_long_float *)malloc(sizeof(idl_long_float) * nrows);
#endif
}
if (flags & CDF_shipIndicators)
    coldata_ptr->ColDataUnion.doubleCol.indicator =
#ifdef DCE
    (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
#else
    (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
#endif
else
    coldata_ptr->ColDataUnion.doubleCol.indicator = NULL;
} else {
    coldata_ptr->colDataType = CDT_float;
    coldata_ptr->ColDataUnion.floatCol.nrows = 0;
    coldata_ptr->ColDataUnion.floatCol.data =
#ifdef DCE
    (idl_short_float *)rpc_ss_allocate(sizeof(idl_short_float) * nrows);
#else
    (idl_short_float *)malloc(sizeof(idl_short_float) * nrows);
#endif
if (flags & CDF_shipIndicators)
    coldata_ptr->ColDataUnion.floatCol.indicator =
#ifdef DCE
    (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
#else
    (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
#endif
else
    coldata_ptr->ColDataUnion.floatCol.indicator = NULL;
}
break;
case TN_CHAR:
    nulls = 2;
    new_lensql = lensql + 1;
    if (lensql > 8) {
        /* string strategy */
        coldata_ptr->colDataType = CDT_string;
        coldata_ptr->ColDataUnion.stringCol.nrows = 0;
        coldata_ptr->ColDataUnion.stringCol.data =
#ifdef DCE
        (string *)rpc_ss_allocate(sizeof(string) * nrows);
#else
        (string *)malloc(sizeof(string) * nrows);
#endif
    } else
        break;
Figure 160 (Part 30 of 53). Dynamic SQL Server Manager
#ifdef DEBUG2
    printf("Setup allocated %d storage for TN_STRING \n", 
        (sizeof(string) * nrows));
#endif
    if (flags & CDF_shipIndicators)
        coldata_ptr->ColDataUnion.stringCol.indicator =
#ifdef DCE
        (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
#else
        (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
#endif
    else
        coldata_ptr->ColDataUnion.stringCol.indicator = NULL;
#endif
    else
        coldata_ptr->ColDataUnion.stringCol.indicator = NULL;
    } else { /* char strategy */
    nbytes = nrows * new_lensql;
    coldata_ptr->colDataType = CDT_char;
    coldata_ptr->ColDataUnion.charCol.nrows = 0;
    coldata_ptr->ColDataUnion.charCol.nbytes = 0;
    coldata_ptr->ColDataUnion.charCol.data =
#ifdef DCE
        (char *)rpc_ss_allocate(sizeof(char) * nbytes);
#else
        (char *)malloc(sizeof(char) * nbytes);
#endif
#else DEBUG2
    printf("Setup allocated %d storage for TN_CHAR \n", 
        (sizeof(char) * nbytes));
#endif
    if (flags & CDF_shipIndicators)
        coldata_ptr->ColDataUnion.charCol.indicator =
#ifdef DCE
        (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
#else
        (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
#endif
    else
        coldata_ptr->ColDataUnion.charCol.indicator = NULL;
    }
/* sqlda_ptr->sqlvar [ j ].sqlen = lensql */
break;
case TN_DEC:
nulls = 2;
scale = lensql / 256;
precision = lensql - ( scale * 256);
#ifdef DB2
printf("TN_DEC/DB2 precision = %d scale = %d \n",precision,scale);
fflush(stdout);
if (( precision < 5 ) & (scale == 0)) {
    /* use small integer */
    new_lensql= 2;
sqlda_ptr->sqlvar [ j ].sqltype = T_SMINT;
coldata_ptr->colDataType = CDT_smallint;
coldata_ptr->ColDataUnion.smallintCol.nrows = 0;
coldata_ptr->ColDataUnion.smallintCol.data =
#endif
    (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
#else
    (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
#endif
if (flags & CDF_shipIndicators)
coldata_ptr->ColDataUnion.smallintCol.indicator =
#ifdef DCE
    (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
#else
    (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
#endif
else
coldata_ptr->ColDataUnion.integerCol.indicator = NULL;
} else if (( precision < 10 ) & (scale == 0)) {
    /* use large integer */
    new_lensql= 4;
sqlda_ptr->sqlvar [ j ].sqltype = T_INT;
coldata_ptr->colDataType = CDT_integer;
coldata_ptr->ColDataUnion.integerCol.nrows = 0;
coldata_ptr->ColDataUnion.integerCol.data =
#ifdef DCE
    (idl_long_int *)rpc_ss_allocate(sizeof(idl_long_int) * nrows);
#else
    (idl_long_int *)malloc(sizeof(idl_long_int) * nrows);
#endif

Figure 160 (Part 32 of 53). Dynamic SQL Server Manager
if (flags & CDF_shipIndicators)
  coldata_ptr->ColDataUnion.integerCol.indicator =
#else DCE
  (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
#else
  (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
#endif

else
  coldata_ptr->ColDataUnion.integerCol.indicator = NULL;
} else if ( precision < 7) {
  /* use float */
  new_lensql= 4;
  sqlda_ptr->sqlvar [ j ] .sqltype = T_FLOAT;
  coldata_ptr->colDataType = CDT_float;
  coldata_ptr->ColDataUnion.floatCol.nrows = 0;
  coldata_ptr->ColDataUnion.floatCol.data =
#endif DCE
  (idl_short_float *)rpc_ss_allocate(sizeof(idl_short_float) * nrows);
#else
  (idl_short_float *)malloc(sizeof(idl_short_float) * nrows);
#endif

if (flags & CDF_shipIndicators)
  coldata_ptr->ColDataUnion.floatCol.indicator =
#else DCE
  (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
#else
  (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
#endif

else
  coldata_ptr->ColDataUnion.floatCol.indicator = NULL;
} else {
  /* use double */
  new_lensql= 8;
  sqlda_ptr->sqlvar [ j ] .sqltype = T_FLOAT;
  coldata_ptr->colDataType = CDT_double;
  coldata_ptr->ColDataUnion.doubleCol.nrows = 0;
  coldata_ptr->ColDataUnion.doubleCol.data =

Figure 160 (Part 33 of 53). Dynamic SQL Server Manager
#ifdef DCE
(idl_long_float *)rpc_ss_allocate(sizeof(idl_long_float) * nrows);
#else
(idl_long_float *)malloc(sizeof(idl_long_float) * nrows);
#endif
    if (flags & CDF_shipIndicators)
        coldata_ptr->ColDataUnion.doubleCol.indicator =
    #ifdef DCE
        (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
    #else
        (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
    #endif
    else
        coldata_ptr->ColDataUnion.doubleCol.indicator = NULL;
    }
    sqlda_ptr->sqlvar [ j ] .sqllen = new_lensql;
#else
    #ifdef DEBUG2
    printf ("TN_DEC/DB2/2 precision = %d scale = %d \n", precision, scale);
    fflush(stdout);
    #endif
    new_lensql = precision + 1;
    nbytes = nrows * new_lensql;
    coldata_ptr->colDataType = CDT_packed;
    coldata_ptr->ColDataUnion.packedCol.nrows = 0;
    coldata_ptr->ColDataUnion.packedCol.nbytes = 0;
    coldata_ptr->ColDataUnion.packedCol.data =
    #ifdef DCE
        (char *)rpc_ss_allocate(sizeof(char) * nbytes);
    #else
        (char *)malloc(sizeof(char) * nbytes);
    #endif
    if (flags & CDF_shipIndicators)
        coldata_ptr->ColDataUnion.packedCol.indicator =
    #ifdef DCE
        (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
    #else
        (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
    #endif
    else
        coldata_ptr->ColDataUnion.packedCol.indicator = NULL;
    #endif
    break;

