CICS/ESA and TCP/IP for MVS
Sockets Interface

Document Number GG24-4026-00

September 1993

International Technical Support Center
San Jose
Take Note!

Before using this information and the product it supports, be sure to read the general information under “Special Notices” on page xiii.

First Edition (September 1993)

This edition applies to TCP/IP Version 2 Release 2.1 for MVS, 5735-HAL.

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

An ITSC 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 Center
Dept. 471, Building 098
5600 Cottle Road
San Jose, California 95193-0001

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 1993. 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

This document gives practical advice for implementing, operating and writing applications that use the IBM CICS to TCP/IP Version 2 Release 2.1 for MVS Sockets Interface (part of 5735-HAL - TCP/IP Version 2 Release 2.1 for MVS).

This document is intended for technical professionals involved in the planning, installation, programming and/or maintenance of a TCP/IP application that involves a CICS/ESA system.

A knowledge of application programming under CICS (in either COBOL or C) and the basics of coding TCP/IP sockets applications is assumed.

DS CO (192 pages)
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Special Notices</td>
<td>xiii</td>
</tr>
<tr>
<td>Preface</td>
<td>xv</td>
</tr>
<tr>
<td>How This Document Is Organized</td>
<td>xv</td>
</tr>
<tr>
<td>Related Publications</td>
<td>xvi</td>
</tr>
<tr>
<td>International Technical Support Center Publications</td>
<td>xvi</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>xvii</td>
</tr>
<tr>
<td><strong>Chapter 1. Introduction</strong></td>
<td></td>
</tr>
<tr>
<td>1.1 CICS and TCP/IP</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Introduction to TCP/IP</td>
<td>1</td>
</tr>
<tr>
<td>1.2.1 The TCP/IP Protocol Suite</td>
<td>2</td>
</tr>
<tr>
<td>1.2.2 The Application Layer</td>
<td>2</td>
</tr>
<tr>
<td>1.2.3 The CICS to TCP/IP Sockets Interface</td>
<td>3</td>
</tr>
<tr>
<td>1.2.4 The High-Level Protocol Layer</td>
<td>3</td>
</tr>
<tr>
<td>1.2.5 The Internet Layer</td>
<td>3</td>
</tr>
<tr>
<td>1.2.6 The Network Access Layer</td>
<td>3</td>
</tr>
<tr>
<td>1.3 What Is a Socket?</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Alternatives to Using the CICS to TCP/IP Sockets Interface</td>
<td>4</td>
</tr>
<tr>
<td>1.5 TCP/IP and CICS Intercommunication Facilities</td>
<td>5</td>
</tr>
<tr>
<td><strong>Chapter 2. CICS to TCP/IP Sockets Interface</strong></td>
<td>7</td>
</tr>
<tr>
<td>2.1 Structure of the CICS to TCP/IP Sockets Interface</td>
<td>7</td>
</tr>
<tr>
<td>2.1.1 Interface between CICS and the TCPIP Address Space</td>
<td>7</td>
</tr>
<tr>
<td>2.1.2 TRUE Implementation for the CICS to TCP/IP Sockets Interface</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Components of the CICS to TCP/IP Sockets Interface</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Calls to the Sockets Interface</td>
<td>10</td>
</tr>
<tr>
<td>2.3.1 Processing the Initial Call</td>
<td>10</td>
</tr>
<tr>
<td>2.3.2 Processing Subsequent Calls</td>
<td>11</td>
</tr>
<tr>
<td><strong>Chapter 3. Implementing the CICS to TCP/IP Sockets Interface</strong></td>
<td>13</td>
</tr>
<tr>
<td>3.1 Maintenance Required on TCP/IP Version 2 Release 2.1 for MVS</td>
<td>13</td>
</tr>
<tr>
<td>3.2 Changes to the CICS Startup JCL</td>
<td>14</td>
</tr>
<tr>
<td>3.3 Definitions Needed in CICS</td>
<td>15</td>
</tr>
<tr>
<td>3.3.1 Transaction Definitions</td>
<td>15</td>
</tr>
<tr>
<td>3.3.2 Program Definitions</td>
<td>17</td>
</tr>
<tr>
<td>3.3.3 BMS Mapset Definition</td>
<td>20</td>
</tr>
<tr>
<td>3.3.4 Transient Data Queues</td>
<td>21</td>
</tr>
<tr>
<td>3.3.5 Terminal Definition</td>
<td>22</td>
</tr>
<tr>
<td>3.4 Changes to COBOL Compilation Procedures</td>
<td>23</td>
</tr>
<tr>
<td>3.5 Changes to C/370 Compilation Procedures</td>
<td>27</td>
</tr>
<tr>
<td>3.6 Definitions Needed in TCPIP Data Sets</td>
<td>29</td>
</tr>
<tr>
<td>3.6.1 The PROFILE Data Set</td>
<td>29</td>
</tr>
<tr>
<td>3.6.2 The DATA Data Set</td>
<td>31</td>
</tr>
<tr>
<td><strong>Chapter 4. Operating the CICS to TCP/IP Sockets Interface</strong></td>
<td>33</td>
</tr>
<tr>
<td>4.1 Enabling the CICS to TCP/IP Sockets Interface</td>
<td>33</td>
</tr>
<tr>
<td>4.1.1 Manual Enable</td>
<td>33</td>
</tr>
<tr>
<td>4.1.2 Automatic Enable</td>
<td>34</td>
</tr>
</tbody>
</table>
9.2.4 Combining the Security Exit and CICS Preset Terminal Security  
9.3 Sample Security Implementation  
9.4 Summary  

Chapter 10. Problem Determination  
10.1 Operating the Sockets Interface  
10.1.1 Enabling the Sockets Interface  
10.1.2 Disabling the Sockets Interface  
10.1.3 Installation Checklist  
10.2 Running Application Programs  
10.2.1 Problem Determination Methods in CICS  
10.2.2 Problem Determination Methods for the TCP/IP Network  

Appendix A. The Supplied Sample Applications  
A.1 Sample Application Programs  
A.1.1 Naming Convention for the Sample Applications  
A.1.2 The Messaging System  
A.1.3 The Record Retrieval System  
A.2 Loading the Sample Applications from the Diskette  
A.2.1 Contents of the Diskette  
A.2.2 Transferring the Sample Applications from the Diskette  
A.3 Preparing the Sample Distributed Message System  
A.3.1 Defining the Sample Message System to CICS  
A.3.2 Preparing the CICS Server Region  
A.3.3 Preparing the CICS Client  
A.3.4 Preparing the OS/2 Client  
A.3.5 Preparing the AIX Client Program  
A.4 Using the Sample Distributed Message System  
A.4.1 Starting the CICS Client  
A.4.2 Starting the OS/2 Client  
A.4.3 Starting the AIX Client  
A.4.4 Example of Using the Sample Clients  
A.5 Preparing the Sample Record Retrieval System  
A.5.1 Defining the Record Retrieval System to CICS  
A.5.2 Preparing the CICS Programs  
A.6 Using the Sample Record Retrieval System  
A.6.1 Starting the COBOL Iterative Server  
A.6.2 Starting the C/370 Iterative Server  
A.6.3 Starting the COBOL Concurrent Server  
A.6.4 Starting the C/370 Client  
A.6.5 Starting the COBOL Client  

Appendix B. The Sockets API for COBOL and Assembler Language Programs  
B.1 Socket Calls in COBOL  
B.1.1 Ordered List of Socket Calls  
B.1.2 ACCEPT  
B.1.3 BIND  
B.1.4 CLOSE  
B.1.5 CONNECT  
B.1.6 Fcntl  
B.1.7 GetClientID  
B.1.8 GetHostID  
B.1.9 GetHostName  
B.1.10 GetPeerName  

Contents vii
Figures

1. TCP/IP Protocol Layers ........................................ 2
2. Using SNA to Connect CICS under MVS with TCP/IP .................. 5
3. CICS to TCP/IP Sockets Interface Subtasks ........................ 8
4. Main Components of the CICS to TCP/IP Sockets Interface .............. 9
5. CICS Startup JCL with CICS to TCP/IP Sockets Interface ............... 14
6. RDO Definition of the CSKD Transaction ................................ 16
7. RDO Definition of the CSKE Transaction ................................ 16
8. RDO Definition of the CSKL Transaction ................................ 17
9. Definition of Program EZACIC00 ................................... 18
10. Definition of Program EZACIC01 .................................... 19
11. Definition of Program EZACIC02 .................................... 19
12. Definition of Program EZACICSC ................................... 20
13. Definition of Program EZACICSS .................................... 20
14. Definition of Mapset EZACICM ....................................... 21
15. DCT Entries Required by the Listener ................................ 21
16. Sample TCT for Sequential Terminal Definition ....................... 22
17. DFHEITVL - Unmodified JCL for Compilation of a COBOL Program ... 24
18. Link-Edit Output Using the Standard Compilation Procedure ........... 25
19. Modified JCL for Compilation of a COBOL Program .................... 26
20. Output from Link-Edit with Modified Compilation Procedure .......... 27
21. Modified JCL for Compilation of a C/370 Program ..................... 28
22. Definition of the TCP/IP Profile .................................... 30
23. Definition of TCP/IP Data ........................................... 31
24. CSKE - Panel for Manual Enable of the Sockets Interface .............. 33
25. Sample JCL for Automatic Enabling of the Sockets Interface .......... 34
26. Successful Enable of Sockets Interface (i) .......................... 35
27. Successful Enable of Sockets Interface (ii) .......................... 35
28. Successful Enable of Sockets Interface (iii) .......................... 35
29. Error Messages while Enabling - Trying to Use a Reserved Port ........ 36
30. Error Messages while Enabling - TCPIPUSERID Incorrect .............. 36
31. CSKD - Panel for Disable of the Sockets Interface ................... 37
32. CSMT Messages for Quiescent Disable of the Sockets Interface ......... 38
33. TCPM Messages for Quiescent Disable of the Sockets Interface ......... 38
34. CSKD - Panel for Disable of the Sockets Interface - Confirm Immediate Disable ........................................... 38
35. Message for Immediate Disable of Sockets Interface ................... 39
36. Iterative Server ................................................... 43
37. Concurrent Server and Child Server .................................. 45
38. sendn() Function .................................................. 46
39. recvn() Function .................................................. 47
40. TCP/IP Buffering .................................................. 48
41. Using Acknowledgments to Preserve Logical Record Separations .......... 49
42. Starting a Child Server Using a Transient Data Queue .................. 53
43. Two Phase Commit Model ........................................... 54
44. Simplified Commit Model ........................................... 55
45. Using the Commit Log in the Server .................................. 56
46. Sample Child Server (IC) - EXEC CICS RETRIEVE Command .......... 60
47. Child Server Started by the EXEC CICS START Command ............. 61
48. Sample Child Server (IC) - Parameters Retrieved from the Listener .... 62
49. Working Storage Declaration for STANDARD-PARMS. ................. 62
50. Sample Child Server (IC) - TAKESOCKET Call ........................ 63
<table>
<thead>
<tr>
<th>Sample Child Server (IC) - SEND Call</th>
<th>Sample Child Server (IC) - RECVFROM Call</th>
<th>Sample Child Server (IC) - CLOSE Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Child Server Started by the EXEC CICS WRITEQ TD Command</td>
<td>Sample Child Server (TD) - EXEC CICS READQ TD Command</td>
<td>Sample Child Server (TD) - Parameters Received from the READQ TD</td>
</tr>
<tr>
<td>Structure of Our Concurrent Server</td>
<td>Sample Concurrent Server - RECEIVE Command</td>
<td>Sample Concurrent Server - INITAPI Call</td>
</tr>
<tr>
<td>Sample Concurrent Server - SOCKET Call</td>
<td>Sample Concurrent Server - SETSOCKOPT Call</td>
<td>Sample Concurrent Server - BIND Call</td>
</tr>
<tr>
<td>Sample Concurrent Server - LISTEN Call</td>
<td>Sample Concurrent Server - GETCLIENTID Call</td>
<td>Fields Used in GETCLIENT-ID Call</td>
</tr>
<tr>
<td>Sample Concurrent Server - SELECT Call</td>
<td>Sample Concurrent Server - Repeating the SELECT Call</td>
<td>Sample Concurrent Server - EXEC CICS INQUIRE TASK</td>
</tr>
<tr>
<td>Sample Concurrent Server - TESTING Response from SELECT</td>
<td>Sample Concurrent Server - ACCEPT Call</td>
<td>Sample Concurrent Server - GIVESOCKET Call</td>
</tr>
<tr>
<td>Sample Concurrent Server - Starting the Child Server</td>
<td>Iterative Server</td>
<td>INITAPI Call for CICS Client</td>
</tr>
<tr>
<td>CONNECT Call for the CICS Client</td>
<td>SEND and RECVFROM Calls from the CICS Client</td>
<td>CLOSE Call for the CICS Client</td>
</tr>
<tr>
<td>Data Passed to the Listener</td>
<td>Data Flow for a ‘read-message’ Request</td>
<td>INITAPI Call for CICS Client</td>
</tr>
<tr>
<td>Sample CICS Client - INITAPI Call</td>
<td>Sample CICS Client - SOCKET Call</td>
<td>Sample CICS Client - CONNECT Call</td>
</tr>
<tr>
<td>Sample OS/2 and AIX Clients - SOCKET Call</td>
<td>Sample CICS Client - CONNECT Call</td>
<td>Sample OS/2 and AIX Clients - CONNECT Call</td>
</tr>
<tr>
<td>Sample CICS Client - SEND Call</td>
<td>Sample OS/2 and AIX Clients - SEND Call</td>
<td>Sample CICS Client - RECVFROM Call</td>
</tr>
<tr>
<td>Sample OS/2 and AIX Clients - RECVFROM Call</td>
<td>Sample CICS Client - RECV Call</td>
<td>Sample CICS Client - CLOSE Call</td>
</tr>
<tr>
<td>Sample OS/2 and AIX Clients - CLOSE Call</td>
<td>Sample OS/2 and AIX Clients - SOCLOSE Call</td>
<td>Sample AIX Client - CLOSE Call</td>
</tr>
<tr>
<td>The Security Exit Invoked by the IBM-Supplied Listener</td>
<td>Data Passed to the Security Exit in the COMMAREA</td>
<td>Recommended Security Checks</td>
</tr>
<tr>
<td>Sample Information for Security Exit</td>
<td>COBOL Declarations Used by Our Sample Security Exit</td>
<td>Check the Client’s Host IP Address</td>
</tr>
<tr>
<td>Check the Client’s ID</td>
<td>The Security Exit Invoked by the IBM-Supplied Listener</td>
<td>Data Passed to the Security Exit in the COMMAREA</td>
</tr>
<tr>
<td>CICS/ESA and TCP/IP for MVS</td>
<td>100. Data Passed to the Security Exit in the COMMAREA</td>
<td>101. Recommended Security Checks</td>
</tr>
<tr>
<td>102. Sample Information for Security Exit</td>
<td>103. COBOL Declarations Used by Our Sample Security Exit</td>
<td>104. Check the Client’s Host IP Address</td>
</tr>
<tr>
<td>105. Check the Client’s ID</td>
<td>106. The Security Exit Invoked by the IBM-Supplied Listener</td>
<td></td>
</tr>
<tr>
<td>Step</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>Return Response to the Listener</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>Check whether the Client Is Authorized for the Server</td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>JCL to Execute DFHTUP</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>Sample CICS Auxiliary Trace Output</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Sample CICS Auxiliary Trace Output</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>Sample GTF Trace Output</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>Sample Security Exit to Start a Server at a Terminal</td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>Modified DCT Entry to Start a Server against a Terminal</td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>Sample CEDF Panel for a TAKESOCKET Call</td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>Sample CEDF Panel for a TAKESOCKET Call (hexadecimal values)</td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>Messages Received for a Normal Enable</td>
<td></td>
</tr>
<tr>
<td>117</td>
<td>Message Received for a Normal Disable</td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>SYSUDUMP and SYSABEND Statements in CICS Startup JCL</td>
<td></td>
</tr>
<tr>
<td>119</td>
<td>GWA and TIE Extracts from a Formatted CICS System Dump</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>PING Command and Responses - Route Active</td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>PING Command and Responses - Route Inactive</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>Naming Convention for the Sample Applications</td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>Transferring Files to MVS Using FTP</td>
<td></td>
</tr>
<tr>
<td>124</td>
<td>Sample Configuration</td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>Data Needing Modification in Program SIULSECC</td>
<td></td>
</tr>
<tr>
<td>126</td>
<td>Default Values in SENDMAP-ROUTINE</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>Compile the OS/2 Client</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>Compile the AIX Client</td>
<td></td>
</tr>
<tr>
<td>129</td>
<td>CICS Client: Initial Panel</td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>CICS Client: Write a Message to BARNEY</td>
<td></td>
</tr>
<tr>
<td>131</td>
<td>OS/2 Client: Query BARNEY’s Mail-log</td>
<td></td>
</tr>
<tr>
<td>132</td>
<td>OS/2 Client: Read a Message from BARNEY’s Mail-log</td>
<td></td>
</tr>
<tr>
<td>133</td>
<td>OS/2 Client: Write a Message to WILMA</td>
<td></td>
</tr>
<tr>
<td>134</td>
<td>AIX Client: Query WILMA’s Mail-log</td>
<td></td>
</tr>
<tr>
<td>135</td>
<td>AIX Client: Write a Message to FRED</td>
<td></td>
</tr>
<tr>
<td>136</td>
<td>CICS Client: Query FRED’s Mail-log</td>
<td></td>
</tr>
<tr>
<td>137</td>
<td>CICS Client: Read the Message from WILMA</td>
<td></td>
</tr>
<tr>
<td>138</td>
<td>Iterative Server</td>
<td></td>
</tr>
<tr>
<td>139</td>
<td>Concurrent and Child Servers</td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>Check whether Sockets Interface Is Being Disabled</td>
<td></td>
</tr>
</tbody>
</table>
Special Notices

This publication is intended to help technical professionals involved in the planning, installation, programming and/or maintenance of a TCP/IP application that involves a CICS/ESA system to carry out that task.

The information in this publication is not intended as the specification of any programming interfaces that are provided by 5735-HAL - IBM TCP/IP Version 2 Release 2 for MVS.

See the PUBLICATIONS section of the IBM Programming Announcement for 5735-HAL - IBM TCP/IP Version 2 Release 2 for MVS 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 document 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 Commercial Relations, IBM Corporation, Purchase, NY 10577.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. 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.

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

The following terms, which are denoted by an asterisk (*) in this publication, are trademarks of the International Business Machines Corporation in the United States and/or other countries:

- ACF/VTAM
- AIX
- CICS
- CICS/ESA
- CICS/6000
- CICS/MVS
- CICS/400
- OS/2
- OS/400
- IBM
- PS/2
- RISC System/6000
- VTAM

© Copyright IBM Corp. 1993
The following terms, which are denoted by a double asterisk (**) in this publication, are trademarks of other companies:

Microsoft

Microsoft Corporation
Preface

This document is intended to help technical professionals involved in the planning, installation, programming and/or maintenance of a TCP/IP application that involves a CICS/ESA system to carry out that task.

The document contains a description of the function of the IBM CICS to TCP/IP Version 2 Release 2.1 for MVS Sockets Interface (which we refer to as the CICS to TCP/IP Sockets Interface or the Sockets Interface from here onwards). It provides instructions on operating the Sockets Interface, examples of applications that use the Sockets Interface, advice on how to implement a security system for the Sockets Interface, and suggestions on how to carry out problem determination procedures.

How This Document Is Organized

The document is organized as follows:

- Chapter 1, "Introduction"
  This chapter gives a brief overview of the CICS to TCP/IP Sockets Interface, and introduces some of the terminology that we will use throughout the document.

- Chapter 2, "CICS to TCP/IP Sockets Interface"
  This chapter gives an overview of the operation of the Sockets Interface, and describes the components that make up the Sockets Interface.

- Chapter 3, "Implementing the CICS to TCP/IP Sockets Interface"
  This chapter describes the definitions that you have to make within your CICS system to use the Sockets Interface. It also describes the changes that you need to make to the JCL in order to start your CICS system, and the changes to the jobs you use to compile application programs under CICS.

- Chapter 4, "Operating the CICS to TCP/IP Sockets Interface"
  This chapter describes how to operate the Sockets Interface.

- Chapter 5, "Application Design Considerations"
  This chapter describes the considerations for designing an application that uses the Sockets Interface.

- Chapter 6, "Overview of the Sample Systems"
  This chapter describes the function of the sample applications that we used to demonstrate some of the features and facilities of the Sockets Interface.

- Chapter 7, "Server Design for the Sample Systems"
  This chapter describes the server programs that we used to implement the sample applications.

- Chapter 8, "Client Design for the Sample Systems"
  This chapter describes the client programs that we used to implement the sample applications.
• Chapter 9, “Implementing Security in the CICS to TCP/IP Sockets Interface”
  This chapter describes one possible way of implementing security in applications that use the Sockets Interface.

• Chapter 10, “Problem Determination”
  This chapter discusses some methods for solving any problems you may have when using the Sockets Interface.