Figure 160 (Part 34 of 53). Dynamic SQL Server Manager
case TN_VCHAR:
    /* VCHAR field 1 <= 254 */
    nulls = 2;
    new_lensql= lensql + 3; /* 2 bytes for varchar length */
    coldata_ptr->colDataType = CDT_string;
    coldata_ptr->ColDataUnion.stringCol.nrows = 0;
    coldata_ptr->ColDataUnion.stringCol.data =
    #ifdef DCE
        (string *)rpc_ss_allocate(sizeof(string) * nrows);
    #else
        (string *)malloc(sizeof(string) * nrows);
    #endif
    #ifdef DEBUG2
        printf("Setup allocated %d storage for TN_VCHAR \n", (sizeof(string) * nrows));
    #endif
    if (flags & CDF_shipIndicators)
        coldata_ptr->ColDataUnion.stringCol.indicator =
    #ifdef DCE
        (idl_short_int *)rpc_ssAllocate(sizeof(idl_short_int) * nrows);
    #else
        (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
    #endif
    else
        (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
    #endif
    else
        coldata_ptr->ColDataUnion.stringCol.indicator = NULL;
        break;

    case TN_LVCHAR:
    /* LVCHAR field > 254 */
    nulls = 2;
    new_lensql= lensql + 3; /* 2 bytes for varchar length */
    coldata_ptr->colDataType = CDT_string;
    coldata_ptr->ColDataUnion.stringCol.nrows = 0;
    coldata_ptr->ColDataUnion.stringCol.data =
    #ifdef DCE
        (string *)rpc_ss_allocate(sizeof(string) * nrows);
    #else
        (string *)malloc(sizeof(string) * nrows);
    #endif
    #ifdef DEBUG2
        printf("Setup allocated %d storage for TN_LVCHAR \n", (sizeof(string) * nrows));
    #endif
    coldata_ptr->ColDataUnion.stringCol.indicator = NULL;
    /* allocate storage for lon varchar to get data */
    hdata_ptr =
    #ifdef DCE
        (char *)rpc_ss_allocate(new_lensql);
    #else
        (char *)malloc(new_lensql);
    #endif
if (flags & CDF_shipIndicators)
    coldata_ptr->ColDataUnion.stringCol.indicator =
#endif DCE
    (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
#else
    (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
#endif
else
    coldata_ptr->ColDataUnion.stringCol.indicator = NULL;
    sqlda_ptr->sqlvar[j].sqlind = (short *)hdata_ptr;
    sqlda_ptr->sqlvar[j].sqldata = (hdata_ptr + nulls);
    set_sql = 0;
    break;
    case TN_GVCHAR:
    break;
    case TN_GCHAR:
    break;
    case TN_DATE:
        nulls = 2;
        new_lensql = lensql + 1;
        nbytes = nrows * new_lensql;
        coldata_ptr->colDataType = CDT_date;
        coldata_ptr->ColDataUnion.dateCol.nrows = 0;
        coldata_ptr->ColDataUnion.dateCol.nbytes = 0;
        coldata_ptr->ColDataUnion.dateCol.data =
#endif DCE
        (char *)rpc_ss_allocate(nbytes);
#else
        (char *)malloc(nbytes);
#endif
if (flags & CDF_shipIndicators)
    coldata_ptr->ColDataUnion.dateCol.indicator =
#endif DCE
    (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
#else
    (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
#endif
else
    coldata_ptr->ColDataUnion.dateCol.indicator = NULL;
    break;
    case TN_TIME:
        nulls = 2;
        new_lensql = lensql + 1;
        nbytes = nrows * new_lensql;
        coldata_ptr->colDataType = CDT_time;
        coldata_ptr->ColDataUnion.timeCol.nrows = 0;
        coldata_ptr->ColDataUnion.timeCol.nbytes = 0;
        coldata_ptr->ColDataUnion.timeCol.data =
        (char *)rpc_ss_allocate(nbytes);
        (char *)malloc(nbytes);
#endif
        if (flags & CDF_shipIndicators)
            coldata_ptr->ColDataUnion.timeCol.indicator =
#endif DCE
            (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
#else
            (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
#endif
else
            coldata_ptr->ColDataUnion.timeCol.indicator = NULL;
            break;
#ifdef DCE
    (char *)rpc_ss_allocate(nbytes);
#else
    (char *)malloc(nbytes);
#endif
if (flags & CDF_shipIndicators)
coldata_ptr->ColDataUnion.timeCol.indicator =
#ifdef DCE
    (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
#else
    (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
#endif
#else
    coldata_ptr->ColDataUnion.timeCol.indicator = NULL;
/* sqlda_ptr->sqlvar [ j ] .sqllen = lensql; */
break;
case TN_TMSTMP:
    nulls = 2;
    new_lensql= lensql + 1;
    nbytes = nrows * new_lensql;
coldata_ptr->colDataType = CDT_tmstmp;
coldata_ptr->ColDataUnion.tmstmpCol.nrows = 0;
coldata_ptr->ColDataUnion.tmstmpCol.nbytes = 0;
coldata_ptr->ColDataUnion.tmstmpCol.data =
#ifdef DCE
    (char *)rpc_ss_allocate(nbytes);
#else
    (char *)malloc(nbytes);
#endif
if (flags & CDF_shipIndicators)
coldata_ptr->ColDataUnion.tmstmpCol.indicator =
#ifdef DCE
    (idl_short_int *)rpc_ss_allocate(sizeof(idl_short_int) * nrows);
#else
    (idl_short_int *)malloc(sizeof(idl_short_int) * nrows);
#endif
#else
    coldata_ptr->ColDataUnion.tmstmpCol.indicator = NULL;
/* sqlda_ptr->sqlvar [ j ] .sqllen = lensql; */
break;
default:
    printf(" invalid type \%d \n", sqlda_ptr->sqlvar [ j ] .sqltype);
    break;
} /* end switch */
#ifdef DEBUG2
    printf("While end \n");
    fflush(stdout);
#endif
break;
} /* end while */

Figure 160 (Part 37 of 53). Dynamic SQL Server Manager
if (set_sql) {
    dataoff = dataoff + new_lensql+ nulls;
    if ( dataoff > buff_size) {
        retc = 8;    /* Buffer is Too Small */
        return(retc);
    }
    if ( nulls > 0 )
        sqlda_ptr->sqlvar [ j ] .sqlind = (short *)data_ptr;
    else
        sqlda_ptr->sqlvar [ j ] .sqlind = NULL;
    sqlda_ptr->sqlvar [ j ] .sqldata = (data_ptr + nulls);
    memcpy(data_ptr,BLANKS,new_lensql+nulls);
    data_ptr = data_ptr + new_lensql+ nulls;
}
/*--------------------------------*/
result->colWidth [ j ] = new_lensql;
if (flags & CDF_shipLabels) {
    result->colLabel [ j ] =
#ifdef DCE
    (string)rpc_ss_allocate(labellen + 1);
#else
    (string)malloc(labellen + 1);
#endif
    strcpy(result->colLabel [ j ] ,
    #ifdef DEBUG2
    columnname);
    #ifdef DEBUG2
    printf("Label length %d %s \n", labellen, result->colLabel [ j ]);
    #endif
    }
    /*--------------------------------*/
#else DCE
    printf("BufferSize = %d Used = %d\n", buff_size, dataoff );
#endif
    retc = 0;
    return(retc);
} /* end SetupOut */