• Appendix B, “The Sockets API for COBOL and Assembler Language Programs”
  This appendix gives detailed information on the API for COBOL and Assembler language programs used with the Sockets Interface.

• Appendix A, “The Supplied Sample Applications”
  This appendix describes the sample code provided with this document, and gives instructions for loading the sample programs onto your system.

Related Publications

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

• *TCP/IP Sockets Interface for CICS Users Guide*, GC31-7015
• *TCP/IP Version 2 Release 2.1 for MVS: Planning and Customization*, SC31-6085.

International Technical Support Center Publications

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

• *Bibliography of International Technical Support Centers Technical Bulletins*, GG24-3070.

The following International Technical Support Center publications contain a more detailed discussion of some of the topics covered in this document:

• *TCP/IP Tutorial and Technical Overview*, GG24-3376
• *CICS Transaction Design: Pseudo-Conversational or Conversational* GG24-3805.
Acknowledgments

This document is the result of a residency project run at the International Technical Support Center - San Jose during early 1993.

The project was designed and managed by:

Hugh Smith International Technical Support Center, San Jose.

This document was written by:

Ivo Larizzatti IBM Brazil

Gary Taylor IBM United Kingdom Laboratories.

Thanks to the following people for the invaluable advice and guidance provided in the production of this document:

Robert Stephenson IBM Raleigh

Larry Garrettson IBM Raleigh

Al Dixon IBM United Kingdom Laboratories.

In addition I would like to thank those who took the time to review and edit this document, especially:

Bob Yelavich Dallas Systems Center

Heinz Klein IBM Brazil

Karin Pankrath IBM Germany.
Chapter 1. Introduction

In this chapter we provide a brief description of the CICS to TCP/IP Sockets Interface. We introduce some of the terminology we will be using as we discuss the CICS to TCP/IP Sockets Interface. We compare the CICS to TCP/IP Sockets Interface with other IBM products that can be used to allow communication between transmission control protocol/internet protocol (TCP/IP) applications and CICS systems.

The topics we cover are:
- CICS and TCP/IP
- Introduction to TCP/IP
- What Is a Socket?
- Alternatives to using the CICS to TCP/IP Sockets Interface
- TCP/IP and CICS intercommunication facilities.

1.1 CICS and TCP/IP

Many companies have reached a point where, to support their business operations, they need to run applications that are geographically dispersed and involve both hardware and software from many vendors. The challenge for such companies is to make all of the information stored in any one of its systems available to the other components of the system in as transparent a manner as possible.

One of the common ways of linking these different systems is to use TCP/IP. TCP/IP allows applications on dissimilar computers to communicate with each other. One of the other attributes offered by TCP/IP is that it has worldwide potential. The Internet is the common worldwide TCP/IP network. Note that the Internet (with an uppercase I) is different from an internet.

The CICS to TCP/IP Sockets Interface (part of TCP/IP Version 2 Release 2.1 for MVS) makes it easier to integrate CICS based applications running under MVS with other systems that are using TCP/IP. There are some other alternatives to using the Sockets Interface, see Section 1.4, “Alternatives to Using the CICS to TCP/IP Sockets Interface” on page 4.

The Sockets Interface is a set of programs provided by IBM that allow you to write applications programs that will run under CICS in an MVS environment and communicate with other TCP/IP applications. Chapter 2, “CICS to TCP/IP Sockets Interface” on page 7 describes in more detail how the Sockets Interface is actually implemented.

1.2 Introduction to TCP/IP

This section contains a brief introduction to TCP/IP for those who have not used it before. For a full description, refer to the TCP/IP Tutorial and Technical Overview, GG24-3376.

TCP/IP is a suite of protocols that allow application-to-application (or process-to-process) communication across a set of interconnected data...
networks. The interconnected data networks are more usually called internets
(or sometimes internetworks). TCP/IP also has a set of conventions for forming
an internet and routing traffic over it.

Because TCP/IP consists of a widely accepted suite of protocols, it is often used
for distributed computing in a multivendor environment. Applications are built to
allow each component to be used to its best advantage. Large mainframes may
be used for data storage or for volume or specialized printing operations, while
desktop systems may be used to display graphics output, provide personal
services, and perform operations that suit the specific design of the hardware
and software.

1.2.1 The TCP/IP Protocol Suite

Figure 1 shows the layers in the TCP/IP protocol suite.

```
Application Layer
      |                      Sockets Interface
      |                      |
High-Level Protocol
      Layer

Internet Layer

Network Access Layer
```

*Figure 1. TCP/IP Protocol Layers*

1.2.2 The Application Layer

Standard application protocols exist that allow you to implement some functions
without having to do your own application programming. These protocols
include:

- **SMTP** The Simple Mail Transfer Protocol (allows electronic mail
  communication)
- **FTP** File Transfer Protocol (allows files to be moved from one environment
to another)
- **TELNET** A protocol that allows remote terminal access.

While these application protocols are useful, they do not cover all application
requirements. To be able to create our own applications that meet these
requirements we need to have a programming interface to TCP/IP.
1.2.3 The CICS to TCP/IP Sockets Interface

TCP/IP provides an application programming interface (API)—the Sockets API—for use if the standard protocols do not meet your application requirements. The CICS to TCP/IP Sockets Interface brings this programming interface to CICS systems running under MVS, thus allowing TCP/IP applications in other processors to interoperate with CICS applications. The CICS system appears as multiple sockets applications, permitting multiple peer-to-peer applications to execute concurrently and transparently. The CICS to TCP/IP Sockets Interface is a significant step forwards in integrating SNA host systems and data into the multiprotocol environment.

CICS programs that use the Sockets Interface can be written in COBOL, Assembler, or C/370 languages. The Sockets Interface also provides an ASCII-EBCDIC conversion routine, an EBCDIC-ASCII conversion routine, and a Listener function to listen for and accept connection requests, and start the appropriate CICS transactions to service those requests.

When using the CICS to TCP/IP Sockets Interface you will usually write a pair of programs, one to run on either side of the TCP/IP network. In general these programs will either be clients (in that they make a request of the other program), or servers (in that they service the request). We will use the terms “client” and “server” in this sense throughout this document. The CICS programs that use the CICS to TCP/IP Sockets Interface may be either clients or servers.

1.2.4 The High-Level Protocol Layer

Transmission control protocol (TCP)—provides a reliable connection-oriented communication protocol over the internets. TCP guarantees that any data we send will arrive at the other end, in the correct order and complete.

1.2.5 The Internet Layer

Internet protocol (IP) — provides very fast connectionless data transfer between two end points in a network. Since the data transfer provided by IP is not guaranteed (delivery is on a best efforts basis) we need the higher level TCP to ensure delivery of data.

1.2.6 The Network Access Layer

The network access layer provides the interface to the network hardware. TCP/IP can use many network interfaces. For example the network access layer could provide an interface to a local area network (LAN), or an X.25 network.
1.3 What Is a Socket?

A TCP/IP application resides at an end point in an internet. Each end point in an internet has a unique address:

- The IP address of the host system on which it resides
- The port number on that host through which it can be accessed.

The complete description of a connection between two processes is called an association. It consists of:

- The protocol being used
- The IP address of the local host
- The local port number
- The IP address of the foreign (or remote) host
- The foreign port number.

A half-association consists of:

- The protocol being used
- The IP address of one host
- The port number being used on that host.

This half association is also known as a socket. For the Sockets Interface, the protocol being used is always TCP.

Server programs identify themselves to TCP/IP by using a BIND command (saying that they are now available on a particular port number of a host system). Client programs identify which server they want to use by specifying the IP address of the server’s host, and the port number on that host at which the server is bound.

For a complete description of ports and sockets refer to the TCP/IP Tutorial and Technical Overview (GG24-3376).

1.4 Alternatives to Using the CICS to TCP/IP Sockets Interface

CICS is available on many different platforms that can use TCP/IP to communicate with other applications. For example, CICS OS/2* allows us to run CICS on a PS/2*, and CICS/6000* allows us run CICS on a RISC System/6000*. Since AIX* and OS/2* support TCP/IP, it would be possible to implement a system like the one shown in Figure 2 on page 5.
In both these cases, it is possible to write a sockets application that will communicate with CICS (either CICS OS/2 or CICS/6000), and then use CICS intercommunication facilities to communicate over an SNA link with another member of the CICS family (this could be CICS/ESA*, CICS/MVS*, CICS/VSE*, or CICS/400* as well as another CICS OS/2 or CICS/6000 system).

In the case of CICS OS/2, you need to bridge between the TCP/IP application and CICS OS/2 using the CICS OS/2 external call interface (ECI). CICS/6000, on the other hand, supports the sockets API directly.

### 1.5 TCP/IP and CICS Intercommunication Facilities

The CICS to TCP/IP Sockets Interface only allows you to write application programs that use the Sockets API to communicate with other TCP/IP applications. You still have to use SNA protocols to connect CICS/ESA or CICS/MVS to other members of the CICS family of products if you want to use any of the standard CICS intercommunication functions:

- Transaction routing
- Function shipping
- Distributed program link
- Distributed transaction processing
- Asynchronous processing.
CICS/6000 is the only member of the CICS family of products that can use a TCP/IP network for the CICS intercommunication functions, and then only when communicating with another CICS/6000 system. ACF/VTAM* Version 3 Release 4.2 multiprotocol transport feature (MPTF) allows the VTAM* flows for CICS ISC to be carried over a TCP/IP network (but in this case CICS still thinks that the connection is an SNA one, not a TCP/IP one).
Chapter 2. CICS to TCP/IP Sockets Interface

In this chapter we describe the structure of the CICS to TCP/IP Sockets Interface and its major components, and describe what occurs when you make a call to the Sockets Interface from your CICS program.

The topics we cover are:
- Structure of the CICS to TCP/IP Sockets Interface
- Components of the CICS to TCP/IP Sockets Interface
- Calls to the Sockets Interface.

2.1 Structure of the CICS to TCP/IP Sockets Interface

TCP/IP Version 2 Release 2.1 for MVS operates in multiple address spaces. One address space contains the code for TCP, User Datagram Protocol (UDP), IP, the sockets interface, and the support for TELNET. This address space is usually called the TCPIP address space, and is the name we use for it throughout this document.

2.1.1 Interface between CICS and the TCPIP Address Space

To use TCP/IP from a CICS application program, we need to provide an interface between the CICS address space and the TCPIP address space. This is done using the CICS task-related user exit (TRUE) mechanism, a standard way of allowing CICS application programs under MVS to access resources that do not belong to CICS. For more information on TRUEs see the Customization Guide for your release of CICS. Any implementation of a TRUE mechanism (also called an adapter) always consists of the following:
- A stub program, which intercepts calls from your CICS application program
- A TRUE program, which translates the call into a form acceptable to the non-CICS resource manager, in this case the TCPIP address space
- Administrative routines, which are used to enable (initialize) and disable (terminate) the mechanism.

2.1.2 TRUE Implementation for the CICS to TCP/IP Sockets Interface

As part of the implementation of the TRUE mechanism for the CICS to TCP/IP Sockets Interface, we also have an extra component in the TRUE mechanism. This is TCP/IP enabling code running in the CICS address space. This code runs under separate MVS subtasks, rather than as part of the main CICS task control block (TCB). These subtasks invoke TCP/IP functions by making MVS program calls (PCs) to the Inter-User Communication Vehicle (IUCV) interface of TCP/IP Version 2 Release 2.1 for MVS.

The subtasks are required because the TCPIP address space puts the calling TCB (the TCB under which the PC calls to IUCV were made) into a wait. The PC calls cannot be made from under the CICS TCB (if they were, all activity in the CICS region would have to wait until the call to TCPIP completed).
A separate subtask is created for each CICS task that uses the interface, and each subtask has a unique identifier. These MVS subtasks are reentrant and reusable.

In the CICS to TCP/IP Sockets Interface, program EZACIC03 provides the function of the subtask that handles the communication between the CICS address space and the TCPIP address space. The subtask is attached by the CICS TRUE program (EZACIC01) during the processing of the first call to the stub (EZACICAL) from any application program issuing socket calls.

Figure 3 shows the subtasks in relation to the CICS TCB in the CICS address space and TCP/IP for MVS interface.

---

**Figure 3. CICS to TCP/IP Sockets Interface Subtasks**

Each subtask remains attached until one of the following occurs:

- The CICS socket task is detached.
- The transaction to disable the Sockets Interface is invoked.
- CICS is shut down.
- An error is returned from TCP/IP after a socket call is made.

The subtask is detached and MVS frees up all associated storage for that subtask if any of the above events occurs.
2.2 Components of the CICS to TCP/IP Sockets Interface

Figure 4 shows the main components of the CICS to TCP/IP Sockets Interface.

![Diagram of components]

**UTILITY SUBROUTINES**
EZACIC04, EZACIC05, EZACIC06

**LISTENER USER WRITTEN USER**
EZACIC02 SERVER CLIENT

**STUB (EZACICAL) TIE**

**TASK RELATED USER EXIT**
EZACIC01

**DISABLE EZACIC00**

**GLOBAL WORK AREA (GWA)**

**IUCV / TCP/IP**

![Figure 4. Main Components of the CICS to TCP/IP Sockets Interface](image_url)
The main components of CICS to TCP/IP Sockets Interface are the following:

- EZACIC00 is the program that IBM supplies to enable and disable the Sockets Interface. We discuss the use of this program to enable and disable the Sockets Interface more fully in Chapter 4, “Operating the CICS to TCP/IP Sockets Interface” on page 33.
- The TRUE program (EZACIC01) provides the interface between CICS tasks and MVS subtasks. It can be invoked by an application program. CICS will invoke it during task termination or shutdown.
- The global work area (GWA) is used by the TRUE. It contains information such as the MVS version and release, CICS version and release, TCPIPIP address space job name, Listener port number, and adapter status.
- Task interface element (TIE) contains the areas used for communication between the CICS task, the TRUE, and the subtasks.
- MVS subtask and exit (EZACIC03) handles the connection with the TCP/IP address space, validates the socket call function, and establishes addressability to the GWA and TIE.
- The adapter stub (EZACICAL) intercepts the socket calls from the user programs, and passes them to the TRUE program.
- The Listener program (EZACIC02) establishes a path to the TCPIPIP address space, listens for connection requests from clients, accepts the requests, and starts CICS transactions to service the requests. It is an example of a concurrent server, and is supplied by IBM as part of the CICS to TCPIPIP Sockets Interface. We discuss concurrent servers in Chapter 5, “Application Design Considerations” on page 41.
- The utility subroutines (EZACIC04, EZACIC05, and EZACIC06) are used for converting from EBCDIC to ASCII, ASCII to EBCDIC, and to convert a COBOL array of character variables to a bit-mask array (the bit-mask array is used by the SELECT socket call).
- EZACIC07 is the C language interface module. It intercepts socket calls from C language programs, and then passes them to the TRUE program through EZACICAL.
- Your own client and server programs are the final components of the CICS to TCP/IP Sockets Interface.

2.3 Calls to the Sockets Interface

Each time a CICS application program makes a call to the Sockets Interface to use a TCP/IP function, the stub program EZACICAL is called. This stub invokes the TRUE program (EZACIC01) using the DFHRMCAL macro.

2.3.1 Processing the Initial Call

The first call that a CICS application program makes to the Sockets Interface is either an INITAPI or a TAKESOCKET call. These calls register the CICS task with TCPIPIP as a sockets application. The first call invokes the TRUE, which attaches an MVS subtask (EZACIC03) for the task and synchronizes CICS task management with MVS subtask management using event control blocks (ECBs).

TCP/IP allocates a minimum of 50 paths from the task to the socket when the CICS task registers itself. These paths themselves are often called sockets.
this document we use the term socket descriptor to refer to a path, although strictly socket descriptor means the number that a TCP/IP sockets application uses to address a path.

Transactions that issue the INITAPI call can specify the number of socket descriptors they want to use by changing the INITAPI-MAXSOC parameter. The maximum allowed value is 5000, and the minimum is 50. The Listener asks for 50 socket descriptors.

A socket transaction that issues a TAKESOCKET as its first socket call is given 50 socket descriptors. The first of these socket descriptors is used for the conversation with the client. Therefore, the application has 49 socket descriptors left to use.

2.3.2 Processing Subsequent Calls

Each subtask performs the following functions:

- Validates the socket call
- Converts the socket call to an IUCV format
- Sends the request to TCPIP using IUCV
- Receives the response from TCPIP
- Builds the socket call response from the IUCV response.

After performing these functions, the subtask makes the information available to the user program.
Chapter 3. Implementing the CICS to TCP/IP Sockets Interface

In this chapter we describe the steps required to implement the CICS to TCP/IP Sockets Interface. This discussion assumes that both CICS and TCP/IP Version 2 Release 2.1 for MVS are already installed and operating on MVS.

The topics we cover are:

• Maintenance required on TCP/IP Version 2 Release 2.1 for MVS
• Changes to the CICS startup JCL
• Definitions needed in CICS
• Changes to COBOL compilation procedures
• Changes to C/370 compilation procedures
• Definitions needed in TCPIP data sets.

Naming Convention. Throughout this document we have used tcpip as a replacement for the high-level qualifier for the TCP/IP data set names. For example, in our system we have a data set called TCPIP.V2R2M1.SEZATCP; in the text we refer to this as tcpip.SEZATCP. Where you see tcpip, you should replace it with your own high-level qualifier.

3.1 Maintenance Required on TCP/IP Version 2 Release 2.1 for MVS

You should ensure that program temporary fixes (PTFs) UN42952, UN43356 and UN40677 (or any later PTFs that replace them) have been applied to your TCP/IP Version 2 Release 2.1 for MVS system.
3.2 Changes to the CICS Startup JCL

Figure 5 shows the JCL we used to start our CICS region during our tests. The changes that we had to make are highlighted.

```
//CICSRS2A JOB (999,POK),¢ARIZZATTI¢,CLASS=A,MSGCLASS=T,
    NOTIFY=&SYSUID,MSGLEVEL=(1,1)
//CICS EXEC PGM=DFHSIP,REGION=32M,TIME=1440,
    PARM=SYSIN
//SYSIN DD *
SIT=6S,
START=AUTO,
DCT=IP,
GRPLIST=TCPLIST,
GMTEXT="WELCOME TO CICS/ESA V3.3.0 WITH TCP/IP SOCKETS INTERFACE",
APPLID=SCMCICSA
.END
//DFHXRMSG DD DISP=SHR,DSN=CICS330.CNTL.CICS.DFHXRMS
//DFHXRCTL DD DISP=SHR,DSN=CICS330.CNTL.CICS.DFHXRCTL
//STEPLIB DD DISP=SHR,DSN=CICS330.SDFHAUTH
    DD DISP=SHR,DSN=SYS1.COBOL.V1R3M2.COB2CICS
    DD DISP=SHR,DSN=COBOL.V1R3M2.COB2LIB
//DFHRPL DD DISP=SHR,DSN=CICS330.SDFHLOAD
    DD DISP=SHR,DSN=SYS1.COBOL.V1R3M2.COB2CICS
    DD DISP=SHR,DSN=COBOL.V1R3M2.COB2LIB
//DFHTEMP DD DISP=SHR,DSN=CICS330.CNTL.CICS.DFHTEMP
//DFHINTRA DD DISP=SHR,DSN=CICS330.CNTL.CICS.DFHINTRA
//LOGUSR DD SYSOUT=*,DCB=(DSORG=PS,RECFM=V,BLKSIZE=136)
//MSGUSR DD SYSOUT=*,DCB=(DSORG=PS,RECFM=V,BLKSIZE=136)
//TCPDATA DD SYSOUT=*,DCB=(DSORG=PS,RECFM=V,BLKSIZE=136)
```

Figure 5. CICS Startup JCL with CICS to TCP/IP Sockets Interface

- 1 - You must concatenate the data set tcpip.SEZALINK to DFHRPL. This data set contains the modules EZACIC03 and IUCVMULT. See Section 3.3.2, “Program Definitions” on page 17 for more details.

- 2 - You must concatenate the data set tcpip.SEZATCP to DFHRPL. This data set contains all the others modules required by CICS to execute the Sockets Interface. See Section 3.3.2, “Program Definitions” on page 17 for more details.

- 3 - The IBM-supplied Listener transaction writes messages to a transient data queue (TDQ) called TCPM during startup. If you use the Listener, you must add an entry to the CICS destination control table (DCT) for this queue. If you implement TCPM as an extra-partition queue (as we recommend), you must also add a data definition (DD) statement, like the one shown in
3.3 Definitions Needed in CICS

In this section we show the definitions that you need to make if you want to use the IBM-supplied programs and transactions. You may want to replace some of these programs and transactions with your own versions in the future.

To be able to use the Sockets Interface you need to define some extra resources to CICS. These definitions include:

- Transactions
- Programs
- BMS Mapset
- Transient data queues.

You can start the CICS to TCP/IP Sockets Interface automatically when CICS is initialized. One method of doing this is to use a sequential terminal. In this case, you also need to define a sequential terminal in the Terminal Control Table (TCT). See Section 4.1.2, “Automatic Enable” on page 34 for more details.

3.3.1 Transaction Definitions

Figure 6 on page 16, Figure 7 on page 16 and Figure 8 on page 17 show the entries required in CICS Resource Definition Online (RDO) to define the three transactions that IBM supplies with the Sockets Interface. These transactions are:

- **CSKD**—used to disable the Sockets Interface
- **CSKE**—used to enable the Sockets Interface
- **CSKL**—the Listener.
<table>
<thead>
<tr>
<th>Define</th>
<th>Transaction : CSKD</th>
<th>ALIASES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td>TCP/IP</td>
<td>Alias ==&gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>Description =&gt; Disable Sockets Interface</td>
<td>TASKReq =&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program =&gt; EZACICO0</td>
<td>XTRanid =&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Twasize =&gt; 00000</td>
<td>TPName =&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Profile =&gt; DFHCICST</td>
<td>XTPname =&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partitionset =&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Status =&gt; Enabled</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Primedsize : 00000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Taskdataloc ==&gt; Below</td>
<td>RECOVERY =&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Taskdatakey ==&gt; User</td>
<td>DTimout =&gt; No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remote Attributes</td>
<td>Indoubt =&gt; Backout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic =&gt; No</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remotesystem =&gt;</td>
<td>SPurge =&gt; No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remotename =&gt;</td>
<td>TFUrgo =&gt; No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tprof =&gt;</td>
<td>DUmp =&gt; Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Localq =&gt;</td>
<td>TRACE =&gt; Yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 6. RDO Definition of the CSKD Transaction**

<table>
<thead>
<tr>
<th>Define</th>
<th>Transaction : CSKE</th>
<th>ALIASES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td>TCP/IP</td>
<td>Alias ==&gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>Description =&gt; Enable Sockets Interface</td>
<td>TASKReq =&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program =&gt; EZACICO0</td>
<td>XTRanid =&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Twasize =&gt; 00000</td>
<td>TPName =&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Profile =&gt; DFHCICST</td>
<td>XTPname =&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partitionset =&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Status =&gt; Enabled</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Primedsize : 00000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Taskdataloc ==&gt; Below</td>
<td>RECOVERY =&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Taskdatakey ==&gt; User</td>
<td>DTimout =&gt; No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remote Attributes</td>
<td>Indoubt =&gt; Backout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic =&gt; No</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remotesystem =&gt;</td>
<td>SPurge =&gt; No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remotename =&gt;</td>
<td>TFUrgo =&gt; No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tprof =&gt;</td>
<td>DUmp =&gt; Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Localq =&gt;</td>
<td>TRACE =&gt; Yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7. RDO Definition of the CSKE Transaction**
3.3.2 Program Definitions

We must define some of the programs that make up the Sockets Interface to CICS. Some of the programs are part of the Sockets Interface itself, and some provide supporting function. These programs are supplied by IBM in tcpip.SEZATCP, and so SEZATCP must be added to the DFHRPL concatenation for your CICS system (see Figure 5 on page 14).

The programs we must define to CICS are listed below:

**EZACIC00** is the connection manager program used by the IBM-supplied transactions CSKE and CSKD to enable and disable the CICS to TCP/IP Sockets Interface.

**EZACIC01** is the TRUE program.

**EZACIC02** is the Listener program used by the transaction CSKL. This transaction is started when you enable the CICS to TCP/IP Sockets Interface using the CSKE transaction.

**EZACICM** is a BMS mapset that contains all the maps used by the transactions that enable and disable the CICS to TCP/IP Sockets Interface.

The following programs do not need to be defined to CICS, but they do have to exist in a data set concatenated to DFHRPL. These programs are supplied by IBM in tcpip.SEZATCP.

---

**Figure 8. RDO Definition of the CSKL Transaction**

We explain the function of CSKE, CSKD and CSKL in more detail in Chapter 4, “Operating the CICS to TCP/IP Sockets Interface” on page 33.
EZACIC04 is the program used to convert EBCDIC data within CICS to the ASCII format required in TCP/IP networks and workstations.

EZACIC05 is the program used to convert ASCII data coming from the TCP/IP network to the EBCDIC format required by CICS.

EZACIC06 is the program used to convert a COBOL array of character variables into a bit-mask array, which is used by the SELECT call. It also converts the bit-mask array back into a COBOL array.

EZACIC07 is the C language interface program.

EZACICAL is the application stub that invokes the TRUE and passes on the CICS application’s socket call.

Two additional programs that must be available to CICS are in tcpip.SEZALINK. SEZALINK must be added to the DFHRPL concatenation for your CICS system (see Figure 5 on page 14).

EZACIC03 is the program that passes data between the CICS sockets task and the IUCV interface of TCP/IP Version 2 Release 2.1 for MVS.

IUCVMULT is loaded by EZACIC03, and maintains a list of paths to the TCPIP address space.

The following two optional programs are supplied as samples in tcpip.SEZATCP:

EZACICSS is a sample server that comes with TCP/IP. It establishes the connection between CICS and TCPIP, and receives client requests from workstations.

EZACICSC is another sample server that works with the Listener provided by IBM (EZACIC02).

Figure 9, Figure 10 on page 19 and Figure 11 on page 19 show the program definitions needed if you are using the IBM-supplied transactions.

CEDA DEFINE
  PROGRAM : EZACIC00
  Group : TCPPI
  Description ==> Connection Manager
  Language ==> Assembler
  Reload ==> No
  Resident ==> No
  Usage ==> Normal
  USELpacopy ==> No
  Status ==> Enabled
  RS1 : 00
  Cedf ==> Yes
  Datolocation ==> Below
  EXE CKey ==> User
  REMOTE ATTRIBUTES
    REMOTESystem ==> 
    REMOTEName ==> 
    Transid ==> 
    EXECUtionset ==> Fullapi

Figure 9. Definition of Program EZACIC00
The two program definitions shown in Figure 12 on page 20 and Figure 13 on page 20 are optional. They are the definitions for the sample socket programs that are supplied with the Sockets Interface.
3.3.3 BMS Mapset Definition

If you are using CSKE and CSKD to enable and disable the Sockets Interface, you need to define mapset EZACICM as shown in Figure 14 on page 21. This is the mapset used by EZACIC00, and contains the maps you can see in Figure 24 on page 33, Figure 31 on page 37 and Figure 34 on page 38.
3.3.4 Transient Data Queues

Figure 15 shows the entries used in the CICS DCT to define the TCPM TDQ for the Sockets Interface. They are needed if you use the IBM-supplied Listener transaction.

```c
DFHDCT TYPE=SDSCI, X
  BLKSIZE=136, X
  DSCNAME=TCPDATA,
  RECFORM=VARUNB, X
  RECSIZE=132, X
  TYPEFILE=OUTPUT
  ...
...
DFHDCT TYPE=EXTRA, X
  DESTID=TCPM, 2 X
    DSCNAME=TCPDATA
  ...
...
DFHDCT TYPE=INTRA, X
  DESTID=SR11, X
  DESTFAC=FILE, 3 X
  TRIGLEV=1, X
  TRANSID=SR11
  ...
...
```

Figure 15. DCT Entries Required by the Listener

The Listener writes to the TCPM queue while the Sockets Interface is enabled. Your own sockets applications can also write to this queue using EXEC CICS WRITEQ TD commands. We recommend that you define TCPM as an extra-partition TDQ (Figure 15 1 and 2) so that any messages produced by the Sockets Interface can be viewed from outside the CICS region. Remember that you must also have a DD statement for this extra-partition TDQ. Note 3 in Figure 5 on page 14 shows the entry for TCPDATA, which is an example of the DD statement you could use.

The IBM-supplied Listener transaction can start a child server using a TDQ (for more information see Chapter 5, “Application Design Considerations” on page 41). If you want to use this method of starting child servers you must define the queues you will be using in the DCT Figure 15 3 shows the entry we used for a sample child server, SISSRI1C, that is started by this method.
Because SRI1 is the transaction identifier used to invoke this child server, it must be named in the TRANSID parameter of the DCT entry.

### 3.3.5 Terminal Definition

To automate the startup of the Sockets Interface using the IBM-supplied transaction CSKE, you need to define a sequential terminal to CICS. Figure 16 shows the entries required in the CICS Terminal Control Table (TCT) to have the CICS to TCP/IP Sockets Interface activated automatically following CICS initialization. This is explained in more detail in Section 4.1.2, “Automatic Enable” on page 34.

```plaintext
TCTIP
TITLE *DFHTCTIP - CICS SAMPLE TERMINAL CONTROL TABLE - CRLP*
DFHTCT TYPE=INITIAL,
  SUFFIX=IP, X
  STARTER=YES, ALLOWS $ IN SUFFIX X
  ACCMETH=(NONVTAM,VTAM) ALL ACCESS METHODS
  *
  TITLE *DFH$TCTS - COPYBOOK OF TCT ENTRIES FOR SEQUENTIAL (CRLPX)
       TERMINAL* 
***********************************************************************
* MODULE NAME = DFH$TCTS *
* DESCRIPTIVE NAME = Sample TCT Entries for Sequential Terminal *
*---------------------------------------------------------------------*
* DESCRIPTION : *
* THIS MEMBER CONTAINS THE FOLLOWING BSAM ENTRIES: *
* TRMDNT DSCNAME TERMINAL TYPE *
* ------- ------- ------------------- *
* SAMA TCPIN 2540 CARD READER *
* TCPOUT 1403 LINE PRINTER *
*---------------------------------------------------------------------*
* *
  DFHTCT TYPE=SDSCI, X
  DEVICE=2540, X
  DSCNAME=TCPIN *
  *
  DFHTCT TYPE=SDSCI, X
  DEVICE=1403, X
  DSCNAME=TCPOUT *
  *
  DFHTCT TYPE=LINE, X
  ACCMETH=BSAM, X
  TRMTYPE=CRLP, X
  INAREAL=80, X
  ISADSCN=TCPIN, X
  OSADSCN=TCPOUT *
  *
  DFHTCT TYPE=TERMINAL, X
  TRMDNT=SAMA, X
  LPLEN=80, X
  PGESIZE=(24,80), X
  ERRATT=NO, X
  TRMSTAT=TRANSACTION
  DFHTCT TYPE=FINAL
  END ,
```

*Figure 16. Sample TCT for Sequential Terminal Definition*

If you want to automate the Sockets Interface in this way, you must also check the value specified for end of data input (EODI) in the CICS system initialization table (SIT). If there is no specification for EODI then your system is using the default value, E0 (the hexadecimal representation of \). This character is used to
mark the end of data input, and acts as a logical representation of using the Enter key. If you do not want to use \, or if your installation has already used a different value for EODI, you need to change the sample input as shown in Figure 25 on page 34.

3.4 Changes to COBOL Compilation Procedures

To write CICS programs that contain calls to the CICS to TCP/IP Sockets Interface in VS COBOL II, you need to modify your procedure for translating, compiling and link-editing the program.

Figure 17 on page 24 shows DFHEITVL, which is the IBM-supplied procedure for performing a translation, compilation and link-edit of a CICS program written in VS COBOL II. The areas where DFHEITVL needs to be changed are highlighted. The notes (shown as 1 and 2 ) allow you to compare this procedure with the modified one shown in Figure 19 on page 26.

The procedure contains four steps:
1. TRN translates the COBOL program.
2. COB compiles the translated COBOL program.
3. COPYLINK reblocks module DFHEILIC to be used by LKED.
4. LKED link-edits the final module to a LOADLIB.
CICS/ESA and TCP/IP for MVS

Figure 17. DFHEITVL - Unmodified JCL for Compilation of a COBOL Program
If you use DFHEITVL in standard form, the problem shown in Figure 18 will occur.

This error occurs because the external symbol DFHEI1 has already been defined in module EZACICAL. To prevent this problem, use the procedure shown in Figure 19 on page 26.
The following modifications must be made to the compilation procedure:

- The COPYLINK step (1) must be removed.
- The concatenation of $DSN=&&COPYLINK$ in the SYSLIN DD of the LKED step (2) must be removed.

These modifications eliminate the error, enabling the linkage-editor to execute correctly, as shown in Figure 20 on page 27.
3.5 Changes to C/370 Compilation Procedures

To compile a C/370 program that contains calls to the Sockets Interface, you need to change the standard procedure for C/370 compilation (DFHEITDL) that is provided with CICS.

Make the following modifications:

- Add these data sets to the SYSLIB DD statement for the C jobstep (see 1 in Figure 21 on page 28):
  - tcpip.SEZATCP
  - tcpip.SEZACMAC
  - tcpip.SEZACMTX

- Add these data sets to the SYSLIB DD statement for the LKED jobstep (see 2 in Figure 21 on page 28):
  - tcpip.SEZATCP
  - tcpip.SEZACMAC
  - tcpip.SEZACMTX

- Add an INCLUDE for the module EZACIC07 to the SYSIN DD * statement in the LKED jobstep (see 3 in Figure 21 on page 28):
Figure 21. Modified JCL for Compilation of a C/370 Program
3.6 Definitions Needed in TCPIP Data Sets

To implement the Sockets Interface we need to make definitions in two TCPIP data sets. These are:

- `tcpip.PROFILE.TCPIP`
- `tcpip.TCPIP.DATA`

For more detailed information about the function of these data sets, refer to *TCP/IP Version 2 Release 2.1 for MVS: Planning and Customization*.

3.6.1 The PROFILE Data Set

To define the CICS region to TCP/IP for MVS, you have an ALLOWAPPL entry in the BEGINVTAM statement of the `tcpip.PROFILE.TCPIP` data set that allows CICS to access the TCPIP address space. If you want to reserve ports for CICS, then you should make entries in the PORT statement of the `tcpip.PROFILE.TCPIP` data set.
This is a sample configuration file for the TCPIP address space. For more information about this file, see `Configuring the TCPIP Address Space` and `Configuring the Telnet Server` in the Planning and Customization Manual.

```plaintext

; TCP/IP PROFILE

; Reserve PORTs for the following servers.

; NOTE: A port that is not reserved in this list can be used by any user. If you are have TCP/IP hosts in your network that reserve ports in the range 1-1023 for privileged applications, you should reserve them here to prevent users from using them.

PORT

3000 TCP CICSRS2A ; Socket reserved for CICSRS2A
3015 TCP CICSRS2B ; Socket reserved for CICSRS2B

; Define the VTAM parameters required for the TELNET server

BEGINVTAM

ALLOWAPPL * ; Allow all applications that have not been specified to be accessed

ENDVTAM
```

Figure 22. Definition of the TCP/IP Profile

1 - PORT statement

This statement specifies the ports that are reserved for each application. In our environment we defined two entries, one for CICSRS2A (port number 3000) and other for CICSRS2B (port number 3015). CICSRS2A and CICSRS2B are the jobnames of our CICS regions.

These two entries reserve port 3000 for exclusive use by CICSRS2A and port 3015 for exclusive use by CICSRS2B. Other MVS jobs that want to access TCP/IP on MVS are prevented from using these ports (for example, a CICS sockets application in CICSRS2A cannot issue a BIND call for port 3015). If you use the standard supplied transaction CSKE for starting the CICS to TCP/IP Sockets Interface, and you try to use a port that is reserved for another application, the transaction CSKL will abend with code BIND.
Ports that are not defined in the PORT statement can be used by any application, including CICSRS2A and CICSRS2B (even if they are already using ports that have been reserved for them).

2 - BEGINVTAM statement

In this statement you must define either an "ALLOWAPPL " entry (which allows all applications to access the TCPIP address space), or specific "ALLOWAPPL" entries for those applications that can access TCPIP. In our environment, we used "ALLOWAPPL ".

3.6.2 The DATA Data Set

The DATA data set must have a entry for the name of the started procedure used to start the TCPIP address space. In our system, we used the default name TCPIP.

You must provide the TCPIPUSERID when you initialize the CICS to TCP/IP Sockets Interface using the CSKE transaction, as described in Chapter 4, “Operating the CICS to TCP/IP Sockets Interface” on page 33.

```
;
;***********************************************************************
; * Name of Data Set: tcpip.TCPIP.DATA *
; *
; This data, TCPIP.DATA, is used to specify configuration *
; information required by TCP/IP client programs. *
; *
;***********************************************************************
;
; TCPIPUSERID specifies the name of the started procedure which was *
; used to start the TCPIP address space. TCPIP is the default.
;
; TCPIPUSERID TCPIP

..............
..............
..............

;***********************************************************************
;
; End of file.
;
```

Figure 23. Definition of TCP/IP Data
Chapter 4. Operating the CICS to TCP/IP Sockets Interface

In this chapter we describe how to operate the CICS to TCP/IP Sockets Interface. The discussion assumes that you are using the IBM-supplied transactions to enable and disable the interface, and that you are using the Listener as your concurrent server.

The topics we cover are:
- Enabling the sockets interface
- Disabling the sockets interface.

4.1 Enabling the CICS to TCP/IP Sockets Interface

To allow CICS to communicate with TCP/IP on MVS, you must first enable the Sockets Interface. This can be done in one of the following ways:

- Manually by an operator entering the CSKE transaction
- Automatically at CICS initialization using the CICS sequential terminal facility.

**Note:** TCP/IP must be active in MVS.

4.1.1 Manual Enable

The CICS to TCP/IP Sockets Interface can be started manually by entering the CICS transaction CSKE.

This transaction invokes the program EZACIC00, which sends the map shown in Figure 24 to the terminal:

```
CSKE
CICS TASK-RELATED USER EXIT
CONNECTION MANAGER
ENABLE CICS-TCP/IP API

TCPME00
PF1=HELP PF3=QUIT
```

*Figure 24. CSKE - Panel for Manual Enable of the Sockets Interface*
To enable the Sockets Interface, you must enter values for TCPIPUSERID and PORT:

**TCPIPUSERID** The name of the TCPIP address space (as defined in tcpip.TCPIP.DATA)

**PORT** The port number required by the Listener (the port number specified in tcpip.PROFILE.TCPIP if you have reserved a port for this CICS system).

The program EZACIC00 determines if the interface is already active. If the interface is not active, EZACIC00 establishes the GWA for the Sockets Interface and issues a CICS enable for the TRUE. EZACIC00 sends the following message to the terminal:

- EXIT PROGRAM ‘EZACIC01’ (TRUE) ENABLED, TRANSACTION ‘CSKL’ STARTED.

If the interface is already active, EZACIC00 sends the following message to the terminal:

- EXIT PROGRAM ‘EZACIC01’ (TRUE) ALREADY ENABLED, ENABLE IGNORED

### 4.1.2 Automatic Enable

If you are using CSKE to enable the Sockets Interface, you can use the CICS sequential terminal facility to automate the process. Because CSKE expects to receive the port number and the TCPIPUSERID as terminal input, it is not possible to use a CICS program list table for post initialization (PLTPI) to automate CSKE.

To start the Sockets Interface automatically you must make the following definitions:

- Update the TCT to include a sequential terminal entry. See the example in Figure 16 on page 22.
- Include a DD statement for both the input and output terminals (TCPIN and TCPOUT) in the CICS startup JCL, as shown in Figure 25. Both of these data sets should have a record length of 80 characters.

```cics
//*----------------------------------------------------------------
//** SEQUENTIAL TERMINAL DATA
//*----------------------------------------------------------------
//TCPIN DD SYSOUT=*,DCB=BLKSIZE=80,RECFM=F
//TCPOUT DD SYSOUT=*,DCB=BLKSIZE=80,RECFM=F
//CSKE
//TCPIN 3000
/*
```

*Figure 25. Sample JCL for Automatic Enabling of the Sockets Interface*

You can now submit the CICS start up JCL, the Sockets Interface will be started automatically.

The information required in the TCPIN DD is as follows:

**CSKE** The CICS transaction code for enabling the Sockets Interface

**TCP** 3000 **TCP** is the TCPIPUSERID of the TCPIP region on MVS, and 3000 is the port to which the Listener will bind.
The four blanks between TCPIP and 3000 in this card are the field separator (FLDSEP) characters assembled with the ENABLE-API main map. These blanks must always be present to separate the TCPIPUSERID field from the PORT number field.

The backslash (\) is the default end-of-data indicator taken from the EODI option of the SIT. If your installation uses some other EODI, change this accordingly.

CICS/ESA will issue messages DFHTC2507 and DFHTC2500, placing the line out of service because of an input event rejection. You should ignore these messages. You could suppress them by adding an extra line to the JCL after the line that says TCPIP\ 3000. This additional line would contain CESF GOODNIGHT. If you were to do this, you would find that the Sockets Interface would not start automatically when you did a warm start of CICS/ESA, since the CESF GOODNIGHT command places the sequential terminal out of service. A warm start retains this status; since the terminal is out of service, no input is accepted from the device.

4.1.3 Messages Received When Enabling the Sockets Interface

If the Sockets Interface is enabled successfully, the following messages appear:

- The message shown in Figure 26 appears on the console.
- The message shown in Figure 27 appears in the TCPM queue (the queue that you defined to receive messages from the Sockets Interface, as described in Section 3.3.4, “Transient Data Queues” on page 21).
- The message shown in Figure 28 appears in the CSMT queue.