Figure 160 (Part 38 of 53). Dynamic SQL Server Manager
/*--------------------------------------------------------------*/
/*---------------------------SQLVER-------------------------------*/
/*-------------------------------------------------------------*/
int VerSql (char *stmt_ptr)
{
  int retc;
  short int stmt_len;
  short int i;
/---------------------------------------------*/
/* VALID SQL CALLS */
/---------------------------------------------*/
char valcmd [10] [9] = {
  "COMMIT", "DELETE", "EXPLAIN", "INSERT", 
  "LOCK", "ROLLBACK", "SELECT", "UPDATE", "CONNECT", "FETCH" 
};
stmt_len = strlen(stmt_ptr);
for (i = 0; i < 10; i++) {
  if (strstr(stmt_ptr, valcmd[i]) != NULL) {
    #ifdef DEBUG1
    printf("Valid function %s detected \n", valcmd[i]);
    #endif
    return(0);
  }
}
return(8);
} /* end SqlVer */
/-------------------------------------------------------------*/
/*---------------------------MakeDynamic------------------------*/
/*-------------------------------------------------------------*/
idl_char *MakeDynamic(char *s)
{
  char *p;
  #ifdef DCE
  p = (char *)rpc_ss_allocate(strlen(s) + 1);
  #else
  p = (char *)malloc(strlen(s) + 1);
  #endif
  strcpy(p, s);
  return(idl_char *)p;
} /* end MakeDynamic */

Figure 160 (Part 39 of 53). Dynamic SQL Server Manager
int ConRow(SQLDA *sqlda_ptr, char *buffer_ptr, VLHandle result, short int recnt, short int flags)
{
    int retc;
    int i, j, k, l;
    int npos;
    int ncols;
    unsigned short colwidth;
    string new_s_ptr;
    string cdt_ptr;
    ColDataType *coldata_ptr;
    char *data_ptr;
    char *nul_ptr;
    short int *ind_ptr;
    short int lensql;
    char *manynulls = "/0/0/0/0/0/0/0/0/0/0/0/0/0/0/0/0/0/0/0/0/0/0/0/0/0/0/0";

    ncols = result->ncols;
    --recnt;
    #ifdef DB2
    /* T_DEC has been converted to */
    /* T_INT T_SMINT T_FLOAT */
    #endif
    #ifdef DEBUG1
    printf("conrow handling %d nrows for record %d \n", ncols, recnt);
    #endif
    for ( j = 0; j < ncols; j++ ) {
        coldata_ptr = result->colData + j;
        colwidth = *(result->colWidth + j);
        ind_ptr = sqlda_ptr->sqlvar[j].sqlind;
        data_ptr = sqlda_ptr->sqlvar[j].sqldata;
        lensql = sqlda_ptr->sqlvar[j].sqllen;
        #ifdef DEBUG1
        printf("column %d type %d lensql %d colwidth %d ",
            data_ptr, sqlda_ptr->sqlvar[j].sqltype, lensql, colwidth);
        #endif
        if (j) { /* Print new line between rows */
            printf("\n");
        }
        retc = ConRow(sqlvar[j].sqlind, sqlvar[j].sqldata, sqlvar[j].sqllen);
        if ( l ) {
            /* Set returned value to -1 */
            l = -1;
        }
    }
    #endif
}

Figure 160 (Part 40 of 53). Dynamic SQL Server Manager
while (1) {
    switch(sqlda_ptr->sqlvar[j].sqltype) {
        /*ifdef DB2 */
        case T_DEC:
            printf("(CONROW) not yet Processed type %d \n", sqlda_ptr->sqlvar[j].sqltype);
            break;
        case TN_DEC:
            printf("(CONROW) not yet Processed type %d \n", sqlda_ptr->sqlvar[j].sqltype);
            break;
        /*#endif */
        case T_INT:
            coldata_ptr->ColDataUnion.integerCol.data[recnt] = *(idl_long_int *)data_ptr;
        #ifdef DEBUG2
            printf("T_INT %d \n", *(idl_long_int *)data_ptr);
        #endif
            break;
        case T_SMINT:
            coldata_ptr->ColDataUnion.smallintCol.data[recnt] = *(idl_short_int *)data_ptr;
        #ifdef DEBUG2
            printf("T_SMINT %d \n", *(idl_short_int *)data_ptr);
        #endif
            break;
        case T_FLOAT:
            if (lensql > FLOATLEN) {
                coldata_ptr->ColDataUnion.doubleCol.data[recnt] = *(idl_long_float *)data_ptr;
            } else {
                coldata_ptr->ColDataUnion.floatCol.data[recnt] = *(idl_short_float *)data_ptr;
            }
        #ifdef DEBUG2
            printf("T_FLOAT %g \n", *(idl_short_float *)data_ptr);
        #endif
            break;
    }
}

Figure 160 (Part 41 of 53). Dynamic SQL Server Manager
break;
case T_CHAR:
nul_ptr = data_ptr + (colwidth-1);
*(char *)nul_ptr = NUL;
if
(coldata_ptr->colDataType == CDT_string) {
  l = strlen(data_ptr);
  l = l - 1;
  for ( k = l ; k >= 0 ; k-- ) {
    if ( *((char *)data_ptr + k) != ' ' ) break;
  } /* endfor */
  k = k + 1;
#ifdef DEBUG2
  printf("Cdt_string length %d ", k);
  fflush(stdout);
#endif
  k = k + 1;
#ifdef DCE
  new_s_ptr = (string)rpc_ss_allocate(k);
#else
  new_s_ptr = (string)malloc(k);
#endif
/* printf("Print stringprt1 %d \n",new_s_ptr );
  fflush(stdout); */
coldata_ptr->ColDataUnion.stringCol.data [ recnt ] =
  new_s_ptr;
nul_ptr = data_ptr + ( k- 1);
*(char *)nul_ptr = NUL;
strcpy(new_s_ptr, (string) data_ptr);
#ifdef DEBUG2
  printf ("T_STRING %s length %d\n", new_s_ptr,strlen(new_s_ptr));
#endif
} else {