20.06.05 JOB31128 +DFHAP1204I SCMCICSA COBOL2 is being initialized.
20.06.05 JOB31128 +DFHAP1205I SCMCICSA C/370 is being initialized.
20.06.06 JOB31128 +DFHSI1517 SCMCICSA Control is being given to CICS.
20.06.07 JOB31128 +EZACIC01-001I SCMCICSA: EZACIC03 ATTACHED, TCB=008C8C60 TASK=0000026 OPID=?? TERM=???? TRAN=CSKL THU.03/11/93 20:06:08

Figure 26. Successful Enable of Sockets Interface (i)

03/10/93 18:24:39 CICS/TCPIP: LISTENER READY TO ACCEPT REQ'T THRU PORT : 3000

Figure 27. Successful Enable of Sockets Interface (ii)

EZACIC00-050I EXIT PROGRAM EZACIC01 ENABLED, TRAN=CSKL STARTED... OPID=?? TERM=SAMA TRAN=CSKE Thu.03/11/93 20:06:07
EZACIC01-001I SCMCICSA: EZACIC03 ATTACHED, TCB=008C8C60 TASK=0000026 OPID=?? TERM=???? TRAN=CSKL Thu.03/11/93 20:06:08

Figure 28. Successful Enable of Sockets Interface (iii)

4.1.4 Errors in Enabling the Interface

If the CSKL transaction abends with a code of BIND when you try to use CSKE to enable the interface, the jobname for your CICS region does not match the name specified for this port in tcpip.PROFILE.TCPIP. In other words, you are trying to use a port that has been reserved for another application. Messages similar to those shown in Figure 29 on page 36 appear in CSMT.
Similarly, if the CSKL transaction abends with a code of IAPI when you try to use CSKE to enable the interface, the value you have specified for TCPIPUSERID on the CSKE panel does not match the one you specified in tcpip.TCPIP.DATA. Messages similar to those shown in Figure 30 appear in CSMT.

4.2 Disabling the CICS to TCP/IP Sockets Interface

The Sockets Interface can be terminated manually by entering the CICS transaction CSKD.

When you enter CSKD, the panel in Figure 31 on page 37 is sent to the terminal. You have two options for disabling the CICS to TCP/IP Sockets Interface:

- A quiescent disable
- An immediate disable.
4.2.1 Quiescent Disable

A quiescent disable terminates the interface in an orderly fashion. It prevents any new tasks from using the Sockets Interface, but allows any existing tasks to terminate normally before the Sockets Interface is disabled. This is the normal way of disabling the Sockets Interface and should be used in most cases.

The following is the sequence of steps for a quiescent disable:

1. The following message is sent to the terminal requesting the disable:
   DISABLE-API HONORED AND DELAYED UNTIL ALL USING TASKS HAVE COMPLETED

2. Any new CICS tasks trying to use the CICS to TCP/IP Sockets Interface are denied access.

3. All existing CICS tasks using the CICS to TCP/IP Sockets Interface continue through to task termination.

4. CSKL terminates. This may take up to three minutes.

5. The message shown in Figure 32 on page 38 is sent to the CSMT TDQ.

6. The message shown in Figure 33 on page 38 is sent to the TDQ defined to receive the Sockets Interface messages (TCPM in our system) when all activity is complete and the Sockets Interface is disabled.
4.2.2 Immediate Disable

The immediate disable option should only be used in extreme situations, since it causes an abrupt termination of the interface. Therefore, if you choose option 2 (immediate disable) from the CSKD menu you will be asked to confirm that you want to do an immediate disable. Figure 34 shows the panel you are presented with if you choose an immediate disable. The highlighted message only appears if you select option 2.

If you reply Y (yes) to this question:

1. The following message is sent to the terminal requesting the disable:

   EXIT PROGRAM ‘EZACIC01’ (TRUE) DISABLED. AUDIT TRAIL IN TD QUEUE ‘CSMT’

2. CSKL is purged.

3. All CICS tasks are immediately denied access to the Sockets Interface.

   - Tasks that have previously connected to the Sockets Interface successfully abend with code AETA on the next socket call they make.
• New tasks trying to connect to the Sockets Interface abend with code AEY9 on the first socket call they make.

4. The message shown in Figure 35 is sent to CSMT.

```
EZACIC00-112W EXIT PROGRAM (EZACIC01) IMMEDIATELY DISABLED. OF ID=??? TERM=051C TRAN=CSKD FRI.03/12/93 20:25:22
```

Figure 35. Message for Immediate Disable of Sockets Interface
Chapter 5. Application Design Considerations

In this chapter we describe the design considerations for an application that uses the CICS to TCP/IP Sockets Interface. We discuss considerations related to the fact that we have a transaction processor as part of the system. We also discuss constructing distributed applications, and highlight the additional considerations for transferring variable-length records.

The topics we cover are:

• General considerations for CICS in a TCP/IP Network
• Distributed applications
• Choosing between iterative and concurrent servers
• Transferring data between client and server.

5.1 General Considerations for CICS in a TCP/IP Network

Integrating CICS and TCP/IP brings new considerations into the task of designing distributed applications. To make the most efficient use of CICS within the network, we must understand the ideas behind transaction processing. Four important topics for CICS are:

• Efficient application design—short and long-running tasks
• Recoverable resources—update commits and backouts
• Security in a Distributed System
• Design restrictions.

5.1.1 Efficient Application Design

Because CICS can run a large volume of transactions, it is important to be aware of the resources that each transaction will access. You should try to keep a transaction from using too many CICS resources for too long a period. The sooner a transaction frees up a resource, the earlier another transaction can access it.

Bottlenecks and poor performance in a CICS system are often caused by poor program and transaction design. Transactions that interface to users should typically be designed so that they do not wait during a user’s think time. If the transaction waits for user input, it will be using system resources (such as storage) during the user’s think time, without being productive. Users will often take several seconds, perhaps even minutes, to prepare their next input; in systems that may be handling high volumes of transactions, this is undesirable.

If the server is a CICS transaction, it should remain active only long enough to perform its function. It should not wait for the client, unless the next message from the client is expected soon (not the case if the client interfaces directly to a user).

Each CICS sockets transaction also uses a “path” to TCPIP on MVS; that is, the socket calls are executed under separate TCBs. There are a limited number of paths, and a long-running socket transaction will typically hold its path for a long period. Too many long-running sockets transactions in a CICS region may cause
problems because of insufficient free paths. However, short-lived socket transactions hold a path only while they are running, which is a relatively short period.

If you have written application code to run in a CICS system before, these considerations should be familiar to you since they usually cause you to use pseudo-conversational programming techniques in CICS, in preference to conversational techniques. *CICS Transaction Design: Pseudo-Conversational or Conversational* (GG24-3805-00) contains more information on these design considerations.

Although you should write short-lived transactions, you cannot use an EXEC CICS RETURN TRANSID command in your child server, so you cannot write truly pseudo-conversational applications.

### 5.1.2 Recoverable Resources

The ability to provide data consistency for recoverable resources is a basic attribute of CICS. In existing client-server applications, using LU6.2 or CICS multiregion operation (MRO), a synchronization protocol is used that allows both the client and server to be aware of the state of each other’s resources. Should any failure occur, such as a physical break in the link between the two systems, both client and server can back out updates to reach a common level from which they can safely proceed once the failure has been corrected. This means that when updating critical information, CICS is very robust.

The sockets application programming interface (API) on TCP/IP, however, does not implement such a protocol. It is up to the application designers to agree on a private protocol that will offer a degree of protection to critical resources. Such a protocol is needed to ensure that if the client and the server both update recoverable resources, then changes to those resources are consistent. The private protocol allows the client and server to ensure that either all the changes that they have made are committed, or all changes are backed out.

### 5.1.3 Security in a Distributed System

Security is essential in a CICS system because of the data in files and databases to which it has access. In order for many users to safely access a CICS system, that system must be secure. Security is even more important when you consider the additional number of users who will have access when you connect CICS to a TCP/IP network.

In a CICS system the basic security mechanism is to check whether the user at a particular terminal is authorized to perform a particular function. Because the servers used for CICS to TCP/IP Sockets Interface applications do not run as terminal attached tasks within CICS, we have to take extra precautions to ensure that the system is secure. These precautions are described in Chapter 9, “Implementing Security in the CICS to TCP/IP Sockets Interface” on page 95.

### 5.1.4 Design Restrictions

Some functions that are commonly used when writing CICS transactions are not available to applications that are using the CICS to TCP/IP Sockets Interface. These restricted functions include the following:

- Transaction routing is not available for the child servers.
For example, you cannot have the Listener running in one CICS region and start a child server in another CICS region that will talk to the same client.

- There are no EXEC CICS SEND or RECEIVE commands for the socket. The Sockets Interface uses calls to sockets modules rather than CICS SEND or RECEIVE commands.

Some users may want to receive input and then route the new transaction to some “back-end” application owning region (AOR). You cannot do this with the TCP/IP sockets support. You may want to consider distributed program link (DPL) instead. Here the TCP/IP initiated CICS transaction first enters one program, and the first program in turn links to the “real” application in another AOR using DPL.

### 5.2 Distributed Applications

In a distributed application there are typically two components, a client process and a server process. The server is a process that carries out work for the client. The CICS to TCP/IP Sockets Interface uses three different types of server processes:

- Iterative servers
- Concurrent servers
- Child servers.

These are described in the following sections.

#### 5.2.1 Iterative Servers

An iterative server processes the client’s request itself (see Figure 36). While the iterative server is busy processing one client’s request, it cannot accept new requests from other clients, so they will be queued. Iterative servers should be used only for requests that can be processed quickly; otherwise, other client requests could wait a long time for service.

![Figure 36. Iterative Server](image-url)
5.2.2 Concurrent Servers

A concurrent server starts a new process to perform the processing for the client (for the CICS to TCP/IP Sockets Interface this will be a new CICS task). Because the time taken to start a new process is very short, the concurrent server can then accept new client requests while the CICS task it has just started is concurrently processing the previous request. Concurrent servers should be used when the time to process a client’s requests may be large, such as when transferring files.

5.2.3 Child Servers

We will call the new process started by the concurrent server, a child server. This is because it is a “child” of the concurrent server. This naming convention helps us to be precise when talking about different types of servers in the CICS region. If you are using a concurrent server, the bulk of the processing will be performed by its child servers.

5.2.4 More about Servers

The CICS to TCP/IP Sockets Interface provides a concurrent server called the Listener. The function provided by the Listener will probably be acceptable in most cases, but you may want to write a replacement concurrent server (perhaps using the Listener as a model) if the Listener does not meet all your requirements.

The Listener will start a child server (which you have to write yourself) to do the work requested by the client. Meanwhile the Listener is able to service other incoming requests. You can also write your own iterative server to service short requests from clients.

Both concurrent and iterative servers are long-running CICS transactions. Because such transactions can tie up CICS resources for long periods, they should be kept to a minimum number.

In the UNIX environment, concurrent servers work by performing fork() and exec() functions to spawn off new tasks. We use EXEC CICS START commands to start the new tasks in CICS. In UNIX, the spawned, or child, task inherits copies of all of the attributes of the parent task; this includes the socket descriptors that the parent task was using. In CICS we must explicitly pass the socket descriptor to the newly started child server transaction. The GIVESOCKET() and TAKESOCKET() functions provided in TCP/IP for MVS allow us to do this, as shown in Figure 37 on page 45.
5.3 Choosing between Iterative and Concurrent Servers

If you use an iterative server to process requests from a client, it is busy while it is doing this and cannot accept any new requests. When it is not busy, it will still retain the MVS subtask it has been using to communicate with the TCPIP address space.

If you use a concurrent server, the child server will only exist when it is needed, as will the MVS subtask it uses to communicate with the TCPIP address space. However, there is an overhead in starting a new task in CICS for the child server, and in attaching a new MVS subtask for the child server.

We were not able to make any measurements to compare the performance of the two servers, but we believe that you should, in general, use a concurrent server with child servers rather than an iterative server. The IBM-supplied Listener provides two methods of starting child servers, one of which allows you to minimize the cost of starting new CICS tasks and attaching MVS subtasks. The first method is to start a new child server to process each client request. The second method uses CICS TDQs to queue a client request to an existing child server, or to start a new child server if one does not already exist.

The second method can avoid the cost of attaching a new CICS task and MVS subtask, and is suitable for applications where the Listener will be receiving many client requests for small pieces of work which are not response time critical. Because the child server can be terminated when there is no work to process, it does not have the disadvantage of retaining resources that are not being used that an iterative server would have.

If you decide to implement your own concurrent server you should consider implementing a design that allows you to start child servers using either of the above methods.
5.4 Transferring Data between Client and Server

TCP/IP, like LU6.2 and other protocols, buffers the data that is being sent to optimize performance. However, there is a difference in how the buffering works which affects the way you design your own programs.

With LU6.2 the data stream is blocked and unblocked by VTAM. This makes programming simpler because a receive request in an application program will get the data transmitted by the corresponding send in another application program (if this is not so, perhaps because data has been truncated, your application program will be notified of the error). In effect you do not have to be aware of the buffering when you write your application programs.

On the other hand, with TCP/IP you need to be aware that the data is being buffered, since several send requests to transmit data may be buffered together and sent in one block to the receiving application. The receiving application has to be able to do its own unblocking.

TCP/IP will return the number of bytes that were actually sent or received. For example, if you try to send 5000 bytes, TCP/IP may only transfer 2100 bytes, and you will have to resend the rest. Alternatively, you may send 80 bytes, followed by 120 bytes, followed by 100 bytes. TCP/IP may transfer this as one send of 300 bytes. Because TCP/IP returns the actual number of bytes transferred, the application can then decide whether it needs to send or receive more data.

Figure 38 and Figure 39 on page 47 show two functions that can be used to ensure that you send or receive exactly the number of bytes that you want.

```c
/* This function ensures that exactly nbytes are sent */

int sendn (sd, ptr, nbytes, flags)
int sd;
char *ptr;
int nbytes, flags;
{
    int num_left, num_sent;
    num_left = nbytes; 1
    while (num_left > 0) 2
    {
        num_sent = send(sd, ptr, num_left, flags); 3
        if (num_sent <= 0)
            return (num_sent); 4 /* error on send */
        num_left -= num_sent; 5
        ptr += num_sent; 6
    }
    return (nbytes - num_left); 7 /* should be nbytes */
}
```

Figure 38. sendn() Function

In Figure 38 we first set num_left to be equal to the number of bytes we want to transfer 1. We then check to see if we have sent all the data 2, and if there is any data left to be sent we try to transfer all that remains in one go 3. If there is an error, num_sent will be less than or equal to zero, and we report this
to the caller 4. Otherwise, num_sent shows the number of bytes actually transmitted, and we subtract this from the number of bytes that were attempting to send 5. A pointer that shows the start of the remaining data to be transferred is moved to correct position 6, and we loop around again. When we have transferred all the data, we return with the number of bytes transferred 7.

Figure 39 shows the corresponding function for receiving data.

```c
/* This function ensures that exactly $n$ bytes are received */
int recvnfrom (sd, ptr, nbytes, flags, name, namelen)
int sd;
char *ptr;
int nbytes, flags;
struct sockaddr_in *name;
int *namelen;
{
    int num_left, num_recvd;
    num_left = nbytes;
    while (num_left > 0)
    {
        num_recvd = recvfrom(sd, ptr, num_left, flags, name, namelen);
        if (num_recvd <= 0)
            return (num_recvd); /* error on recv */
        else if (num_recvd == 0)
            break; /* EOF */
        num_left -= num_recvd;
        ptr += num_recvd;
    }
    return (nbytes - num_left); /* will be >= 0 */
}
```

Figure 39. recvn() Function

The data transfer method shown in Figure 38 on page 46 and Figure 39 works well for clients and servers that transmit fixed-length data, since both the client and the server know how many bytes they are expected to send to or receive from the other. If we are dealing with variable-length data, we need to add to this technique.

Let us consider the case where we have a server that is sending several variable-length records to a client. In this example we have a loop in the server that performs a send call for each record. If these records are, say, 130, 120, and 85 bytes in length, the three send calls in the server may only result in one single physical send from TCP/IP; the data may be buffered and sent as block of 335 bytes.

The client will keep doing receive calls until it cannot receive any more data. Its receive calls will have been coded so that they request the maximum record size, say 200 bytes. What may occur is that the client will only do two receives: the first will receive 200 bytes, and the second will receive 135 bytes. The logical separation of the records has not been preserved.
We can deal with this situation in (at least) three different ways:

- Insert end-of-record markers in the data.
- Add a length-of-record field at the start of each record.
- Use a send-receive sequence to flush the TCP/IP buffers.

Figure 40 illustrates the problem, and Figure 41 on page 49 shows how the send-receive sequence can be used to overcome it.

```
SERVER : TCP/IP BUFFER : CLIENT :
Send 130 bytes
   A

Send 120 bytes
   B   A   B

Send 85 bytes
   C   A   B   C

Receive 200 bytes
   B   C   A   B

Receive 200 bytes
   B   C
<flush>

(135)
```

Figure 40. TCP/IP Buffering

This method relies on the fact that a send followed by a receive causes the TCP/IP buffer to be flushed. The client and server would be coded so that the client acknowledges the receipt of each record, and the server waits for an acknowledgment before sending the next record (and vice versa). Although we have more flows with this method, it provides a simple solution, ensuring that the logical record separations are maintained throughout the transfer.
Figure 41. Using Acknowledgments to Preserve Logical Record Separations
5.5 Summary

We have extra considerations when designing TCP/IP socket applications that involve a CICS/ESA or CICS/MVS system:

- We need to ensure that we optimize our use of CICS resources, designing short-lived child server transactions wherever possible.

- If we are dealing with critical resources, we must design our applications in such a way as to minimize the risk of loss of data integrity.

- Because of the nature of distributed applications and networks we must also be concerned with security, to protect our data.

We also need to consider the practical problems involved in transferring data between clients and servers in a distributed system. TCP/IP does not handle logical record separations. We must use methods in our clients and servers that ensure that these record separations are preserved.
Chapter 6. Overview of the Sample Systems

In this chapter we describe the applications that we created to show some of the features of the CICS to TCP/IP Sockets Interface. The programs that make up these applications are included on the diskette that is supplied with this document. Appendix A, “The Supplied Sample Applications” on page 123 describes the content of the diskette, and how you can install the sample applications on your system.

We used two different sets of application programs, one to show how to create a new application, based on the Listener, and the other to show how to create your own concurrent and iterative servers.

The two sample applications that we used are:

- The Messaging System
- The Record Retrieval System.

6.1 The Messaging System

Our main sample application is a distributed messaging system, which we use to illustrate the design ideas that we discussed in Chapter 5, “Application Design Considerations” on page 41. In this application, messages are stored centrally on a CICS/ESA system; remote clients can process these messages. CICS/ESA is used to implement the server function, controlling the central message repository. We implement clients on:

- CICS/ESA
- OS/2
- AIX.

CICS/ESA uses the EBCDIC character format; OS/2 and AIX use ASCII. We specify that all data transmissions using sockets will be in ASCII. This requires the CICS/ESA programs to convert their data before sending and after receiving.

6.1.1 Function of the Messaging System

The messaging system provides the following functions:

- Write one message to another users mail-log
- Read one message from a mail-log
- Inquire on a mail-log
- Delete one message
- Delete a mail-log.

Each user has a mail-log. Each user must be known to the server system. A user signs on to a client, which then sends requests to the server system. In our sample, any user can use the client on any platform.

This sample application is implemented using a concurrent server (the Listener). The child servers that are started by the Listener are implemented using a separate CICS transaction for each function.
In this sample application, we transfer variable-length records between the clients and the servers, using the method shown in Figure 41 on page 49 to preserve logical record separations.

Because the server function is implemented under CICS/ESA, we need to incorporate the following CICS transaction design considerations in the design of the server:

- Efficient application design
- Recoverable resources
- Security.

### 6.1.2 Efficient Application Design

We may have a high demand for the server function, or the demand may grow significantly over time. To allow for growth and to make the best use of existing resources, we need to optimize the server’s performance. One way of doing this is to ensure that resources that have been used by a CICS transaction are freed up as soon as possible. Transactions that stay active or suspended in a CICS system use up resources (for example, storage). To optimize performance we normally want to ensure that a transaction terminates when it has finished doing useful work.

Looking at the specification of the application, we can isolate the individual server functions and implement them as separate transactions. These transactions will be active only long enough to perform the requested function; they will then terminate. Creating separate transactions for each function gives us better control of the server system, since it allows us to easily determine the functions being performed. For example, we can find out how many requests there were to write a message to another user's mail-log by implementing the system this way, and then using the CICS/ESA shutdown statistics.

The Listener will be the only long-running Sockets Interface transaction in the CICS/ESA system. The design we have chosen (that is, the decision to terminate child servers when they have processed one request) means that clients have to initiate a connection with the CICS system for each function they want to perform, and then terminate the connection on completion of the function. We have chosen this design because we assume that there will be a low volume of requests to use the messaging system. This may seem to be an additional overhead for the client, but the benefits gained in the server system outweigh any additional overhead.

The shorter the time the child server transactions spend active, the less chance they have of experiencing problems because of other applications. Because there is less of a chance of the CICS/ESA system reaching limits in the system, for example the max-task limit. The system should have a better overall performance than would be the case if we had many long-running server transactions.

The short-lived child server transactions hold a path to the TCPIP address space for only a short time. A finite number of paths are available, and in a heavily loaded system we want to optimize the use of these paths; otherwise we may meet problems due to a lack of available paths.

If the cost of starting a new CICS task for each client request proves to be too expensive, for example, in a heavily loaded system where performance is
critical, the Listener transaction provided with the CICS to TCP/IP Sockets Interface offers an alternative method for starting child servers. In the initial send to the Listener, a client can specify that the server be initiated be means of a TDQ, as shown in Figure 42.

![Figure 42. Starting a Child Server Using a Transient Data Queue](image)

Only one instance of the child server SRI1 will exist in the CICS system at any one time. If multiple requests come in for that server, the data will be queued in the TDQ. The server must be coded to repeatedly read the TDQ until there is no more data, at which point it can close the connection and terminate. The advantage of this method is that there is no overhead for starting multiple transactions; the disadvantage is that all client requests to this server are single-threaded, so response times may be extended.

In our sample client-server system, the Inquire child server is started by the transient data method, and it single-threads the client requests. All other child servers are initiated through CICS task control, using an EXEC CICS START command.

### 6.1.3 Recoverable Resources

The message repository in the CICS system is implemented as a protected resource. We want to prevent any lost or corrupted messages. Therefore, all updates to this resource must be checked, and any problems reported back to the client. We need to implement a protocol that both client and server will follow in the event of a problem. Two basic types of problem can occur from the client’s point of view: ones that can be recognized, and ones that cannot.

A recognizable problem is one of which both the server and client are aware. For example, the server cannot add the message to the appropriate repository for some reason and informs the client of the situation. In this case we must ensure that the client and server agree on what to do next. In this example, they may agree to retry the attempt, or abandon it.

An unrecognizable problem is one in which the server experiences a problem but is unable to communicate the problem to the client. For example, if the child server transaction abends, or the TCP/IP connection is destroyed, the client will be informed about an error, but it will be unable to determine at what stage during the server’s processing the problem occurred.

The two phase commit model (Figure 43 on page 54) can be used to explain the stages that would be required to ensure that the commits in a client and server are synchronized.
In Figure 43 the client controls the commit process; the client tells the server exactly when to commit the update.

In our sample system the client only invokes one server function at a time, and only the child server updates recoverable resources (in other words, we do not need to worry about the case where we are updating recoverable resources in more than one location at the same time). Because of this, our solution is slightly simpler than the full two-phase commit model.

Figure 44 on page 55 shows that our server will send acknowledgments to the client after updating a protected resource when:

- 1. The update has successfully completed \textit{(prepare to commit)}.
- 2. The update is committed \textit{(committed)}.

This method allows the server to verify the status of the TCP/IP connection before committing the update and to synchronize the client. If the client does not return the acknowledgment to the server, the server will assume that the client has experienced a problem and will back out the update.
If the child server was updating a recoverable resource and it abended during processing, the client will want to find out whether the update was committed or not. The simplest action is to inquire on the state of the server, using a separate CICS server transaction. If the server has not committed the update, the client must back out any changes it has made (in the general case where our client may also be updating resources), or possibly send a message to the end user telling them that the update has failed. If the server has committed the update, then the client must commit (if it was updating resources), or possibly send a message to the end user stating that the data was committed. To do this inquiry, the server system must keep a log. This log records whether the commit has been performed or not for each user (see Figure 45 on page 56).

```
<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update</td>
<td>I have updated</td>
</tr>
<tr>
<td></td>
<td>and am preparing</td>
</tr>
</tbody>
</table>
|                 | to commit                       | 1
| Acknowledge     | I have committed                | 2
```

Figure 44. Simplified Commit Model
Server
...
...

send (preparing to commit)
write data to log Log
receive ack
commit update
delete log <log deleted>
send (committed)
close

Figure 45. Using the Commit Log in the Server

The client invokes a new child server to inquire on the old child server’s state.

When the new child server carries out the inquiry, it checks the old child server’s log. If it does not exist, then the client assumes the update was committed. If it does exist, then the client assumes the child server transaction abended before the commit was performed, and so it backs out. The new child server then deletes the log.

Using this simple recovery mechanism, we have narrowed down the window in which the child server state cannot be resolved to one point. If the child server transaction abends while deleting the log, then the log remains, but the commit has been performed. The check transaction will therefore incorrectly assume that the commit has not been made, and it will inform the client. We believe that this is an acceptable risk in our sample.

This recovery mechanism is implemented in the Write-message child server and in the Delete-message and Delete-message-log child servers, because these are the only child servers that update a recoverable resource.

6.1.4 Security

Security is covered in Chapter 9, “Implementing Security in the CICS to TCP/IP Sockets Interface” on page 95.
6.2 The Record Retrieval System

Our second sample application is a record retrieval system, which shows how you can code your own concurrent server or iterative server. The record retrieval system has only one simple function: to read a record from the sample file (FILEA) provided with CICS/ESA. Since the messaging system demonstrates the other design considerations, the record retrieval system does not include code for recovery, error checking or other such functions.

The client program specifies the key of the record that is to be read, and the server reads the record and passes it back to the client (or returns a message saying RECORD NOT FOUND if it cannot find the record).

This function is implemented as both a concurrent server with its associated child server, and an iterative server. Both servers use the same client program.
Chapter 7. Server Design for the Sample Systems

In this chapter we describe the design and implementation of the three different types of server that we used in the sample programs.

The topics we cover are:

- Child servers (invoked by the Listener provided by the CICS to TCP/IP Sockets Interface)
- Concurrent Server
- Iterative Server.

The child server implements the function described in Section 6.1, “The Messaging System” on page 51, while the concurrent and iterative servers implement the function described in Section 6.2, “The Record Retrieval System” on page 57.

7.1 Child Servers

We coded five different programs, one to serve each of the functions available to the client as described in Section 6.1, “The Messaging System” on page 51. These programs are all invoked from the Listener.

SISSRI1C inquires on the mail-log messages for a specific user.
SISSRR1C reads a specific message from a user’s mail-log.
SISSRW1C writes a message to a user’s mail-log.
SISSRD1C deletes one specific message from a user’s mail-log.
SISSRD2C deletes all messages from a user’s mail-log.

7.1.1 Invoking a Child Server Using the Listener

A child server can be started by the Listener in different ways, depending on the information passed by the client. To invoke a child server that has a transaction identifier of TRAN within CICS, you can use any of the following methods:

TRAN,(userdata) causes the Listener to initiate the child server transaction (TRAN, in this case) immediately using an EXEC CICS START command.

TRAN,(userdata),IC,000005 causes the Listener to initiate TRAN using an EXEC CICS START INTERVAL command with a delay of 5 seconds. We did not use this type of start in our samples.

TRAN,(userdata),TD causes the Listener to write a message (userdata) to a TDQ named TRAN. TRAN is an intra-partition queue with a trigger-level set to 1.

If your child server has a different transaction identifier (FUN1, for example) then you could replace TRAN with FUN1 in these examples.
7.1.2 Invoking the Sample Child Servers Using the Listener

We wanted to show two different types of child server initialization in our sample application, so we created four programs (SISSRD1C, SISSRD2C, SISSRR1C, and SISSRW1C), which are initiated by an EXEC CICS START command, and one program (SISSRI1C), which is initiated by the trigger-level of the TDQ.

We describe one sample of each type of initialization in detail, using programs SISSRW1C and SISSRI1C to show the basic steps. We look only at the functions that use TCP/IP socket calls.

7.1.3 Child Server Started by an EXEC CICS START Command (SISSRW1C)

Figure 47 on page 61 shows the steps needed to invoke a child server from the Listener using an EXEC CICS START command. In this figure (and in Figure 54 on page 65, which shows the steps needed to invoke a child server from the Listener using a TDQ), the names socket c, socket s and socket cs refer to the socket descriptor in the client application, the socket descriptor in the Listener (server) application, and the socket descriptor for the child server, respectively.

In this example the client program wants to write a message to a user’s mail-log, so it requests the appropriate server function by sending the following information to the Listener:

```
SRW1,USERID
```

where SRW1 identifies the transaction that the Listener is to start (and thus the function that is requested) and USERID is information for the Security Exit.

The Listener accepts the connection, creates a new socket descriptor for this connection, and starts the child server transaction, SRW1, using an EXEC CICS START command. Transaction SRW1 invokes program SISSRW1C. You can find the source code for SISSRW1C in the MESSAGE directory of the diskette included with this document.

The Listener passes information to the child server in a data area when it issues the EXEC CICS START command. This includes information about the socket descriptor that the child server is going to use. The child server needs this information to allow it to use the correct socket descriptor. The first thing that the child server has to do is to use an EXEC CICS RETRIEVE command to retrieve the information passed by the Listener. Figure 46 shows the EXEC CICS RETRIEVE command in program SISSRW1C.

```
RETRIEVE-ROUTINE.
  MOVE SPACES TO TCPSOCKET-PARM.
  MOVE LENGTH OF TCPSOCKET-PARM TO MSG-LEN.
  EXEC CICS RETRIEVE INTO(TCPSOCKET-PARM)
    LENGTH(MSG-LEN)
  END-EXEC.

RETRIEVE-ROUTINE-EXIT.
  EXIT.
```

Figure 46. Sample Child Server (IC) - EXEC CICS RETRIEVE Command

The information that is retrieved from the Listener is placed in the data structure shown in Figure 48 on page 62.
### CLIENT

Create a stream socket `c` with the `socket()` call.

(Optional)
Bind socket `c` to a local address with the `bind()` call.

Connect socket `c` to a host using a `connect()` call.

Tell TCP/IP that you are now accepting connections on socket `s` by using the `listen()` call.

Monitor the activity on socket `s` for any incoming data or exceptions with the `select()` call.

Accept the connection and create a new socket `cs` with the `accept()` call.

Socket `s` remains available to accept new connections from other clients.

Start server transaction using `EXEC CICS START` and pass socket `cs` using the `givesocket()` call.

<table>
<thead>
<tr>
<th>CHILD SERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use <code>EXEC CICS RETRIEVE</code> to retrieve the information passed by the Listener's <code>EXEC CICS START</code> command.</td>
</tr>
<tr>
<td>Retrieve the passed socket <code>cs</code>, using <code>takesocket()</code></td>
</tr>
</tbody>
</table>

Close socket `cs`, using the `close()` call.

Accept another connection or close socket `s` with the `close()` call.

Transfer data on socket `c`, using the `send()` and `recv()` calls.

Close socket `c` with the `close()` call.

### LISTENER

Create a stream socket `s` with the `socket()` call.

Bind socket `s` to a local address with the `bind()` call.

Tell TCP/IP that you are now accepting connections on socket `s` by using the `listen()` call.

Accept the connection and create a new socket `cs` with the `accept()` call.

Socket `s` remains available to accept new connections from other clients.

Transfer data on socket `cs` using the `send()` and `recv()` calls.

Close socket `cs` with the `close()` call.

---

*Figure 47. Child Server Started by the EXEC CICS START Command*
After the child server has executed the RETRIEVE command, CLIENT-IN-DATA contains the information passed by the client program (USERID in our example), and GIVE-TAKE-SOCKET contains the socket descriptor that we need.

We are now ready to issue our first socket call in the child server program. Before we do that we need to explain a little about how the socket calls are implemented in a COBOL program. First, the COBOL declaration shown in Figure 49 is used by each call.

For all the calls in this program we move a value to the field COMMANDA, which tells EZACICAL which type of socket call this is. Section B.1.1, “Ordered List of Socket Calls” on page 144 gives a complete list of the valid codes. The individual calls are described in detail in Appendix B, “The Sockets API for COBOL and Assembler Language Programs” on page 143. The call descriptions give details of what each of the parameters mean.

TOKEN always contains the value TCPIPIUCVSTREAMS.

SOCKETD is not used in every call. For example, the GETCLIENTID call does not need the SOCKETD parameter. If it is used it contains the number of the socket descriptor that this call is to use.

The child server retrieves the socket descriptor passed to it by the Listener using a TAKESOCKET call. We move 32 to COMMANDA to tell EZACICAL that...
this is a TAKESOCKET call, and then move GIVE-TAKE-SOCKET to
TAKE-SOCKET-LDESC, TAKE-SOCKET-SOCKNO and SOCKETD so that EZACICAL
knows which socket descriptor to work with. Figure 50 on page 63 shows the
paragraph in SI SSRW1C which includes the TAKESOCKET call.

```
TAKESOCKET-Routine.
MOVE 32 TO COMMANDA.
MOVE AF-INET TO TAKE-SOCKET-DOMAIN.
MOVE LSTN-NAME TO TAKE-SOCKET-NAME.
MOVE LSTN-SUBTASKNAME TO TAKE-SOCKET-TASK.
MOVE GIVE-TAKE-SOCKET TO TAKE-SOCKET-LDESC
TAKE-SOCKET-SOCKNO SOCKETD.
CALL &EZACICAL& USING TOKEN COMMANDA
TAKE-SOCKET-HZERO
TAKE-SOCKET-CLIENTID
TAKE-SOCKET-LDESC
TAKE-SOCKET-SOCKNO
TAKE-SOCKET-ERR
TAKE-SOCKET-RET.
```

Figure 50. Sample Child Server (IC) - TAKESOCKET Call

After the TAKESOCKET call has completed, the child server is connected to the
client and can begin to transfer data on the socket using SEND and RECVFROM
calls. Our sample child server application sends a confirmation message (OK)
to the client, as shown in Figure 51, indicating that it is ready to receive the
message that the client wants to write to the mail-log.

Note that we call the EBCDIC to ASCII translation routine (EZACIC04) to
translate the data to ASCII before sending it to the client.

```
SEND-OK-Routine.
MOVE &OK& SEND-BUF.
MOVE 2 TO SEND-NBYTE.
MOVE 0 TO SEND-FLAGS.
CALL &EZACIC04& USING TOASCII-TOKEN SEND-BUF
SEND-NBYTE.
MOVE 20 TO COMMANDA.
CALL &EZACICAL& USING TOKEN COMMANDA SOCKETD
SEND-NBYTE
SEND-FLAGS
SEND-DZERO
SEND-BUF
SEND-ERR
SEND-RET.
```

Figure 51. Sample Child Server (IC) - SEND Call

The client receives the confirmation, formats the message that it wants to be
written in the mail-log, and sends it back to the child server. The child server
receives the message using the RECVFROM call, as shown in Figure 52 on
page 64.
RECV-MSG-Routine.
MOVE 16 TO COMMANDA.
MOVE 0 TO RECVFROM-FLAGS.
MOVE 500 TO RECVFROM-NBYTE.
MOVE LOW-VALUES TO RECVFROM-BUFF.

CALL ¢EZACICAL¡ USING TOKEN COMMANDA SOCKETD
RECVFROM-ZERO
RECVFROM-FLAGS
RECVFROM-NBYTE
RECVFROM-FROM
RECVFROM-BUFF
RECVFROM-ERR
RECVFROM-RET.

CALL ¢EZACIC05¢ USING TOEBCDIC-TOKEN
RECVFROM-BUFF RECVFROM-NBYTE.

CLOSE-Routine.
MOVE 3 TO COMMANDA.
CALL ¢EZACICAL¡ USING TOKEN COMMANDA SOCKETD
CLSE-ZERO
CLSE-ERR
CLSE-RET.

IF CLSE-RET < 0
THEN GO TO ERROR-RETURN-Routine.

Note that we use the ASCII to EBCDIC translation routine (EZACIC05) to translate the client data to EBCDIC format.

After we have received the data, we write it to the user's mail-log. The child server could now send more data to the client. In our design each child server executes just one request, so now the child server executes a CLOSE call to release the socket descriptor, as shown in Figure 53.

When the socket has been released the program issues an EXEC CICS RETURN command to end the transaction.

7.1.4 Child Server Started by a Transient Data Queue (SISSRI1C)

Figure 54 on page 65 shows us that the design of a server started by using a TDQ is similar to that for a server started by the EXEC CICS START command.

The client program requests a server function by sending the following data to the Listener:

SRI1,USERID,TD

where SRI1 identifies the function. USERID is information for the security exit and TD indicates that the server is to be started using transient data.
The Listener accepts the connection, creates a new socket descriptor for this connection, and writes to a TDQ called SRI1, which has a trigger level of 1 (for a sample of the DCT entry for this queue see Figure 15 on page 21).
The first action that the child server has to take is to read the data that was written by the Listener. You must code a EXEC CICS READQ TD command in your program to do this, as shown in Figure 55 on page 66.

```plaintext
SETUP-ROUTINE.

MOVE SPACES TO TCPSOCKET-PARM.
MOVE LENGTH OF TCPSOCKET-PARM TO MSG-LEN.
EXEC CICS READQ TD QUEUE('SRI1')
   INTO(TCPSOCKET-PARM)
   LENGTH(MSG-LEN)
END-EXEC
```

Figure 55. Sample Child Server (TD) - EXEC CICS READQ TD Command

The data structure that you name in the INTO option of the EXEC CICS READQ command must have the format shown in Figure 56. You may change the names for the data items, but not the types or lengths.

```plaintext
01 TCPSOCKET-PARM.
   02 GIVE-TAKE- SOCKET PIC 9(8) BINARY.
   02 LSTN-NAME PIC X(8).
   02 LSTN-SUBTASKNAME PIC X(8).
   02 CLIENT-IN-DATA PIC X(36).
   02 SOCKADDR-IN.
      03 SIN-FAMILY PIC 9(4) BINARY.
      03 SIN-PORT PIC 9(4) BINARY.
      03 SIN-ADDR PIC 9(8) BINARY.
      03 SIN-ZERO PIC X(8).
```

Figure 56. Sample Child Server (TD) - Parameters Received from the READQ TD

SISSRI1C executes a TAKESOCKET call to accept the socket descriptor from the Listener.

The child server is connected to the client and can begin to read the messages from the mail-log and send the information to the client. Each message sent by the child server with the SEND call must receive an acknowledgement from the client (OK).

After the last message has been transferred to the client, the child server terminates the connection by executing a CLOSE call.

After closing the socket descriptor, the child server returns to SETUP-ROUTINE and issues another EXEC CICS READQ TD command to check if there are any more requests from other clients. If a new request exists, the child server executes a TAKESOCKET again and starts the conversation with the new client. If there are no new requests, the child server gets a QZERO return code from the EXEC CICS READQ TD command and issues an EXEC CICS RETURN command to end the transaction.

You can find the source code of SISSRI1C in the MESSAGE directory of the diskette included with this document.
7.2 The Sample Concurrent Server (SISUCSVC)

SISUCSVC is one of the sample servers in the record retrieval system. You can find the source code for SISUCSVC in the RETRIEVE directory of the diskette that came with this document.

Figure 57 on page 68 shows the socket calls used by a concurrent server program. The SELECT call is used to prevent the ACCEPT call from blocking the application. If the ACCEPT call is used alone it blocks (waits) until there is an input event. There will never be an input event if the Sockets Interface is being closed, so we must use the SELECT call to allow us to escape if there has not been any input for a given length of time.

The concurrent server we have implemented performs the same socket functions as the Listener. The supplied Listener requests only 50 socket descriptors. In most cases this is enough for a concurrent server, since one of its socket descriptors is only in use until a child server issues a TAKESOCKET call for that socket descriptor, at which point the concurrent server can close that socket descriptor and reuse it. If you find that the 50 socket descriptors requested by the Listener are not enough, you will need to write your own concurrent server.

7.2.1 Starting the Concurrent Server

We start our concurrent server by entering the following transaction at a terminal:

```
UCSV-3015
```

where UCSV is the transaction code that we have decided to use for our concurrent server and 3015 is the port number that will be used by this concurrent server (this can be any unused port number, but your client program needs to know which port number you are using).

7.2.2 Function of SISUCSVC

The concurrent server needs to know which port is being used, because we will use this information in the BIND call. We must perform an EXEC CICS RECEIVE command to obtain the port number that was specified when the UCSV transaction was entered. The EXEC CICS RECEIVE command used in SISUCSVC is shown in Figure 58 on page 69.
Figure 57. Structure of Our Concurrent Server
**SETUP-ROUTINE.**

MOVE LENGTH OF INPUT-W-AREA TO CLENG.
EXEC CICS RECEIVE INTO (INPUT-W-AREA)
   MAXLENGTH (CLENG)
END-EXEC.

* * WE ALSO NEED TO INITIALISE THE SOCKET-TABLE ... *

PERFORM SET-SOCKET-TBL THRU SET-SOCKET-TBL-EXIT
   VARYING INDX FROM 2 BY 1 UNTIL INDX EQUAL 50.

**SETUP-ROUTINE-EXIT.**
EXIT.

**SET-SOCKET-TBL.**
MOVE LOW-VALUE TO SOCKET-DESC (INDX).

**SET-SOCKET-TBL-EXIT.**
EXIT.

---

**Figure 58. Sample Concurrent Server - RECEIVE Command**

In Figure 58 we have also initialized SOCKET-TABLE, which is used to show the current status of any of our socket descriptors. For example, if the value of SOCKET-DESC (INDX) is A, we are about to issue an ACCEPT call for this socket descriptor. We have made the table large enough to handle 50 socket descriptors, since we are going to request 50 socket descriptors in our INITAPI call.