Figure 160 (Part 42 of 53). Dynamic SQL Server Manager
ifdef DEBUG2
    printf("Cdt_char length %d ", lensql);
    fflush(stdout);
#endif
new_s_ptr = coldata_ptr->ColDataUnion.charCol.data + ( recnt * colwidth);
/* printf(“Print stringp2 %d \n”,new_s_ptr );
    fflush(stdout); */
strcpy(new_s_ptr, (string) data_ptr);
ifdef DEBUG2
    printf ("T_CHAR %s length %d\n", new_s_ptr,strlen(new_s_ptr));
#endif
}
break;
case T_VCHAR:
case T_LVCHAR:
    l = (short)*data_ptr; /* real len in varchar*/
    data_ptr = data_ptr + 2; /* data_ptr after len */
    l = l - 1;
    for ( k = l ; k >= 0 ; k-- ) {
        if ( *((char *)data_ptr + k) != ' ' ) break;
    } /* endfor */
    k = k + 1;
ifdef DEBUG2
    printf("Cdt_vstring length %d ", k);
    fflush(stdout);
#endif
k = k + 1;
ifdef DCE
    new_s_ptr = (string)rpc_ss_allocate(k);
#else
    new_s_ptr = (string)malloc(k);
#endif
Figure 160 (Part 43 of 53). Dynamic SQL Server Manager
/ * printf(“Print string.ptrl %d \n”,new_s_ptr );
 * fprintf(stdout); */
coldata_ptr->ColDataUnion.stringCol.data [ recnt ] = new_s_ptr;
nul_ptr = data_ptr + ( k- 1);
*(char *)nul_ptr = NUL;
strcpy(new_s_ptr, (string) data_ptr);
#endif DEBUG2
printf(“T_STRING %s length %d\n”,
new_s_ptr,strlen(new_s_ptr));
#endif
break;
case TN_INT:
if (*ind_ptr >= 0 )
coldata_ptr->ColDataUnion.integerCol.data [ recnt ] = *(idl_long_int *)data_ptr;
else
coldata_ptr->ColDataUnion.integerCol.data [ recnt ] = 0;
if (flags & CDF_shipIndicators)
coldata_ptr->ColDataUnion.integerCol.indicator [ recnt ] = *(idl_short_int *)ind_ptr;
#endif DEBUG2
printf(“TN_INT (%d) %d \n”, *(idl_short_int *)ind_ptr ,
*(idl_long_int *)data_ptr );
#endif
break;
case TN_SMINT:
if (*ind_ptr >= 0 )
coldata_ptr->ColDataUnion.smallintCol.data [ recnt ] = *(idl_short_int *)data_ptr;
else
coldata_ptr->ColDataUnion.smallintCol.data [ recnt ] = 0;
if (flags & CDF_shipIndicators)
coldata_ptr->ColDataUnion.smallintCol.indicator [ recnt ] = *(idl_short_int *)ind_ptr;
#endif DEBUG2
printf(“TN_SMINT (%d) %d \n”, *(idl_short_int *)ind_ptr ,
*(idl_short_int *)data_ptr );
#endif
break;

Figure 160 (Part 44 of 53). Dynamic SQL Server Manager
case TN_FLOAT:
    if (lensql > FLOATLEN) {
        if (*ind_ptr >= 0 )
            coldata_ptr->ColDataUnion.doubleCol.data [ recnt ] =
              *(idl_long_float *)data_ptr;
        else
            coldata_ptr->ColDataUnion.doubleCol.data [ recnt ] = 0;
        if (flags & CDF_shipIndicators)
            coldata_ptr->ColDataUnion.doubleCol.indicator [ recnt ] =
              *(idl_short_int *)ind_ptr;
    } else {
        if (*ind_ptr >= 0 )
            coldata_ptr->ColDataUnion.floatCol.data [ recnt ] =
              *(idl_short_float *)data_ptr;
        else
            coldata_ptr->ColDataUnion.floatCol.data [ recnt ] = 0;
        if (flags & CDF_shipIndicators)
            coldata_ptr->ColDataUnion.floatCol.indicator [ recnt ] =
              *(idl_short_int *)ind_ptr;
    }
    break;

case TN_CHAR:
    nul_ptr = data_ptr + (colwidth-1);
    *(char *)nul_ptr = NUL;
    if
    (coldata_ptr->colDataType == CDT_string) {
        if (*ind_ptr >= 0 ) {
            l = strlen(data_ptr);
            l = l - 1;
            for ( k = l ; k >= 0 ; k-- ) {
                if ( *((char *)data_ptr + k) != ' ' ) break;
            } /* endfor */
            k = k + 1;
        } /* endif */
       .GetHashCode("Cdt_n_string length %d ", k);
        fflush(stdout);
    } /* ifdef DEBUG2 */
    k = k + 1;
#endif DCE
    new_s_ptr = (string)rpc_ss_allocate(k);
#else
    new_s_ptr = (string)malloc(k);
#endif

Figure 160 (Part 45 of 53). Dynamic SQL Server Manager
/* printf("Print stringprt1 %d \n", new_s_ptr );
   fflush(stdout);
 */
coldata_ptr->ColDataUnion.stringCol.data[recnt] = new_s_ptr;
nul_ptr = data_ptr + (k - 1);
*(char *)nul_ptr = NULL;
strcpy(new_s_ptr, (string)data_ptr);
} else {
    coldata_ptr->ColDataUnion.stringCol.data[recnt] = NULL;
} 
if (flags & CDF_shipIndicators)
coldata_ptr->ColDataUnion.stringCol.indicator[recnt] = *(idl_short_int *)ind_ptr;
#else DEBUG2
    printf("TN_STRING (%d) %s \n", *(idl_short_int *)ind_ptr, (string)data_ptr);
#endif
#else DEBUG2
    printf("Cdt_n_char length %d ", lensql);
    fflush(stdout);
#endif
new_s_ptr = coldata_ptr->ColDataUnion.charCol.data + 
    (recnt) * colwidth;
/* printf("Print stringprt2 %d \n", new_s_ptr );
   fflush(stdout);
 */
if (*ind_ptr >= 0) {
    strcpy(new_s_ptr, (string)data_ptr);
} else {
    memcpy(new_s_ptr, manynulls, lensql);
}
if (flags & CDF_shipIndicators)
coldata_ptr->ColDataUnion.charCol.indicator[recnt] = *(idl_short_int *)ind_ptr;
}
printf("TN_CHAR (%d) %s \n", *(idl_short_int *)ind_ptr, 
    (string)data_ptr);
break;

Figure 160 (Part 46 of 53). Dynamic SQL Server Manager
case TN_VCHAR:
case TN_LVCHAR:
    if (*ind_ptr >= 0 ) {
        l = (short)*data_ptr; /* real len in varchar*/
        data_ptr = data_ptr + 2; /* data_ptr after len */
        l = l - 1;
        for ( k = l ; k >= 0 ; k-- ) {
            if ( *((char *)data_ptr + k) != ' ' ) break;
        } /* endfor */
        k = k + 1;
        #ifdef DEBUG2
        printf("Cdt_n_vstring length %d ", k);
        fflush(stdout);
        #endif
        k = k + 1;
        #ifdef DCE
        new_s_ptr = (string)rpc_ss_allocate(k);
        #else
        new_s_ptr = (string)malloc(k);
        #endif
        /* printf("Print stringprt1 %d \n", new_s_ptr );
        fflush(stdout); */
        coldata_ptr->ColDataUnion.stringCol.data [ recnt ] =
            new_s_ptr;
        nul_ptr = data_ptr + ( k- 1);
        *(char *)nul_ptr = NUL;
        strcpy(new_s_ptr, (string) data_ptr);
    } else {
        coldata_ptr->ColDataUnion.stringCol.data [ recnt ] = NULL;
    }
    if (flags & CDF_shipIndicators)
        coldata_ptr->ColDataUnion.stringCol.indicator [ recnt ] =
            *(idl_short_int *)ind_ptr;
        #ifdef DEBUG2
        printf("TN_VSTRING (%d) %s \n", *(idl_short_int *)ind_ptr,
            (string)data_ptr );
        #endif
        break;