After we have initialized SOCKET-TABLE we issue the INITAPI call to establish a path to the TCPIP address space, as shown in Figure 59.

---

**INITAPI-ROUTINE.**

MOVE 0 TO COMMANDA.
MOVE ¢IUCVAPI ¢ TO INITAPI-IDENT.
MOVE 50 TO INITAPI-MAXSOC.
MOVE 2 TO INITAPI-MAXAPI.
MOVE EIBTASKN TO SUBTASKNO.

CALL €EZACICAL€ USING TOKEN COMMANDA
   INITAPI-IDENT
   INITAPI-MAXSOC
   INITAPI-MAXAPI
   INITAPI-SUBTSK
   INITAPI-ZERO
   INITAPI-MAXNO
   INITAPI-RET.

---

**Figure 59. Sample Concurrent Server - INITAPI Call**

We use the CICS task number (EIBTASKN) as the first seven characters of the subtask identifier, to ensure that the subtask name will be unique.

The concurrent server uses the SOCKET call to get a new socket descriptor, as shown in Figure 60 on page 70.
We use the SETSOCKOPT call to clear or set the options associated with our socket descriptor, as shown in Figure 61.

In our concurrent server program we set the SO_REUSEADDR option to allow our program to bind to a port number even if that port is still not fully closed. After a bound socket descriptor is closed, it goes into a timed wait state to ensure that all remote peers have acknowledged the close. During this period, the port at which that socket descriptor is bound cannot be reused unless the socket descriptor in a new BIND call has the SO_REUSEADDR option set.

Next, the concurrent server will do a BIND call to bind a unique local name to the specified socket descriptor. We use the port number passed by the user at the beginning of the transaction (3015 in our example). The BIND call for SISUCSVVC is shown in Figure 62 on page 71. The field W-PORT contains the port number provided by the user. Moving the value 0 to BIND-IP-ADDR means that this socket descriptor will be bound to the local host.
The concurrent server now informs TCPIP that it is ready to accept connections from clients. This is done by issuing the LISTEN call, as shown in Figure 63.

The LISTEN call creates a queue for incoming connection requests sent by clients. The field LISTEN-BACKLOG defines the length of this queue. In our sample up to five clients can wait to connect to the concurrent server. Once this queue is full, later CONNECT calls from clients will be refused.

The concurrent server must now execute a GETCLIENTID call, as shown in Figure 64. This function returns the jobname of the CICS region and the name of the MVS subtask associated with the calling CICS transaction. This information must be passed to any child server to allow it to issue a TAKESOCKET call.

The information about the client will be received in the field GETCLIENT-ID, which has the structure shown in Figure 65 on page 72.
The concurrent server can now monitor the activity of a set of socket descriptors to see if any of them are ready for reading or have an exception condition pending. This is done by using the SELECT call, as shown in Figure 66 on page 73.

Before issuing the SELECT call we need to set up the masks that specify which socket descriptors are to be monitored for activity. In our example we are interested only in exception and read responses, so we only need to set up SELECT-EX-MASK and SELECT-RD-MASK. SELECT-WR-MASK should be set to all 0’s.

When we issue the SELECT call we use a timeout value of one minute (SELECT-SECS, which is part of SELECT-TIMEOUT, is declared with an initial value of 60). You may want to use a different value for your system (for example, a smaller value could reduce the delay in shutting down your CICS region).

The field SELECT-NUMFDS represents the number of socket descriptors that are to be monitored by the SELECT call. It should be the number of socket descriptors specified in INITAPI-MAXSOC when we made the INITAPI call plus 1. In our example INITAPI-MAXSOC was set to 50, so we set SELECT-NUMFDS to 51.

The field SELECT-LOM holds the length of the masks used to specify which socket descriptors we want to monitor. You can find information on how to calculate how long the masks will be in Appendix B, “The Sockets API for COBOL and Assembler Language Programs” on page 143.
In our sample the concurrent server executes continuously until you purge the task or start transaction CSKD to disable the Sockets Interface. To achieve this the field FOREVER has a value of 1, and we use the check shown in Figure 67.

When we return from the SELECT call we check to see if the transaction CSKD is running in the system. Figure 68 on page 74 shows the EXEC CICS INQUIRE command that we use to check for CSKD.

The EXEC CICS INQUIRE TASK LIST returns a list of all active or suspended transactions in CICS. Here the list of transaction identifiers is loaded into a variable called TRAN-AREA. We check TRAN-AREA using the COBOL verb INSPECT, to see if transaction CSKD is in this list (field DISABLE-TRAN has already been given the value CSKD).

If CSKD is in the list, then the interface is being closed and so we go to the LEAVE-SERVER routine (which terminates this concurrent server and allows the interface to be closed). The LEAVE-SERVER routine issues a CLOSE call for socket descriptor 0, and ends the concurrent server transaction by issuing an EXEC CICS RETURN command.
EXEC CICS INQUIRE TASK LIST
LISTSIZE(NUM-TASKS)
SET(ADDRESS OF TASK-AREA)
SETTRANSID(ADDRESS OF TRAN-AREA)
END-EXEC.
MOVE 0 TO CTR.
INSPECT TRAN-AREA TALLYING CTR
FOR ALL DISABLE-TRAN.

IF CTR NOT EQUAL 0 THEN
  GO TO LEAVE-SERVER.

Figure 68. Sample Concurrent Server - EXEC CICS INQUIRE TASK

If the CSKD transaction is not in the system, we test the return code of the SELECT call. If it is 0, then a timeout occurred; in our example, we re-execute the SELECT call.

If the return code was not 0, then there has been some sort of activity on at least one socket descriptor. We need to check to see what that activity was. In our case we do not expect a write to have taken place, so we test the value of fields SELECT-RR-MASK (read response) or SELECT-RE-MASK (exception response) to see what occurred. We simply test to see if the masks are all zeros, as shown in Figure 69.

IF SELECT-RR-MASK NOT EQUAL ZEROS THEN
  PERFORM PROCESS-READ THRU PROCESS-READ-EXIT.

IF SELECT-RE-MASK NOT EQUAL ZEROS THEN
  PERFORM PROCESS-EXCP THRU PROCESS-EXCP-EXIT.

Figure 69. Sample Concurrent Server - Testing Response from SELECT

If an exception condition occurred we know that a child server has performed a TAKESOCKET call (we know this because we only check for exception conditions on sockets where we have issued a GIVESOCKET call). Now we know that the socket descriptor has been passed to the child server and it can safely be closed, so we perform a CLOSE call for this socket descriptor. We then return to the SELECT call.

If a read condition has occurred this means that an incoming connection request from a client has been detected on socket descriptor 0. We now accept the request using the ACCEPT call, as shown in Figure 70 on page 75. The ACCEPT call returns a new socket descriptor for the connection.
Figure 70. Sample Concurrent Server - ACCEPT Call

We tell TCPIP that we want to pass this new socket descriptor to a child server by issuing a GIVESOCKET call, as shown in Figure 71. We must specify the new socket descriptor number obtained from the ACCEPT call, and the client name obtained from the GETCLIENTID call.

Figure 71. Sample Concurrent Server - GIVESOCKET Call

Finally, we start the child server transaction using an EXEC CICS START command, as shown in Figure 72. TCPSOCKET-PARM contains the number of the new socket descriptor created during the ACCEPT call (field GIVE-TAKE-SOCKET), the CICS jobname and the subtask name obtained by the GETCLIENTID call (fields CLNT-NAME and CLNT-TASK).

Figure 72. Sample Concurrent Server - Starting the Child Server

For our sample application, the field OUR-SERVER has already been initialized with the value USV2, which is the transaction identifier for the child server in the record retrieval system.

After issuing the EXEC CICS START command, the concurrent server returns to the SELECT routine to wait until another request arrives, or the child server issues a TAKESOCKET call.
We start the client program that connects to this concurrent server by entering the following information at a CICS terminal:

\[ \text{UCLT-090C0F05-3015-000111} \]

where UCLT is the transaction code, 090C0F05 is the hexadecimal representation of the connection IP address of the MVS system on which our server CICS system is running (9.12.15.5), 3015 is the port number of the concurrent server, and 000111 is the record number that we want to retrieve.

### 7.3 The Sample Iterative Server (SISUSRVC)

Figure 73 on page 77 shows an example of the sequence of socket calls used by an iterative server program.

You can find the source code for our sample, SISUSRVC, in the RETRIEVE directory of the diskette included with this document. SISUSRVC is one of the sample servers in the record retrieval system.

#### 7.3.1 Starting the Iterative Server

Start the iterative server by entering the following transaction at a terminal:

\[ \text{USRV-3020} \]

where USRV is the transaction code and 3020 is the port number that will be used by this iterative server (any unused port number).
<table>
<thead>
<tr>
<th>CLIENT</th>
<th>ITERATIVE SERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a stream socket c with the socket() call.</td>
<td>Create a stream socket s with the socket() call.</td>
</tr>
<tr>
<td>(Optional) Bind socket c to a local address with the bind() call.</td>
<td>Bind socket s to a local address with the bind() call.</td>
</tr>
<tr>
<td>Connect socket c to a foreign host with the connect() call.</td>
<td>Tell TCP/IP that you are now accepting connections on socket s by using the listen() call.</td>
</tr>
<tr>
<td></td>
<td>Monitor the activity on socket s for any incoming data or exceptions with the select() call.</td>
</tr>
<tr>
<td></td>
<td>Accept the connection and create a new socket ns with the accept() call.</td>
</tr>
<tr>
<td></td>
<td>Socket s remains available to accept new connections from other clients.</td>
</tr>
<tr>
<td>Transfer data on socket c, using the send() and recv() calls.</td>
<td>Transfer data on socket ns, using the send() and recv() calls.</td>
</tr>
<tr>
<td>Close socket c with the close() call.</td>
<td>Close socket ns with the close() call.</td>
</tr>
<tr>
<td></td>
<td>Accept another connection or close socket s with the close() call.</td>
</tr>
</tbody>
</table>

*Figure 73. Iterative Server*
7.3.2 Function of SISUSRVC

The first thing that the iterative server does is to receive the information entered by the user, as shown in Figure 74. This includes the port number, which we need to make the BIND call later.

```
SETUP-ROUTINE.
   MOVE LENGTH OF INPUT-W-AREA TO CLENG.
   EXEC CICS RECEIVE INTO(INPUT-W-AREA)
       MAXLENGTH(CLENG)
   END-EXEC.
```

Figure 74. Sample Iterative Server - RECEIVE Command

The iterative server establishes a path to the TCPIP address space using an INITAPI call, as shown in Figure 75. In our sample we request 50 socket descriptors from TCPIP (INITAPI-MAXSOC). We use the CICS task number (EIBTASKN) as the first seven characters of the subtask identifier. This ensures that the subtask name will be unique.

```
INITAPI-ROUTINE.
   MOVE 0 TO COMMANDA.
   MOVE 'IUCVAPI' TO INITAPI-IDENT.
   MOVE 50 TO INITAPI-MAXSOC.
   MOVE 2 TO INITAPI-MAXAPI.
   MOVE EIBTASKN TO SUBTASKNO.
   CALL 'EZACICAL' USING TOKEN COMMANDA
        INITAPI-IDENT
        INITAPI-MAXSOC
        INITAPI-MAXAPI
        INITAPI-SUBTSK
        INITAPI-ZERO
        INITAPI-MAXSNO
        INITAPI-RET.
```

Figure 75. Sample Iterative Server - INITAPI Call

The iterative server must now get a new socket descriptor using the SOCKET call, as shown in Figure 76 on page 79.
Figure 76. Sample Iterative Server - SOCKET Call

We can now either clear or set the options associated with our socket descriptor using the SETSOCKOPT call, as shown in Figure 77. In SISUSRVC, as in SISUCSVC, we set the SO_REUSEADDR option to allow our program to bind to a port number even if that port is still not fully closed.

Figure 77. Sample Iterative Server - SETSOCKOPT Call

We now use the BIND call to bind a unique local name to the specified socket descriptor, as shown in Figure 78 on page 80. We use the port number passed by the user in the start of the transaction (3020 in our example), moving this information to the BIND-PORT field. BIND-IP-ADDR has the value 0, which means that the socket descriptor is bound to the local host.
BIND-Routine.

MOVE 2 TO COMMANDA.
MOVE 2 TO BIND-AF-INET.
MOVE W-PORT TO BIND-PORT.
MOVE 0 TO BIND-IP-ADDR.

CALL &EZACICAL: USING TOKEN COMMANDA SOCKETD
BIND-NAME
BIND-ERR
BIND-RET.

Figure 78. Sample Iterative Server - BIND Call

The next socket call is a LISTEN call. It tells TCPIP that the iterative server program is now ready to accept connections. As with SISUCSVC, LISTEN-BACKLOG defines the queue length for requests from clients. Figure 79 shows the LISTEN call for the iterative server. In our example, up to a maximum of two clients can wait to connect to the iterative server. If a third client tries to connect to this server, the connection request is refused.

LISTEN-Routine.

MOVE 13 TO COMMANDA.
MOVE 2 TO LISTEN-BACKLOG.

CALL &EZACICAL: USING TOKEN COMMANDA SOCKETD
LISTEN-ZERO
LISTEN-BACKLOG
LISTEN-ERR
LISTEN-RET.

Figure 79. Sample Iterative Server - LISTEN Call

Note: Normally we would use the SELECT call to monitor the activity on our socket descriptors to see if they are reading, writing, or have a exception condition pending. This would enable us to perform other work if there were no activity on the socket descriptor. To simplify the coding of this sample program, we decided not to use a SELECT call, but to go straight to the ACCEPT call. For a production application you should make sure that you use a SELECT call to prevent the ACCEPT call from blocking the application.

The iterative server is now ready to accept the connection from the client. It issues an ACCEPT call to accept the connection request and return a new socket descriptor for the connection, as shown in Figure 80 on page 81.
Figure 80. Sample Iterative Server - ACCEPT Call

The ACCEPT call will block the application if there are no connection requests. We only exit the ACCEPT once a connection request is detected.

After the ACCEPT call the connection is available to the client and it can communicate with the iterative server using SEND and RECVFROM calls.

When the conversation finishes, the iterative server must close the socket descriptor created by the ACCEPT call. At this point it can either execute another ACCEPT call to connect to another client, or close socket descriptor 0 and end the transaction.

**Note:** SISUSRVC was coded to remain available to accept new connections from other clients. To stop it you must purge the transaction (because we do not check to see if the interface is being disabled when we return from an ACCEPT call).

You start the sample client that connects to this iterative server by entering the following information:

```
UCLT-090C0F05-3020-000100
```

where UCLT is the transaction code, 090C0F05 is the hexadecimal representation of the connection IP address of our CICS system (9.12.15.5), 3020 is the port number of the iterative server and 000100 is the record number that we want to read.
Chapter 8. Client Design for the Sample Systems

In this chapter we discuss how to write client transactions on CICS that use the CICS to TCP/IP Sockets Interface. We cover the design and implementation of the client used in the messaging system sample application described in Chapter 6, “Overview of the Sample Systems” on page 51. We do not address the simpler client that we used for the record retrieval system.

We discuss client design for CICS, OS/2, and AIX platforms. We take the specification for the whole system and produce a set of requirements for the client, and discuss alternative implementation techniques.

The topics we cover are:

- CICS client considerations
- Sample client design.

8.1 CICS Client Considerations

We describe a simple model for a CICS client transaction that uses the Sockets Interface. A CICS client transaction is part of a client-server pair, where the server can be either another CICS sockets transaction, or a program running on a remote workstation.

To be able to issue a SOCKET call, our client must first establish itself as a sockets user using the INITAPI call. This call establishes an IUCV path between the CICS region and the local TCPIP region. From the client’s point of view, this call allows the client to both specify a name for its subtask, and also request a number of socket descriptors to be preallocated to the subtask. An MVS subtask is created for the new CICS client, and is ready to handle socket calls from the CICS client, or to pass data to the CICS client from the TCPIP region. Figure 81 shows the status of the system after the INITAPI call has completed.

<table>
<thead>
<tr>
<th>MVS</th>
<th>CICS</th>
<th>TCPIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITAPI</td>
<td>Subtask</td>
<td></td>
</tr>
<tr>
<td>Client</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subtask</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>V</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 81. INITAPI Call for CICS Client*

After the client has successfully issued the INITAPI call, it needs to request a socket descriptor, which it can use to communicate with the server. It does this
by using the SOCKET call. The SOCKET call takes one of the preallocated socket descriptors for the client, and returns it to the client. The client may now use this socket descriptor to attempt to communicate with the server.

After we have obtained the socket descriptor, we issue a CONNECT call, which does two things. First, it binds the new socket descriptor to an unused, short-lived TCP/IP port (sometimes called an ephemeral port). All TCP/IP based communication takes place between tasks at ports. Secondly, the CONNECT call sends a connection request on the socket to the specified server. A server is identified by the combination of the address of the host on which it is running and the number of the port at which its socket is bound. Figure 82 shows the status of the system at this time.

![Diagram showing the status of the system at the time of a CONNECT call](image)

Figure 82. CONNECT Call for the CICS Client

The server accepts this connection request, and we now have an established connection between client and server.

Using this connection, we can now transfer data between the two programs, using the SEND/RECVFROM or READ/WRITE calls, as shown in Figure 83.

![Diagram showing SEND and RECVFROM calls](image)

Figure 83. SEND and RECVFROM Calls from the CICS Client
After the conversation has completed, either the server or the client issues a CLO\*E call for the connection. The other partner should then complete the close by also issuing a CLO\*E call, terminating the connection. The status of the system at this time is shown in Figure 84 on page 85. The client’s subtask terminates normally upon termination of the client transaction.

![Figure 84. CLO\*E Call for the CICS Client](image)

This is the simplest kind of conversation, in which you only need to use a few of the socket calls. Other socket calls can be used to alter the operating characteristics of a socket descriptor, gather more information about the partner task, or multiplex input/output on multiple socket descriptors.

### 8.2 Sample Client Design

We now look at the specification and implementation of the client used in the sample system.

#### 8.2.1 Client Specification

The client should offer the following functions to the user:

- Write one message to another user’s mail-log
- Read one message from a user’s mail-log
- Query messages in a user’s mail-log
- Delete one message from a user’s mail-log
- Delete an entire mail-log
- Browse a user’s mail-log.

Note that the last function, browse, is not a function that is specifically provided by the server, but is actually a combined function. First, we query the mail-log to find out how many messages are in the user’s mail-log; then we read each message.

For each function, the client should initiate a conversation with the server host, identify a server transaction to execute, carry out a conversation with that server transaction to perform the required function, and then close the connection. The
connection between client and server should exist only for the shortest possible duration to prevent the server transactions from wasting CICS resources.

The client should be capable of handling the following data format:

- To-user—8 bytes
- From-user—8 bytes
- Subject—20 bytes
- Message—432 bytes.

Before they can access the server transactions, users executing the client are required to identify themselves so that their userids can be verified by the Security Exit.

The client should also implement a protocol (to which the server must also adhere) for handling updates to protected resources. This is because TCP/IP does not provide a function equivalent to SNAs SYNCH_LEVEL(SYNCT) support. Both the client and the server should be implemented such that the client is aware of the state of the servers resources, and can act accordingly without damaging the integrity of the data in those resources.

8.2.2 Client Implementation

We now describe how we implemented the sample client for the messaging system. This client communicates with the IBM-supplied Listener transaction.

In contrast to the server system, in which the individual functions are handled by separate transactions, the main client function is one single program. In the CICS implementation, it is a pseudo-conversational program. Essentially, the client program comprises two parts:

- Code that handles the TCP/IP sockets conversations
- Code that handles the data manipulation and user interface.

Although you can design these two areas separately, you must design the client sockets code in conjunction with that of the server. This enables the application designers to ensure that the conversation flows are complete.

In our sample, all servers are started using the Listener transaction. This means that the initial send from the client must be to the Listener, and must contain the relevant information required to start the desired server. The Listener expects data in the format shown in Figure 85 on page 87.

We require that all but one of the servers be started immediately. For those servers, the only format we use in the initial send to the Listener is:

```
TRAN,Data-area
```

We assumed that the function to query messages in a user’s mail-log is heavily used, and so we want to reduce the processing load on CICS as it attaches and detaches new transactions. We do this by using the transient data mechanism provided by the Listener. For this server, we use the following format for the initial send to the Listener:
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAN</td>
<td>The CICS transaction ID that the Listener starts. This field can be one to</td>
</tr>
<tr>
<td>(Data-area)</td>
<td>four character.</td>
</tr>
<tr>
<td>XX</td>
<td>Optional, startup type that can be:</td>
</tr>
<tr>
<td></td>
<td>. IC/ic for Interval Control</td>
</tr>
<tr>
<td></td>
<td>. TD/td for Transient Data.</td>
</tr>
<tr>
<td></td>
<td>If this field is blank, startup is immediate.</td>
</tr>
<tr>
<td>HHMMSS</td>
<td>Optional, hours, minutes and seconds for interval time if the transaction</td>
</tr>
<tr>
<td></td>
<td>is started using interval control.</td>
</tr>
</tbody>
</table>

The client indicates the function it requires by using the transaction ID of the appropriate server. The content of the data-area depends on the function being requested. A simplified data flow diagram for a *read-message* request from the client is shown in Figure 86 on page 88.
The length of the conversation is kept to a minimum and, after it has retrieved the initial send data from the Listener, the server has all the information it requires. The client is also easier to implement in that once it has requested the server function, all it has to do is receive the requested message. Note that any recognized errors that occur in the server will be sent back to the client in place of the message. This ensures that the user is made aware of any problems in the server.