Figure 160 (Part 47 of 53). Dynamic SQL Server Manager
case T_DATE:
    npos = recnt * colwidth;
#elifdef DEBUG2
    printf("Cdt_date length %d ", lensql);  
#endif
    cdt_ptr = (idl_char *)coldata_ptr->ColDataUnion.dateCol.data;
    cdt_ptr = cdt_ptr + npos;
    nul_ptr = data_ptr + (colwidth-1);
    *(char *)nul_ptr = NUL;
    strcpy(cdt_ptr,(idl_char *)data_ptr);
#elifdef DEBUG2
    printf("T_DATE %s 
", (string) data_ptr);
#endif
    break;

case T_TIME:
    npos = recnt * colwidth;
#elifdef DEBUG2
    printf("Cdt_time length %d ", lensql);  
#endif
    cdt_ptr = (idl_char *)coldata_ptr->ColDataUnion.timeCol.data;
    cdt_ptr = cdt_ptr + npos;
    nul_ptr = data_ptr + (colwidth-1);
    *(char *)nul_ptr = NUL;
    strcpy(cdt_ptr,(idl_char *)data_ptr);
#elifdef DEBUG2
    printf("T_TIME %s 
", (string) data_ptr);
#endif
    break;

case T_TMSTMP:
    npos = recnt * colwidth;
#elifdef DEBUG2
    printf("Cdt_tmstmp length %d ", lensql);  
#endif
cdt_ptr = (idl_char *)coldata_ptr->ColDataUnion.tmstmpCol.data;
    cdt_ptr = cdt_ptr + npos;
    nul_ptr = data_ptr + (colwidth-1);
    *(char *)nul_ptr = NUL;
    strcpy(cdt_ptr,(idl_char *)data_ptr);
#else 
    printf("T_TMSTMP %s \n", (string) data_ptr);
#endif 
break;

case TN_DATE:
    npos = recnt * colwidth;
    if (*ind_ptr >= 0 ) {
        nul_ptr = data_ptr + (colwidth-1);
        *(char *)nul_ptr = NUL;
        strcpy(coldata_ptr->ColDataUnion.dateCol.data + npos,
            (idl_char *)data_ptr);
    } else
        strcpy(coldata_ptr->ColDataUnion.dateCol.data + npos,"");
    if (flags & CDF_shipIndicators)
        coldata_ptr->ColDataUnion.dateCol.indicator [ recnt ] =
            *(idl_short_int *)ind_ptr;
#else 
    printf("TN_DATE (%d) %s \n", *(idl_short_int *)ind_ptr,
        coldata_ptr->ColDataUnion.dateCol.data + npos);
#endif 
break;

case TN_TIME:
    npos = recnt * colwidth;
    if (*ind_ptr >= 0 ) {
        nul_ptr = data_ptr + (colwidth-1);
        *(char *)nul_ptr = NUL;
        strcpy(coldata_ptr->ColDataUnion.timeCol.data + npos,
            (idl_char *)data_ptr);
    } else
        strcpy(coldata_ptr->ColDataUnion.timeCol.data + npos,"");
    if (flags & CDF_shipIndicators)
        coldata_ptr->ColDataUnion.timeCol.indicator [ recnt ] =
            *(idl_short_int *)ind_ptr;
#else 
    printf("TN_TIME (%d) %s \n", *(idl_short_int *)ind_ptr,
        coldata_ptr->ColDataUnion.timeCol.data + npos);
#endif 
break;

case TN_TMSTMP:
    npos = recnt * colwidth;
    if (*ind_ptr >= 0 ) {
        nul_ptr = data_ptr + (colwidth-1);
        *(char *)nul_ptr = NUL;
        strcpy(coldata_ptr->ColDataUnion.tmstmpCol.data + npos,
            (idl_char *)data_ptr);
    } else

Figure 160 (Part 49 of 53). Dynamic SQL Server Manager
strcpy(coldata_ptr->ColDataUnion.tmstmpCol.data + npos,"\"");
    if (flags & CDF_shipIndicators)
        coldata_ptr->ColDataUnion.tmstmpCol.indicator[recnt] = *(idl_short_int*)ind_ptr;
#endif DEBUG2
    printf("TN_TMSTMP (%d) %s \n", *(idl_short_int*)ind_ptr,
            coldata_ptr->ColDataUnion.dateCol.data + npos);
#endif DEBUG2
    break;
    default:
#endif DEBUG2
    printf("(CONROW) invalid type detected %d \n", sqlda_ptr->sqlvar[j].sqltype);
#endif DEBUG2
    break;
} /* end SWITCH */
    fflush(stdout);
    break;
} /* end WHILE */
} /* end FOR */
retc = 0;
    return(retc);
} /* end CONROW */
int ComVal(VLHandle result, int flags)
{
  int retc;
  short i, j;
  char *w_ptr;
  int coltype;
  int colwidth;
  ColDataType *coldata_ptr;

  #ifdef DEBUG1
  printf(("COMVAL) rowsProcessed = %d ncols = %d \n", result->nrowsProcessed, result->ncols);
  #endif
  for (i = 0; i < result->ncols; i++) {
    coldata_ptr = result->colData + i;
    colwidth = result->colWidth[i];
    coltype = coldata_ptr->colDataType;
    #ifdef DEBUG2
    printf(("COMVAL) col %d type %d colwidth %d \n", i, coltype, colwidth);
    #endif
    while (1) {
      switch (coltype) {
        case CDT_integer:
        case CDT_smallint:
        case CDT_double:
        case CDT_float:
        case CDT_string:
          coldata_ptr->ColDataUnion.rawCol.nrows = result->nrowsProcessed;
          break;
        case CDT_char:
        }
    }
  }
}

Figure 160 (Part 51 of 53). Dynamic SQL Server Manager
ifdef DEBUG2
    w_ptr = coldata_ptr->ColDataUnion.charCol.data;
    for (j = 0; j < result->nrowsProcessed; j++) {
        printf("TCHAR %d %s l %d\n", j, w_ptr, strlen(w_ptr));
        w_ptr = w_ptr + colwidth;
    }
#endif

case CDT_date:
case CDT_time:
case CDT_tmstmp:
    if (result->flags & CDF_shipIndicators)
        coldata_ptr->ColDataUnion.charCol.nrows =
            result->nrowsProcessed;
    else
        coldata_ptr->ColDataUnion.charCol.nrows = 0;
        coldata_ptr->ColDataUnion.charCol.nbytes =
            result->nrowsProcessed * colwidth;
#endif DEBUG2
    printf("nbytes -> %d\n",
            coldata_ptr->ColDataUnion.charCol.nbytes);
#endif
    break;

case CDT_packed:
    printf("CDT_type %d not yet processed \n",
            coltype);
    if (result->flags & CDF_shipIndicators)
        coldata_ptr->ColDataUnion.packedCol.nrows =
            result->nrowsProcessed;
    else
        coldata_ptr->ColDataUnion.charCol.nrows = 0;
        coldata_ptr->ColDataUnion.packedCol.nbytes =
            result->nrowsProcessed * colwidth;
        /* voorlopig */
        coldata_ptr->ColDataUnion.packedCol.data = NULL;
        coldata_ptr->ColDataUnion.packedCol.indicator = NULL;
    break;

default:
    printf("invalid CDT_type %d detected \n",
            coltype);
    break;
} /* end SWITCH */
break;
} /* end WHILE */
} /* end FOR 2 */
retc =0;