The first thing the client does is to establish itself as a TCP/IP application by using the INITAPI call. Figure 87 on page 89 shows how it does this in our sample CICS client.
INITAPI-ROUTINE.
MOVE 0 TO COMMANDA.
MOVE 'IUCVAPI' TO INITAPI-IDENT.
MOVE 50 TO INITAPI-MAXSOC.
MOVE 2 TO INITAPI-MAXAPI.
MOVE EIBTASKN TO SUBTASKNO.

CALL 'EZACICAL:' USING TOKEN COMMANDA
   INITAPI-IDENT
   INITAPI-MAXSOC
   INITAPI-MAXAPI
   INITAPI-SUBTSK
   INITAPI-ZERO
   INITAPI-MAXSNO
   INITAPI-RET.

   IF INITAPI-RET < 0 THEN
      GO TO RETURN-ROUTINE.

Figure 87. Sample CICS Client - INITAPI Call

In the OS/2 and AIX client, the INITAPI call is not necessary.

Next, we must acquire a socket descriptor, which we use to both establish a
connection to the server, and then communicate with the server. Figure 88
shows the commands we use to do this in our sample CICS client.

SOCKET-ROUTINE.
MOVE 25 TO COMMANDA.
MOVE 2 TO SOCKET-AF.
MOVE 1 TO SOCKET-TYPE.
MOVE 0 TO SOCKET-PROTOCOL.

CALL 'EZACICAL:' USING TOKEN COMMANDA
   SOCKET-HZERO
   SOCKET-AF
   SOCKET-TYPE
   SOCKET-PROTOCOL
   SOCKET-SOCKNO
   SOCKET-ERR
   SOCKET-RET.

   IF SOCKET-RET < 0 THEN
      GO TO RETURN-ROUTINE.

Figure 88. Sample CICS Client - SOCKET Call

Figure 89 on page 90 shows the same function for the OS/2 and AIX clients.
if ((clients_sock_desc = 
    socket(AF_INET, SOCK_STREAM, socket_protocol)) == call_failed) 
{ 
    printf("Problem with socket ()\n"); 
    report_nerrno(errno); 
    exit(3); 
}

In order for the client to communicate with the server, it must connect to it. To connect to a server, a client must know the IP address of the host on which the server is running and the number of the port at which the server is bound. In our sample, since all client requests are sent to the Listener, we need to know the IP address and port number for the Listener.

In our example, to allow for flexibility, we decided to allow the users to specify the server’s IP address and port number. The CICS client has default values that can be modified, but the OS/2 and AIX clients require this information before any connection can be made. The user must enter this information, which means that if the server is moved (either to a different port number, or to a different physical host), we do not need to recompile our clients.

For important network client-server applications, which are unlikely to be moved, it would be possible to hard code the port number. This is the rationale behind the concept of well-known ports. Standard TCP/IP applications, such as TELNET and FTP, are assigned port numbers that are published and well-known throughout the TCP/IP community. This means that if users wants to use TELNET on a remote host, they know that it will be on port 23, since that is the published well-known port for TELNET. Figure 90 shows how we specify the IP address and port number in the CONNECT call of our sample CICS client.

```c
CONNECT-Routine.
    MOVE 4 TO COMMANDA.
    MOVE 2 TO CONN-AF-INET.
    MOVE X2D-FW TO CONN-IP-ADDR.
    MOVE PORTNBI TO CONN-PORT.

    CALL 'EZACICAL' USING TOKEN COMMANDA SOCKETD
        CONN-NAME
        CONN-ERR
        CONN-RET.

    IF CONN-RET < 0 THEN
        GO TO RETURN-Routine.
```

Figure 90. Sample CICS Client - CONNECT Call

Figure 91 on page 91 shows the same function for the OS/2 and AIX clients.
if (connect(clients_sock_desc, (struct sockaddr *) &server_inetaddr, sizeof(server_inetaddr)) == call_failed) {
    printf("Problem with connect()\n");
    report_nerrno(errno);
    exit(4);
}

Figure 91. Sample OS/2 and AIX Clients - CONNECT Call

The steps shown in Figure 90 on page 90 and in Figure 91 are used for each server request. When the client wants to contact a server to request a function, it performs these steps, regardless of which server it is trying to communicate with.

Once the connection has been made, the client must tell the Listener which server is required. This is done on the first SEND call from the client. This send must contain the transaction ID of the server, the required starting mechanism (task control, interval control or transient data), and any data that the client wants to pass to the server. The content of this first send differs according to the required function. Figure 92 shows the commands we used to do this in our sample CICS client.

MOVE SRI1 TO I-TRANID.
MOVE USERID TO USER-INQ.
MOVE LENGTH OF INQUIRE-REQ TO SEND-NBYTE.
MOVE INQUIRE-REQ TO SEND-BUFF.
PERFORM SEND-Routine.

SEND-Routine.
    CALL °EZACIC04¢ USING TOASCII-TOKEN
        SEND-BUFF SEND-NBYTE.
    MOVE 20 TO COMMANDA.
    MOVE 0 TO SEND-FLAGS.
    CALL °EZACICAL¢ USING TOKEN COMMANDA SOCKETD
        SEND-NBYTE
        SEND-FLAGS
        SEND-DZERO
        SEND-BUFF
        SEND-ERR
        SEND-RET.
    IF SEND-RET < 0 THEN
        GO TO RETURN-Routine.

Figure 92. Sample CICS Client - SEND Call

Figure 93 on page 92 shows the same function for the OS/2 and AIX clients.
strcpy(io_buf, SRI1, t);
strcat(io_buf, this_user);
strcat(io_buf, TD);

if (send(clients_sock_desc, io_buf, io_len, send_flags) == call_failed)
{
    printf(Problem with send()\n);
    report_errno (errno);
    exit(5);
}

Figure 93. Sample OS/2 and AIX Clients - SEND Call

Note: In the CICS client, we called the EBCDIC to ASCII translation routine, EZACIC04. This is because we specified that all data being transmitted must be in ASCII. No such translation is required in OS/2 or AIX.

After we have made the initial send, we always expect some data from the started server. This may be either an acknowledgement to indicate that more data can be sent, or the result of the server’s processing. In the case of the server that queries the mail-logs, all of the information it needs to begin work is contained in the initial send from the client. However, the server that writes to the mail-logs requires more data. The data on an initial send to the Listener is limited to 35 characters. Our messages can be up to 468 characters, so we must use another SEND call to transfer the data. To receive data from the server we use the RECVFROM call. Figure 94 shows the commands we use we do this in our sample CICS client.

RECVFROM-ROUTINE.
    MOVE 16 TO COMMANDA.
    MOVE 0 TO RECVFROM-FLAGS.

    CALL &EZACICAL USING TOKEN COMMANDA SOCKETD
       RECVFROM-ZERO
       RECVFROM-FLAGS
       RECVFROM-NBYTE
       RECVFROM-FROM
       RECVFROM-BUFF
       RECVFROM-ERR
       RECVFROM-RET.

    IF RECVFROM-RET < 0 THEN
    GO TO RETURN-ROUTINE.

    CALL &EZACIC05 USING TOEBCDIC-TOKEN
       RECVFROM-BUFF RECVFROM-NBYTE.

Figure 94. Sample CICS Client - RECVFROM Call

Figure 95 on page 93 shows the same function for the OS/2 and AIX clients.
num_bytes = recv(clients_sock_desc, rcv_buf, 50, recv_flags);
if (num_bytes == call_failed)
{
    printf("Problem with recv()\n");
    report_nerrno(errno);
    exit(6);
}

Figure 95. Sample OS/2 and AIX Clients - RECV Call

After we have completed our conversation between client and server, we close the connection. Figure 96 shows the CLOSE call in our sample CICS client.

CLOSE-ROUTINE.
    MOVE 3 TO COMMANDA.
    CALL &EZACICAL USING TOKEN COMMANDA SOCKETD
    CLSE-ZERO
    CLSE-ERR
    CLSE-RET.
    IF CLSE-RET < 0 THEN
        GO TO RETURN-ROUTINE.

Figure 96. Sample CICS Client - CLOSE Call

Figure 97 shows the same function for the OS/2 client.

if ((soclose(clients_sock_desc)) == call_failed)
{
    printf("Problem with soclose()\n");
    report_nerrno(errno);
    exit(8);
}

Figure 97. Sample OS/2 Client - SOCLOSE Call

Figure 98 shows the same function for the AIX client.

if ((close(clients_sock_desc)) == call_failed)
{
    printf("Problem with close()\n");
    report_nerrno(errno);
    exit(8);
}

Figure 98. Sample AIX Client - CLOSE Call
Chapter 9. Implementing Security in the CICS to TCP/IP Sockets Interface

In this chapter we describe how to implement a security scheme in the CICS to TCP/IP Sockets Interface environment. We describe the Security Exit facility, and suggest ways of integrating this facility with existing CICS security features.

In this chapter we assume that all child servers are started through the Listener transaction. If you have other servers (concurrent or iterative) listening on ports, you can code those server programs to link to EZACICSE or a similar user-written security program.

The topics we cover are:

- Why the Sockets Interface needs security
- Checking security for incoming requests
- Sample security implementation
- Summary.

9.1 Why the Sockets Interface Needs Security

CICS provides security mechanisms to protect against unauthorized access to resources. The need to prevent unauthorized access from external sources is clear when you consider that the Sockets Interface enables you to connect CICS to one of the largest worldwide networks, the Internet. This means that many interconnected systems could potentially connect to your CICS region. We describe how you can use the Security Exit and CICS preset terminal security to protect your system.

9.2 Checking Security for Incoming Requests

We recommend that you check security for incoming requests using the Security Exit supplied with the Sockets Interface. In addition to the Security Exit, you can also use CICS preset terminal security.

9.2.1 Security Exit

The Security Exit consists of one program, EZACICSE, which provides the mechanism for security checking in the Sockets Interface (see Figure 99 on page 96).
As shown in Figure 99, the Listener links to the Security Exit before starting a child server task, and passes the information shown in Figure 100 on page 97 to the Security Exit in a COMMAREA. The Security Exit then returns an indicator stating whether or not the request for the server can be allowed.

If EZACICSE is defined to the CICS region, the Listener links to it every time there is an incoming request to start a server transaction. Therefore, EZACICSE should perform minimal processing to avoid any performance degradations in the Listener.

If EZACICSE is not defined to the CICS region, the Listener permits any server transaction to be started.

EZACICSE decides whether or not to allow the Listener to start the server transaction based on the information it is passed. The structure of the data is shown in Figure 100 on page 97. The Switch field is used to pass the decision back to the Listener.
<table>
<thead>
<tr>
<th>Description</th>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CICS transaction ID</td>
<td>4-byte character</td>
<td>CICS tranid requested by client</td>
</tr>
<tr>
<td>Data area</td>
<td>40-byte character</td>
<td>User data received from client</td>
</tr>
<tr>
<td>Action</td>
<td>2-byte character</td>
<td>Method of starting task:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IC - Interval Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>KC - Task Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TD - Transient Data</td>
</tr>
<tr>
<td>Interval control time</td>
<td>6-byte character</td>
<td>Interval requested for IC start in the format HHMMSS</td>
</tr>
<tr>
<td>Address family</td>
<td>Halfword binary</td>
<td>Network Address Family:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AF_INET (2)</td>
</tr>
<tr>
<td>Port</td>
<td>Halfword binary</td>
<td>Requestor’s port number</td>
</tr>
<tr>
<td>Address</td>
<td>Fullword binary</td>
<td>IP address of requestor’s host</td>
</tr>
<tr>
<td>Switch</td>
<td>1-byte character</td>
<td>Switch:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - Permit transaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not 1 - Prohibit transaction</td>
</tr>
<tr>
<td>Msg switch</td>
<td>1-byte character</td>
<td>Indicator set by Listener:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - Permit message to client</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not 1 - No message to client</td>
</tr>
<tr>
<td>Terminal ID</td>
<td>4-byte character</td>
<td>Either TERMID for CICS server, or low-values</td>
</tr>
<tr>
<td>Socket</td>
<td>Halfword binary</td>
<td>Socket descriptor</td>
</tr>
</tbody>
</table>

Figure 100. Data Passed to the Security Exit in the COMMAREA

Data area is a 40-byte field that is used to pass user data to the Security Exit. The Listener can accept only 35 bytes of data from the user, so it will only use the first 35 bytes of Data area.

The last field, Socket, is a new field that has been added with PTF UN40677. It is the socket descriptor from the ACCEPT call that the Listener issued, and it can be used by the Security Exit to communicate directly with the client and inform it of any error situations. The field Msg switch is set by the Listener to let the Security Exit know whether or not it is allowed to send messages to the client. This field is always set to 1 if PTF UN40677 is installed on your system, and is always set to 0 otherwise. To the PTF, you should write your Security Exit so that it checks this switch before attempting to send a message to the client. Because the Security Exit can issue socket calls, it must be link-edited as a Sockets Interface program.
9.2.2 Invoking the Security Exit

If there is a program definition for EZACICSE in the CICS region, then the Listener will attempt to link to it. If EZACICSE exists then one of the following events will occur:

- EZACICSE is linked to and permits the Listener to start the server. The Listener sends a message back to the client informing it of the situation.
- EZACICSE is linked to and prohibits the Listener from starting a server. The Listener sends a message back to the client informing it of the situation.
- The Listener finds that EZACICSE is disabled. The Listener does not start the server, but sends a message back to the client informing it of the situation.
- The Listener finds that EZACICSE cannot be loaded. The Listener does not start the server, but sends a message back to the client informing it of the situation.

If the CICS system is Version 3 Release 3 or higher, EZACICSE does not have to reside on the same CICS system as the Sockets Interface. In this case, the Security Exit can be accessed using DPL. DPL may be required if the resources that the Security Exit needs to do its checking exist on another CICS system. However, this function should be used with care, since it will increase the time taken to accept a client request and start a child server.

Note: If the Security Exit uses the new feature in PTF UN40677 and performs socket calls on the socket supplied by the Listener to send messages to the client, then the Security Exit must reside on the same CICS system as the Listener.

The data passed to the Security Exit from the Listener is a copy of the data that the Listener holds. The Listener will ignore any modifications to this information within the Security Exit, with the exception of the Switch and Terminal ID fields. The Switch field allows the Security Exit to either prohibit or permit the starting of a CICS server. The Terminal ID field allows the Security Exit to specify that the CICS server should be started against a terminal, imposing the terminal’s security on the transaction. If the Security Exit does not alter the Terminal ID field, the server task will not be terminal-attached.

The Security Exit has only the given data on which to decide whether or not to permit the specified server task to be started. It is entirely up to the Security Exit as to the criteria it uses to do this. It can base the decision on the information about the client and the prospective server, or on the data passed by the client.

9.2.3 Preset Terminal Security

CICS preset terminal security allows you to permanently associate a USERID with a terminal that is defined to CICS. CICS establishes security levels for a terminal when it is being installed. The preset security for a terminal is determined by the specification of the USERID parameter on the terminal definition in CICS/ESA. Preset terminal security may be used in conjunction with the Security Exit to provide a method of restricting the authority of external client tasks. This restriction can be achieved by defining a number of terminals and USERIDs with preset security.

The Security Exit should be written to determine the terminal at which to start the requested server transaction; either by identifying the client, or by examining
the data that the client has sent. The server transaction is then be restricted by the authority of the terminal at which it is started. This method could enable the same server transaction to be used by different clients, enforcing different security levels on each client’s server.

### 9.2.4 Combining the Security Exit and CICS Preset Terminal Security

Figure 101 shows the recommended security checks that both the Security Exit and CICS perform.

<table>
<thead>
<tr>
<th>REMOTE SYSTEM</th>
<th>SECURE CICS SYSTEM (with Sockets)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote Client</td>
<td>Security level Enforced by</td>
</tr>
<tr>
<td>Connect</td>
<td>Connection:</td>
</tr>
<tr>
<td>Attach Server</td>
<td>Server attach:</td>
</tr>
<tr>
<td></td>
<td>1 Can this client access this system? EZACICSE checks client’s authority to access this system.</td>
</tr>
<tr>
<td></td>
<td>2 Can this client attach this server? EZACICSE checks client’s authority to attach server.</td>
</tr>
<tr>
<td></td>
<td>3 At what level of security can this server run? EZACICSE establishes a security level for the server, which is enforced using preset terminal security.</td>
</tr>
<tr>
<td>Transaction</td>
<td>1 Has this server the authority to access other resources? CICS checks whether the child server can access other resources (such as files).</td>
</tr>
</tbody>
</table>

*Figure 101. Recommended Security Checks*
We describe a sample implementation is based on the supplied Security Exit. To implement our sample we assume that we know:

- All clients that have access to servers on the system
- All servers on the system
- Which clients can access which servers, and at what levels of security.

Figure 102 on page 101 shows the type of information required. The information should be collected and placed in a data structure suitable for the Security Exit. In our sample, we use CICS data tables to store the information.

Users are expected to identify themselves to the client, at which point the client places the USERID in the first data sent to the Listener. We also obtain the ID of the host on which the client is running from TCP/IP.

In our sample the information on system users and their security profile is held in one data table, while another data table lists the server transactions available for each security profile. Host information is held in a third data table.

In our sample we check the host ID to see if the host is known to us and is authorized to connect to servers in this CICS region. If the request passes this test, we read the data table that holds user information to get the security profile number for the user, and then check the second data table to see if that profile allows the user to access the requested server transaction. If the USERID cannot be found in the data table, that user is not allowed to use any of the child servers in our system.

We do not use CICS preset terminal security in our sample. We also do not use the new feature provided with PTF UN40677, which allows the Security Exit to make socket calls to the client.
Table 101. Sample Information for Security Exit

<table>
<thead>
<tr>
<th>Host-A</th>
<th>CICS</th>
<th>Host-B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client-1</td>
<td>SRV1, SRV2, SRV3</td>
<td>Client-1</td>
</tr>
<tr>
<td>Client-2</td>
<td>SRV1, SRV2</td>
<td>Client-2</td>
</tr>
<tr>
<td>Client-3</td>
<td>SRV4, SRV5, SRV6</td>
<td>Client-3</td>
</tr>
<tr>
<td>Client-4</td>
<td>SRV1, SRV2, SRV3</td>
<td>Client-4</td>
</tr>
</tbody>
</table>

Table 102. Sample Information for Security Exit

Figure 102. Sample Information for Security Exit

Keeping the information in separate data tables allows us a great degree of flexibility when we need add or remove users and hosts, or change security profiles. If the information is held as data within the program, the Security Exit must be recompiled after each change.

To keep the CPU processing in the Security Exit to a minimum in our sample, we constructed three user-maintained data tables containing:

- IP addresses of authorized host machines
- Authorized users and their security profiles
- Server transactions accessible by each security profile.

The Security Exit reads only the information from these data tables. We used the CICS-supplied transaction, CECI, to update the data tables. We use user-maintained data tables for our sample. One result of this is that updates made to the data table are not made automatically to the source data set. To make correct updates, you must either update both the data table and the source data set, or update the source data set and then reload the data table by closing and reopening it.

The supplied Listener transaction always allows the initial connection request from the remote client to be satisfied. The first opportunity for security checking comes after the Listener has received the initial data from the client. The Listener invokes the Security Exit and passes it the data shown in Figure 100 on page 97. In our sample Security Exit, we used the definitions shown in Figure 103 on page 102.
WORKING-STORAGE SECTION.

* LIST OF DATA-TABLES USED TO HOLD SECURITY INFORMATION *

01 SECURITY-TABLES.
   03 D-TAB1 PIC X(6) VALUE 'TCPDT1'.
   03 D-TAB2 PIC X(6) VALUE 'TCPDT2'.
   03 D-TAB3 PIC X(6) VALUE 'TCPDT3'.

01 USERID-DATA.
   03 USERID PIC X(8).
   03 FILLER PIC X(32).

01 SEC-USER-DATA.
   03 SEC-USERID PIC X(8).
   03 SEC-PROFILE PIC 99 VALUE 0.

01 SEC-ADDR-DATA.
   03 SEC-ADDR PIC X(4).

01 SEC-SERV-DATA.
   03 SEC-PROF PIC 99.
   03 SEC-SERVERS PIC X(24).

77 REC-LEN PIC S9(4) COMP.
77 CTR PIC 9 VALUE 0.
77 RESP PIC S9(8) COMP.
77 RESP2 PIC S9(8) COMP.

01 SEC-ADDRESS.

01 NEW-ADDRESS REDEFINES SEC-ADDRESS PIC X(4).

LINKAGE SECTION.