Figure 160 (Part 52 of 53). Dynamic SQL Server Manager
```c
#ifdef DEBUG2
    printf("( COMVAL ) Leaving \n");
    printf("nbytes -> %d\n",coldata_ptr->ColDataUnion.charCol.nbytes);
    fflush(stdout);
#endif
    return(retc);
} /* end COMVAL */
/*--------------------------------------------------------------*/
/*----------SQLCAP SQLCAP SQLCAP----------*/
/*--------------------------------------------------------------*/
sqlcAP(long sqlcode, char *segment, char *output)
{
    long rc;
    char sprintf_buf [ 100 ] ;

    printf("SQLcode in error %d in %s \n",sqlcode,segment);
    strcpy(output,"SQLcode in error ");
    sprintf (sprintf_buf,"%+09i",sqlcode);
    strncat(output,sprintf_buf,9);
    strcat(output,"in ");
    strcat(output,segment);
    fflush(stdout);
    return;
}

Figure 160 (Part 53 of 53). Dynamic SQL Server Manager
```
Appendix G. Installation Prerequisites

The following identify the system requirements for installing IBM OpenEdition DCE Base Services MVS Feature, and the optional IBM OpenEdition DCE User Data Privacy Feature of MVS/ESA System Product V5.1.

The following table, Table 8, lists the products required to be installed before or concurrently with IBM OpenEdition DCE Base Services MVS, (features 5891 or 5892) and the optional IBM OpenEdition DCE User Data Privacy Feature, (features 5988 or 5989) along with those products necessary for the execution of IBM OpenEdition DCE Base Services MVS and the optional IBM OpenEdition DCE User Data Privacy Feature.

Products identified as optional for execution are only necessary for full function. A subset of product function will be available if the indicated product is not installed on your system.

<table>
<thead>
<tr>
<th>Product Name</th>
<th>Program Number</th>
<th>Program FMID</th>
<th>VRM</th>
<th>Required For Install</th>
<th>Required For Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple Virtual Storage/Enterprise Systems Architecture (MVS/ESA)</td>
<td>5655-068 or 5655-069</td>
<td>HBB5510</td>
<td>5.1.0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>OpenEdition System Services and C/370 support feature (5831 or 5832)</td>
<td>5655-068 or 5655-069</td>
<td>HOM1120</td>
<td>1.2.0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>OpenEdition Shell and Utilities</td>
<td>5655-068 or 5655-069</td>
<td>HSU1120</td>
<td>1.2.0</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>AD/Cycle LE/370 RTL (Run Time Library)</td>
<td>5688-198</td>
<td>HMLW310</td>
<td>1.3.0</td>
<td>NO</td>
<td>Optional (See C/370 support feature 5831 or 5832 above)</td>
</tr>
<tr>
<td>AD/Cycle C/370 Compiler</td>
<td>5688-216</td>
<td>HSQ4201</td>
<td>1.2.0</td>
<td>NO</td>
<td>Optional</td>
</tr>
<tr>
<td>Data Facility System Managed Storage (DFSMS/MVS)</td>
<td>5695-DF1</td>
<td>HDZ11B0</td>
<td>1.2.0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>RACF or equivalent</td>
<td>5695-039</td>
<td>HRF2210</td>
<td>2.1.0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Transmission Control Protocol/Internet Protocol (TCP/IP)</td>
<td>5655-HAL</td>
<td>HTCP310</td>
<td>3.1.0</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>System Modified Program Extended (SMP/E)</td>
<td>5688-949</td>
<td>HMP1800</td>
<td>1.8.0</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>ISPF</td>
<td>5665-402</td>
<td>HIF3502</td>
<td>3.5.0</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
### G.1 Required PTFs

<table>
<thead>
<tr>
<th>Product Name</th>
<th>Program FMID</th>
<th>PUT Level</th>
<th>PTF Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Facility System Managed Storage (DFSMS/MVS)</td>
<td>HDZ11B0</td>
<td>PUT9403</td>
<td>UW04232</td>
</tr>
<tr>
<td>SMP/E Release 8 Base</td>
<td>HMP1800</td>
<td>PUT9404</td>
<td>UR41949</td>
</tr>
<tr>
<td>SMP/E Release 8 Base</td>
<td>HMP1800</td>
<td>PUT9404</td>
<td>UR41937</td>
</tr>
<tr>
<td>SMP/E Release 8 Base</td>
<td>HMP1800</td>
<td>PUT9404</td>
<td>UR41820</td>
</tr>
<tr>
<td>SMP/E Release 8 English</td>
<td>JMP1801</td>
<td>PUT9404</td>
<td>UR41955</td>
</tr>
<tr>
<td>SMP/E Release 8 Kanji</td>
<td>JMP1811</td>
<td>PUT9404</td>
<td>UR41956</td>
</tr>
<tr>
<td>AD/Cycle LE/370 Common Exec Environment</td>
<td>HMWL310</td>
<td>PUT9405</td>
<td>UN61462</td>
</tr>
<tr>
<td>AD/Cycle LE/370 NL Resources</td>
<td>JMWL31F</td>
<td>PUT9406</td>
<td>UN61209</td>
</tr>
<tr>
<td>AD/Cycle LE/370 ‘C’ Mixed Case</td>
<td>JMWL35B</td>
<td>PUT9406</td>
<td>UN61210</td>
</tr>
<tr>
<td>AD/Cycle LE/370 ‘C’ Library</td>
<td>JMWL350</td>
<td>PUT9406</td>
<td>UN61215</td>
</tr>
<tr>
<td>AD/Cycle LE/370 Common Exec Environment</td>
<td>HMWL310</td>
<td>PUT9406</td>
<td>UN62072</td>
</tr>
<tr>
<td>AD/Cycle LE/370 Mixed Case English</td>
<td>JMWL31B</td>
<td>PUT9406</td>
<td>UN62073</td>
</tr>
<tr>
<td>OpenEdition MVS Application Services</td>
<td>HOT1120</td>
<td>PUT9407</td>
<td>UW07995</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>HTCP310</td>
<td>PUT9410</td>
<td>UN68237</td>
</tr>
</tbody>
</table>

### G.2 Prerequisites for OpenEdition DCE Application Support 1.1.0

The OpenEdition DCE Application Support Server is a server that allows client applications anywhere in the Distributed Computing Environment (DCE) to utilize the resources of the Customer Information Control System (CICS) and the Information Management System (IMS). The OpenEdition DCE Application Support Server is a separately orderable feature of MVS/ESA SP 5.1. The OpenEdition DCE Application Support Server is not discussed in this book, for more information please refer to *MVS/ESA OpenEdition DCE: Application Support Server for IMS and CICS*. The following table, Table 10 on page 397 lists the products required to be installed before OpenEdition DCE Application Support 1.1.0, and those products necessary for the execution of OpenEdition DCE Application Support 1.1.0. Products identified as optional for execution are only necessary for full function. A subset of product function will be available if the indicated product is not installed on your system.
<table>
<thead>
<tr>
<th>Product Name</th>
<th>Program Number</th>
<th>Program FMID</th>
<th>VRM</th>
<th>Required For Install</th>
<th>Required For Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMS/ESA</td>
<td>5655-409</td>
<td>HMJ3102</td>
<td>3.1.0 or later</td>
<td>YES (AS/IMS only)</td>
<td>YES (AS/IMS only)</td>
</tr>
<tr>
<td>CICS/ESA</td>
<td>5655-018</td>
<td>HCI4100</td>
<td>4.1.0</td>
<td>YES (AS/CICS only)</td>
<td>YES (AS/CICS only)</td>
</tr>
<tr>
<td>ACF/VTAM</td>
<td>5685-085</td>
<td>HVT3401</td>
<td>3.4.1 or later</td>
<td>YES (AS/IMS only)</td>
<td>YES (AS/IMS only)</td>
</tr>
<tr>
<td>AD/Cycle COBOL/370</td>
<td>5688-197</td>
<td>HMWL170</td>
<td>1.1.0</td>
<td>NO</td>
<td>Optional</td>
</tr>
<tr>
<td>OpenEdition DCE Base Services</td>
<td>5655-068 or 5655-069</td>
<td>HMB3120</td>
<td>1.1.0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>OpenEdition DCE Data User Privacy Feature</td>
<td>5655-068 or 5655-069</td>
<td>JMB3121</td>
<td>1.1.0</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>
Index

Special Characters
.ACFile 35
.IDL file 31

A
access control 15
definition 15
access control lists 80
Access to HFS files 107
Access to MVS data sets 107
Access to RACF general resources 107
account (add) 187
ACF 54, 85
ACL 15, 18, 80, 81, 95
ACL handle 95
ACL manager 80
acledit 184
administrator machine 20
AFS 18
aggregate 18
definition 18
Andrew File System
See AFS
Appearance of a DCE server to RACF 106
Application resources controlled by RACF 108
Application Support Server 152
Attribute Configuration File 86
Authentication 15, 77, 79
definition 15
Third-party 15
Authentication Levels 79
Authentication Service 15
definition 15
Authorization 15, 77, 80
definition 15
Automatic Binding 49

B
Binding Methods 50
automatic binding 50
explicit binding 50
implicit binding 50

C
C 21
ANSI C 21
SAA AD/Cycle C/370 21
C/370 21
CCTL 152, 157
DCEManager use 157
CCTL and IMS prerequisites 165
CCTL implementation 154
CCTL interface 153
CCTL start-up table 154
CDS 15, 47, 50, 53
definition 15
CDS Directory (creating) 189
CDS Namespace 189
CDS Object (creating) 189
CDS Object (edit) 190
cdscp 184
directory 184
object 184
cell 20
cell configuration 20
client 20
Client Module 98
Client Module C Source 101
Client/Server 11
Common Header File 101
Communication Protection Levels 80
Compiling and Link Editing 69
ACF 69
Client 69
IDL 69
Server 69
Compiling Client/Server 73
In Batch 73
In TSO/E 73
Compiling the IDL 72
In batch 72
In TSO/E 72
Concepts of DCE Security 77
conformant arrays 128, 129
constructed data types 128
container objects 82
courier 17
definition 17
courier time server 17
creating a DCE account 186
add account 186
add group 186
add organization 186
add principal 186
keytab file 186
KTADD 186
permissions 186
rpc-server-group 186
Creating files 65
HFS 65
In TSO/E and Batch 65
PDS 65
TSO/E OGET 65
TSO/E OPUT 65
D
Data Conversion 4
Data Placement 4
Data Staging 6
data types 127
DBCTL 152
DBCTL interface 152
DCCTL 152
DCE 1
DCE Access Control Lists 82
DCE application server on MVS 105
DCE Authorization 81
DCE client on MVS 112
DCE server start-up parameters 155
DCE System Secret Key Utility 122
DCE technology 1
DCE Threads 13
  thread communication 14
  thread exceptions 14
  thread management 14
  thread scheduling 14
DCECCTL 154
DCECCTL implementation 154
  CTLINIT 154
  CTLTERM 154
DDM 8
Description of the Sample Application 84
Design Philosophy 2
DFS 18
DFSDCEI 154
DFSMS 22
DFSPRP macro 154, 155
  example 155
DFSPZPxx table 154
DFSSCHED 157
DFSTRMAB 157
DFSTRMCM 157
directory 15
Directory services 15
distributed file system (dfs) 18
Distributed Logic 6
Distributed Time Service
  See DTS
Distribution Points 2
distribution possibilities 5
DL/I access 152
DNS 8
domain 20
DTS 17, 20
  definition 20
DTS local server 17
  definition 17
dynamic allocation 139

E
ECIP 1 48
ECIP 2 48
Encrypted DCE password in RACF database 114
endpoint map 43
endpoint mapper 86
endpoints 42
entry point vector (EPV) 39
Environmental variables 191
  _EUV_CDS_CLIENT_CLERK_LINK_TIMEOUT 193
  _EUV_CDS_CLIENT_CLERK_LINK_TIMEOUT 193
  _EUV_CFG_CDSD_MACHINE_NAME 195
  _EUV_CFG_CDSD_MACHINE_NAME 195
  _EUV_CFG_CELL_ID 195
  _EUV_CFG_CELL_NAME 195
  _EUV_CFG_INFORM_LEVEL 195
  _EUV_CFG_INFORM_LEVEL 195
  _EUV_CFG_LOG_FILE 195
  _EUV_CFG_SECD_MACHINE_NAME 195
  _EUV_ECHO_STDIN 192
  _EUV_ENVAR_FILE 192
  _EUV_EXC_ABNE_DUMPS 193
  _EUV_EXC_DUMPS 192
  _EUV_EXC_SW_DUMPS 193
  _EUV_FTRACE 193
  _EUV_HOME 192
  _EUV_IDL_PRELINKER_NAME 195
  _EUV_IDL_WORKDA_UNIT 195
  _EUV_RACF_FACILITY_NAME 193
  _EUV_RACF_FACILITY_NAME 193
  _EUV_RPC_ACL_POOL 193
  _EUV_RPC_COMM_TIMEOUT 192
  _EUV_RPC_DYNAMIC_POOL 191, 193
  _EUV_RPC_MAX_THRESHOLD 193
  _EUV_RPC_MIN_THRESHOLD 193
  _EUV_SEC_KRB5CCNAME_FILE 192
  _EUV_CDS_CLIENT_CLERK_LINK_TIMEOUT 193
  _EUV_CDS_CLIENT_CLERK_LINK_TIMEOUT 193
  _EUV_RPC_ACL_POOL 193
  _EUV_RPC_COMM_TIMEOUT 192
  _EUV_RPC_DYNAMIC_POOL 191, 193
  _EUV_RPC_MAX_THRESHOLD 193
  _EUV_RPC_MIN_THRESHOLD 193
  _EUV_SEC_KRB5CCNAME_FILE 192
  _EUV_SVC_API_DUMPS 191, 195
  _EUV_SVC_DBG_MSG_LOGGING 195
  _EUV_SVC_DBG_MSG_LOGGING 194
  _EUV_SVC_MSG_LEVEL 194
  _EUV_SVC_MSG_LOGGING 194
  BIND_PE_SITE 192
  KRB5CCNAME 192
  NLSPATH 191, 192
  RPC_DEFAULT-ENTRY 191
  RPC_DEFAULT-ENTRY_SYNTAX 191
  RPC_DEFAULT_ENTRY 192
  RPC_DEFAULT_ENTRY_SYNTAX 192
  RPC_RESTRICTED_PORTS 193
  RPC_SUPPORTED_PROTSEQS 193
  SVC_CDSDBG 195
  SVC_DTSDBG 195
  SVC_PLTDBG 195
  SVC_RPCDBG 195
  SVC_SECDBG 195
  TZ 192
Episode 18
EPV 39
ERRCHK  47
Example  84
explicit binding  42, 49, 53, 85

F
file  18, 22
   HFS  22
      replication  18
file service  18
file system  18
fileset  18
   definition  18
Front-Ending  5
ftp  18

G
GDA  15
   definition  15
GDS  15
   definition  15
Global Directory Agent
   See GDA
global time server  17
group  15
   definition  15
Group (add)  186

H
handle_t  54
HFS  22
hierarchical file system
   See HFS

I
id=binding. Binding
identification  15
   definition  15
IDL  127
IDL compiler  13, 85
IDL unions  128
idalloc  69
IEEE  21
Implicit Binding  49, 57
IMS database manager  152
IMS databases  152
initializing server code  38
installation prerequisites  393
Institutional File System (IFS)  18
Integrity  15
inter-process communication  3
Interface Definition  85
interface definition language (IDL)  12

K
Kerberos  15, 79, 87
keytab  188
KTADD  188

L
Language Support  3
LE/370  21
LFS  18
Link-Editing your Application  73
   In Batch  73
local file system (LFS)  19
local time server  17
local time translation  17
login context  80

M
make  17, 18
makefile  259
   mathx client (well-known endpoint)  259
   mathx server (well-known endpoint)  261
makefile (sample)  265
Manager Code stub  158
Manager Module  92
Manager Module C Source  93
Manager split-up  158
ManagerCode functional  160
marshalling  12
Modular Design  3
multi-process  125
Multiple Application  6
Multiple Instances  4
mutex  126

N
Name Based Authorization  83
names  15
   global  15
      definition  15
   local  15
      definition  15
NCA  40
ncacn  40
ncadg  40
Network  4
   network computing architecture  40
Network File System
      See NFS
NFS  8

O
object UUID  43, 61
Open blueprint  8
sample output 208, 219
sample source 200, 212, 259, 267, 288, 307, 317
db2 server 267
DL/I 288
mathx output (explicit binding) 208
mathx client 200
mathx manager 206
mathx server 203
sbind client (string bindings) 212
SBind manager (string bindings) 218
SBind server (string bindings) 215
SQL server 307, 317
SAVEPW command 118
sbind 15
definition 15
shared 15
Security 3, 15, 77, 104
access control 15
access control lists 80
Access to HFS files 107
Access to MVS data sets 107
Access to RACF general resources 107
ACF 85
ACL 80, 81
ACL handle 95
ACL manager 80
Appearance of a DCE server to RACF 106
Application resources controlled by RACF 108
Attribute Configuration File 86
authentication 15, 77, 79
Authentication Levels 79
authorization 15, 77, 80
Client Module 98
Client Module C Source 101
Common Header File 101
Communication Protection Levels 80
Concepts of DCE Security 77
container objects 82
DCE Access Control Lists 82
DCE application server on MVS 105
DCE Authorization 81
DCE client on MVS 112
DCE System Secret Key Utility 122
Description of the Sample Application 84
Encrypted DCE password in RACF database 114
endpoint mapper 86
Example 84
explicit binding 85
IDL compiler 85
integrity 15
Interface Definition 85
login context 80
Manager Module 92
Manager Module C Source 93
Name Based Authorization 83
PAC 78, 80, 81
Security (continued)
principals 77
privacy 15
privileged attribute certificate 78
protection levels 80
RACF compared to DCE security 105
Running the Example Application 101
SAVEPW command 118
sec_acl_bind 95
sec_acl_get_manager_types 96
sec_acl_test_access_on_behalf 97
sec_key_mgmt_get_key() 88
sec_login_setup_identity() 87
sec_login_validate_identity() 89
Security controlled by DCE security 106
Security Module 93
Security Module C Source 98
server 15
Server Implementation 86
Server Module 86
Server Module C Source 92
simple objects 82
security controlled by DCE security 106
Security Module 93
Security Module C Source 98
security server 15, 18
server 20, 37
entry point vector (EPV) 37
implementing remote procedures 37
initializing server code 37
registering the interface 37
Server Implementation 86
server machine 20
Server Module 86
Server Module C Source 92
shell 21, 64
simple objects 82
SNA 8
stateful / stateless servers 4
storage allocation 139
string binding 42, 59
Stubs 12
System administrator 20
System Network Architecture
See SNA
system software support 2
system time 17

T
TCP/IP 8
Third-party authentication 15
Thread 158
Thread Safe 125
Threads 125
ticket 77
time clerk 17
Time services 17
Transarc 18
translation 17
  local time 17
Transmission Control Protocol/Internet Protocol
  See TCP/IP

U
UFS 18
unions 128, 137
universal coordinated time
  See UTC
universal coordinated time (UTC) 17
UNIX 18, 21
unmarshalling 12
user machine 20
UTC 17
  definition 17
UUID 15, 18, 31, 83, 90
UUIDgen 31

W
Well-known endpoint 63
  implementing 63
  running with 63
MVS/ESA OpenEdition DCE:
Application Development Cookbook
Publication No. GG24-4481-00

Your feedback is very important to help us maintain the quality of ITSO Bulletins. Please fill out this questionnaire and return it using one of the following methods:

- Mail it to the address on the back (postage paid in U.S. only)
- Give it to an IBM marketing representative for mailing
- Fax it to: Your International Access Code + 1 914 432 8246
- Send a note to REDBOOK@VNET.IBM.COM

Please rate on a scale of 1 to 5 the subjects below.
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

<table>
<thead>
<tr>
<th>Overall Satisfaction</th>
<th>Organization of the book</th>
<th>Accuracy of the information</th>
<th>Relevance of the information</th>
<th>Completeness of the information</th>
<th>Value of illustrations</th>
<th>Grammar/punctuation/spelling</th>
<th>Ease of reading and understanding</th>
<th>Ease of finding information</th>
<th>Level of technical detail</th>
<th>Print quality</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Please answer the following questions:

a) If you are an employee of IBM or its subsidiaries:
   - Do you provide billable services for 20% or more of your time? Yes___ No___
   - Are you in a Services Organization? Yes___ No___

b) Are you working in the USA? Yes___ No___

c) Was the Bulletin published in time for your needs? Yes___ No___

d) Did this Bulletin meet your needs? Yes___ No___

If no, please explain:

__________________________________________________________________________

What other topics would you like to see in this Bulletin?

__________________________________________________________________________

What other Technical Bulletins would you like to see published?

__________________________________________________________________________

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

__________________________________________________________________________

__________________________________________________________________________

Name

Address

Company or Organization

Phone No.