01 DFHCOMMAREA.
   05 SECURITY-TRANID PIC X(4).
   05 SECURITY-DATA PIC X(40).
   05 SECURITY-ACT PIC X(2).
   05 SECURITY-IC-TIME PIC 9(6).
   05 SECURITY-FAMILY PIC 9(4) COMP.
   05 SECURITY-PORT PIC 9(4) COMP.
   05 SECURITY-ADDR PIC 9(8) COMP.
   05 SECURITY-SWITCH PIC X(1).
   05 FILLER PIC X(1).
   05 SECURITY-TERMID PIC X(4).

Figure 103. COBOL Declarations Used by Our Sample Security Exit

Using the given definitions, the sample Security Exit performs the following steps:

1. Check the client-list to see whether this client is allowed access to this system. This is a two-stage process:
• Check the client’s host IP address to see whether it is an authorized host by reading the data table containing the authorized addresses, as shown in Figure 104 on page 103.

```
READ-ADDR-FILE.
    MOVE SPACES TO SEC-ADDR-DATA.
    MOVE LENGTH OF SEC-ADDR-DATA TO REC-LEN.
    EXEC CICS READ FILE(D-TAB1)
        INTO(SEC-ADDR-DATA)
        RIDFLD(NEW-ADDRESS)
        LENGTH(REC-LEN)
        RESP(RESP)
        RESP2(RESP2)
END-EXEC.
```

**Figure 104. Check the Client’s Host IP Address**

• Check the client’s ID to see whether it is a known client by reading the data table containing the list of known clients, as shown in Figure 105.

```
READ-USER-FILE.
    MOVE SPACES TO SEC-USER-DATA.
    MOVE LENGTH OF SEC-USER-DATA TO REC-LEN.
    EXEC CICS READ FILE(D-TAB2)
        INTO(SEC-USER-DATA)
        RIDFLD(USERID)
        LENGTH(REC-LEN)
        RESP(RESP)
        RESP2(RESP2)
END-EXEC.
```

**Figure 105. Check the Client’s ID**

• If the client fails either test then set the Switch field as shown in Figure 106 to inform the Listener that the client has failed the security check.

```
IF RESP NOT EQUAL DFHRESP(NORMAL) THEN
    MOVE '00' TO SECURITY-SWITCH
    MOVE MSG-N TO MSG-OUT
    GO TO LOG-MSG.
```

**Figure 106. Return Response to the Listener**

2. Check the client-list to see whether this client can access the requested server transaction. If it cannot, set Switch to 0 and return, as shown in Figure 107 on page 104.

3. (Not in supplied sample). Establish the security level for the server by choosing a terminal with the appropriate preset security and placing the terminal identifier in the Terminal ID field.
PERFORM READ-SERV-FILE THRU READ-SERV-FILE-EXIT.

MOVE 0 TO CTR.
INSPECT SEC-SERVERS TALLYING CTR
   FOR ALL SECURITY-TRANID.

IF CTR = 0 THEN
   MOVE 0 TO SECURITY-SWITCH
   MOVE MSG-4 TO MSG-OUT
   GO TO LOG-MSG.

...  

READ-SERV-FILE.
MOVE SPACES TO SEC-SERV-DATA.
MOVE LENGTH OF SEC-SERV-DATA TO REC-LEN.
EXEC CICS READ FILE(D-TAB3)
   INTO(SEC-SERV-DATA)
   RIDFIL(SEC-PROFILE)
   LENGTH(REC-LEN)
   RESP (RESP)
   RESP2 (RESP2)
END-EXEC.

Figure 107. Check whether the Client Is Authorized for the Server

9.4 Summary

When combined with CICS preset terminal security, the Security Exit provides a flexible approach to security in a TCP/IP environment. Unauthorized clients can be denied access to servers in the CICS system. Authorized clients are only allowed access to resources according to the appropriate CICS and RACF definitions. Knowing which clients are authorized to use which servers is the key factor in this security implementation.
Chapter 10. Problem Determination

In this chapter we address some of the problems that you may experience when both operating and using the CICS to TCP/IP Sockets Interface. We also describe various approaches to problem determination that may enable you to identify the cause and produce a solution.

The topics we cover are:
- Operating the Sockets Interface
- Running application programs.

10.1 Operating the Sockets Interface

This section should be used if you cannot enable or disable the Sockets Interface. Since operating problems may occur due to an incorrect installation, we have provided a checklist that you should follow to ensure that the Sockets Interface is correctly installed on your system.

10.1.1 Enabling the Sockets Interface

The first indication that there are problems with the installation occurs when you try to enable the Sockets Interface using the CSKE transaction. If the CSKE enable panel does not appear, then either the CSKE transaction, the EZACIC00 program, or the EZACICM mapset is incorrectly defined to CICS. See Section 3.3, “Definitions Needed in CICS” on page 15 for the required definitions. It is also possible that the required TCPIP data sets are not present in the CICS startup JCL. See Figure 5 on page 14 for examples of the required entries.

If CSKE is correctly defined, you are asked to enter data for the TCPIPUSERID and PORT entries. The TCPIPUSERID name that you enter should be the same as the name given in the TCPIPUSERID statement in tcpip.TCPIP.DATA. If it is not the same, CSKL will abend with a code of IAPI. See Section 4.1.4, “Errors in Enabling the Interface” on page 35.

The PORT number can be any available port number less than 32767. It must not be a well-known port number. It must also not be a port number that has been reserved for another job. The tcpip.PROFILE.TCPIP data set contains a list of reserved port numbers. If you specify one of these port numbers when you use CSKE, CSKL will abend with a code of BIND. See Section 4.1.4, “Errors in Enabling the Interface” on page 35.

If your system has multiple CICS regions accessing TCPIP on MVS, we recommend that you reserve ports in the tcpip.PROFILE.TCPIP data set to avoid the situation where a port is bound to by more than one server.

10.1.2 Disabling the Sockets Interface

The CSKD transaction is used to disable the Sockets Interface. If the disable panel does not appear, the CSKD transaction has not been correctly defined to CICS.

You are offered two disable options: a quiesce, or an immediate disable. The quiesce option flags the Sockets Interface as “quiescing.” All currently running
socket applications will be allowed to terminate normally. Any new socket applications will abend with a code of AEY9. The CSKD transaction that you used to start the procedure ends, but before it does, it starts a second CSKD transaction. This new copy of CSKD waits for all current applications to terminate. An immediate disable terminates all current socket tasks, and no new socket applications are allowed to start.

10.1.2.1 If the Quiescent Disable Does Not Complete

A quiescent disable can take up to three minutes to complete. This is normal. It can take this long because, when there is no socket activity, the Listener checks the state of the Sockets Interface only once every three minutes.

If the quiescent disable is taking more than the three minutes, or is not completing at all, it is probably because you have long-running server transactions that do not recognize that the Sockets Interface is being disabled. The Listener is specially coded to recognize that the Sockets Interface is being quiesced.

Long-running user-written server transactions should also be written to detect a quiescent shutdown. To do this, the server program issues a SELECT call, since we can check to see if the CSKD transaction is in the system when we timeout on the SELECT call. The sample concurrent server program, SISUCSVC, provides an example of this coding, as shown in Figure 68 on page 74.

If your server program issues an ACCEPT call without a SELECT call, it cannot detect a shutdown, and will therefore never terminate due to a quiescent disable.

If you have long running servers in your system that do not check for the existence of CSKD you should always use an immediate disable. All socket applications, even those that are currently running, will abend with a code of AEY9.

If the quiescent disable does not work, you will need to attempt an immediate disable. When you do an immediate disable after a quiescent disable, a copy of the CSKD transaction will remain suspended in the CICS system. This is the quiescent disable copy of CSKD. It will not terminate, and must be purged.

10.1.3 Installation Checklist

We provide the following checklist to ensure that you have correctly installed and configured the CICS to TCP/IP Sockets Interface.

- TCPIP data sets in the CICS startup JCL
  - tcpip.SEZATCP is required in the DFHRPL concatenation. It contains modules EZACIC00, EZACIC01, EZACIC02, EZACIC04, EZACIC05 EZACIC06, EZACICAL, and EZACICM.
  - tcpip.SEZALINK is required in the DFHRPL concatenation. It contains modules EZACIC03, and IUCVMULT.
- CICS resource definitions
  - Programs EZACIC00 (Assembler), EZACIC01 (Assembler), and EZACIC02 (COBOL) need to be defined.
  - Transactions CSKE (EZACIC00), CSKD (EZACIC00), and CSKL (EZACIC02) need to be defined.
  - Mapset EZACICM is required.
  - A DCT entry for TCPM is advised in order to get messages from the CICS to TCP/IP Sockets Interface.
- TCPIP for MVS definitions
  - `tcpip.PROFILE.TCPIP` should contain either ALLOWAPPL *, or an ALLOWAPPL statement for your CICS region.
  - `tcpip.PROFILE.TCPIP` should contain a PORT statement specifying your CICS jobname.

10.2 Running Application Programs

You should use this section if you have enabled the Sockets Interface, but are experiencing problems with sockets applications. The problem determination methods described here concentrate on two distinct areas:

- The CICS system
- The TCP/IP network.

10.2.1 Problem Determination Methods in CICS

If your CICS sockets application is experiencing problems, you can use various methods to find out exactly what is occurring in the CICS system.

10.2.1.1 CICS Auxiliary Trace

The Sockets Interface provides auxiliary CICS trace entries for each socket call. These can be collected by switching on the auxiliary tracing. The trace entries are user trace entries, and are made on entry to and exit from each socket call. The trace point ID of these user trace entries is X'0C7' (199). To format only the socket entries in the auxiliary trace, use the JCL shown in Figure 108 on page 108 to execute DFHTUP:
The following notes that refer to the sample trace shown in Figure 109 on page 109 will help you to interpret the trace information generated by the Sockets Interface. The sample trace shows what occurs after a client task has sent a connection request to the Listener. The Listener accepts the connection, receives the initial data from the client, and starts the requested child server transaction, passing the new socket. The child server issues a TAKESOCKET call, and the Listener then closes that socket.

In the trace entries, PARM2 identifies the particular socket call that created the trace entry. A list of socket calls and their call numbers is given in Appendix B, “The Sockets API for COBOL and Assembler Language Programs” on page 143. The value of the PARM3 field depends on the call, but it is usually the number of the socket being used in the call.
Figure 109 (Part 1 of 2). Sample CICS Auxiliary Trace Output
Figure 109 (Part 2 of 2). Sample CICS Auxiliary Trace Output

1. shows **CSKL**, the Listener transaction, returning from a SELECT call. This was caused by a client task performing a CONNECT call to the Listeners port. In the trace entry we see the ERRNO and RETCODE values.

2. shows the Listener performing an ACCEPT call (PARM2=0001). We see only the second and third parameters on the call. These parameter values are given in hexadecimal format.

3. is the return entry for the ACCEPT call.

4. shows the Listener performing a RECVFROM call, to receive the initial data from the client.

5. is the return from the RECVFROM call. The RETCODE field shows that we have received 18 bytes from the client. The RETCODE and ERRNO fields are given in decimal format.

6. shows the Listener now performing a GIVESOCKET call. We have started the requested server transaction, and are now passing socket 1 to it.

7. is the return from the GIVESOCKET call.

8. shows the Listener going back into a SELECT call. This allows the Listener to be informed when the server has performed the TAKESOCKET call, so that it can close socket 1.

9. is the first call by the server, **SRR1**. We are performing a TAKESOCKET call.

10. shows the Listener returning from the SELECT call.
11. shows the Listener performing a CLOSE call for socket 1.
12. is the return from the TAKESOCKET call, by the server task.
13. is the return from the CLOSE call, by the Listener.

10.2.1.2 GTF Trace

GTF trace entries can be collected for the Sockets Interface. These are generated by the MVS subtask (EZACIC03). The GTF trace entries and the CICS auxiliary trace entries can be merged to form a more complete picture of the steps taken during a sockets conversation.

To get GTF trace entries for socket applications, follow these steps:

1. Start GTF on MVS by entering /S GTF from a console.
2. Either use a PARMLST that specifies the USR parameter, or enter it manually when GTF prompts for parameters. The USR parameter tells GTF to trace all user-trace entries.
3. In CICS, use the CETR transaction to start both the auxiliary trace and CICS GTF trace. Ensure that the Master User Trace Flag is on.
4. Run your sockets applications.
5. Use CETR again to stop the auxiliary trace and CICS GTF trace.
6. Stop GTF on MVS.

Format the GTF trace using IPCS. The command:

```
GTF DSN(SYS1.TRACE) USR JOBNAME(CICSRS2A)
```

formats all user trace entries for job CICSRS2A. Figure 110 on page 112 shows a sample formatted output. The sample shows a section of GTF trace taken while a CICS client transaction was issuing a SOCKET call. The subtask’s GTF entries show the typical sequence that would be expected on any socket API call.
Figure 110 (Part 1 of 2). Sample GTF Trace Output
Figure 110 (Part 2 of 2). Sample GTF Trace Output

- 1 shows the CICS auxiliary trace entry for the SOCKET call (PARM2=25 (X'19')) being made by the Listener.
- 2 shows the GTF entry made on entry to the subtask. The subtask is identified by the SUBID field in the trace entry. This is required to identify trace entries if there is more than one subtask active during the trace.
- 3 shows the subtask is making a call to TCPIP to invoke the SOCKET function.
- 4 shows the subtask is waiting for an interrupt from TCPIP on completion of the SOCKET function.
5 shows the subtask has been interrupted by TCPIP, on completion of the SOCKET function.

6 shows the subtask is posting the ECB for the CICS task, to wake it up after the SOCKET call.

7 shows the subtask returning the reply from the SOCKET call.

8 shows the exit point from the subtask.

9 shows the subtask waiting on its own ECB, which will be posted by the TRUE on the next socket API call.

10.2.1.3 Using CEDF with the Sockets Interface

Another method of debugging your socket applications uses a combination of the CICS diagnostic facility transaction, CEDF, and the Sockets Interface Security Exit (EZACICSE).

We can write a simple security exit that sets the security switch to 1, and sets the terminal ID to a CICS termid. This forces the child server transaction to be started against the specified CICS terminal. In the examples given in Figure 111 on page 115 and in Figure 112 on page 115 we are forcing the child server to run against terminal 051C, assuming that the terminal is acquired and available.

To debug the child server using CEDF we would also have to enable CEDF at this terminal, either by typing CEDF at 051C, or by typing CEDF 051C at any other panel. Having enabled CEDF at 051C, we can execute the server transaction one step at a time, as a normal terminal-attached transaction. Note that the Security Exit is invoked only for requests that go through the Listener transaction. You have to make alternative arrangements if you replace the Listener with your own transaction, or if you are using an iterative server. A sample Security Exit program to do this is shown in Figure 111 on page 115.
IDENTIFICATION DIVISION.
PROGRAM-ID. EZACICSE.
ENVIRONMENT DIVISION.
DATA DIVISION.
*
WORKING-STORAGE SECTION.
*
LINKAGE SECTION.
01 DFHCOMMAREA.
   05 SECURITY-TRANID PIC X(4).
   05 SECURITY-DATA PIC X(40).
   05 SECURITY-ACT PIC X(2).
   05 SECURITY-IC-TIME PIC 9(6).
   05 SECURITY-FAMILY PIC 9(4) COMP.
   05 SECURITY-PORT PIC 9(4) COMP.
   05 SECURITY-ADDR PIC 9(8) COMP.
   05 SECURITY-SWITCH PIC X(1).
   05 SECURITY-MSG PIC X(1).
   05 SECURITY-TERMID PIC X(4).
   05 SECURITY-SOCKETD PIC 9(4) COMP.

PROCEDURE DIVISION.

   MOVE '051C' TO SECURITY-TERMID.
   MOVE '1' TO SECURITY-SWITCH.

   EXEC CICS RETURN END=EXEC.
   GOBACK.

* Sample security exit for debugging.
* CEDF should be enabled on CICS terminal 051C
*
IDENTIFICATION DIVISION.
PROGRAM-ID. EZACICSE.
ENVIRONMENT DIVISION.
DATA DIVISION.
*
WORKING-STORAGE SECTION.
*
LINKAGE SECTION.
01 DFHCOMMAREA.
   05 SECURITY-TRANID PIC X(4).
   05 SECURITY-DATA PIC X(40).
   05 SECURITY-ACT PIC X(2).
   05 SECURITY-IC-TIME PIC 9(6).
   05 SECURITY-FAMILY PIC 9(4) COMP.
   05 SECURITY-PORT PIC 9(4) COMP.
   05 SECURITY-ADDR PIC 9(8) COMP.
   05 SECURITY-SWITCH PIC X(1).
   05 SECURITY-MSG PIC X(1).
   05 SECURITY-TERMID PIC X(4).
   05 SECURITY-SOCKETD PIC 9(4) COMP.

PROCEDURE DIVISION.

   MOVE '051C' TO SECURITY-TERMID.
   MOVE '1' TO SECURITY-SWITCH.

   EXEC CICS RETURN END=EXEC.
   GOBACK.

Figure 111. Sample Security Exit to Start a Server at a Terminal

If your server transactions are invoked using the transient data mechanism provided by the Listener, you need to alter your DCT entry to get the same effect as using the Security Exit. The Terminal-ID returned by the Security Exit is ignored by the Listener if the server is started using transient data. The modifications shown in Figure 112 must be made to the DCT entry:

SRI1 DFHDCT TYPE=INTRA,
      TCP/IP ENTRY FOR
      DESTID=SRI1,
      DESTFAC=TERMINAL,051C,
      TRIGLEV=1,
      TRANSID=SRI1

Figure 112. Modified DCT Entry to Start a Server against a Terminal

When CEDF is started and you begin to execute your sockets application program, you will see CEDF panels on entry to and exit from the TRUE. Figure 113 on page 116 shows a sample CEDF panel for a TAKESOCKET call, just before the program passes control to the TRUE. The individual character parameters for the TAKESOCKET call are highlighted. Note that ARG 000
contains the character string `TCPPIUCVSTREAMS`, and ARG 003 contains the TAKE-SOCKET-CLIENTID.

<table>
<thead>
<tr>
<th>TRANSACTION: SRR1</th>
<th>PROGRAM: SISSRR1C</th>
<th>TASK NUMBER: 0001189</th>
<th>DISPLAY: 00</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS: ABOUT TO EXECUTE COMMAND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL TO RESOURCE MANAGER EZACIC01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001: ARG 000 (&quot;TCPPIUCVSTREAMS&quot;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001: ARG 001 (&quot;............&quot;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001: ARG 002 (&quot;...........CICSRS2A000&quot;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001: ARG 003 (&quot;...........CICSRS2A000&quot;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001: ARG 004 (&quot;............&quot;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001: ARG 005 (&quot;............&quot;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001: ARG 006 (&quot;............&quot;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001: ARG 007 (&quot;............&quot;)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

OFFSET: X'0000636C'  LINE: UNKNOWN  EIBFN = X'109A'

ENTER: CONTINUE
PF1: UNDEFINED   PF2: SWITCH HEX/CHAR   PF3: UNDEFINED
PF4: SUPPRESS DISPLAYS   PF5: WORKING STORAGE   PF6: USER DISPLAY
PF7: SCROLL BACK   PF8: SCROLL FORWARD   PF9: STOP CONDITIONS
PF10: PREVIOUS DISPLAY   PF11: UNDEFINED   PF12: ABEND USER TASK

Figure 113. Sample CEDF Panel for a TAKESOCKET Call

Figure 114 on page 117 shows the same panel with the parameters shown in hexadecimal (accessed by pressing PF2 in CEDF). The individual numeric parameters for the TAKESOCKET call are highlighted. These correspond to the parameters specified on the TAKESOCKET call in the application program (see TAKESOCKET in Appendix B, “The Sockets API for COBOL and Assembler Language Programs” on page 143):

ARG 001 is the COMMANDA field, which is set to 32 (X'20') for a TAKESOCKET call.

ARG 002 is a half-word of zeros.

ARG 003 is TAKE-SOCKET-CLIENTID.

ARG 004 is TAKE-SOCKET-LDESC.

ARG 005 is TAKE-SOCKET-SOCKNO.

On the CEDF panel for the return from the socket call, the parameters ARG 006 and ARG 007 show the TAKE-SOCKET-ERR and TAKE-SOCKET-RET fields, respectively.
10.2.1.4 Message Output by the Sockets Interface

Messages are also output to the CSMT log. These messages indicate normal attach and detach of subtasks, and any exception conditions that may have occurred and caused a subtask to be detached. Figure 115 shows the messages that we receive when enabling the Sockets Interface. The messages indicate the TRUE is enabled, and that a subtask has been attached for the Listener.

Figure 115. Messages Received for a Normal Enable

Normal detach of a subtask gives the message shown in Figure 116.

Figure 116. Message Received for a Normal Disable

10.2.1.5 System Dumps

If you have a situation in which an MVS subtask abends, a dump of the entire region results. The subtask (EZACIC03) does not issue an SDUMP macro in its ESTAE routine, and so the system dump is directed to your CICS job log. In order to analyze useful with these system dumps, you have to redirect them to data sets that can be used with IPCS. To redirect such system dumps, you need to include SYSUDUMP and SYSABEND statements in your CICS startup JCL, as shown in Figure 117 on page 118.
SYSPRINT DD SYSOUT=A
SYSDUMP DD DSN=CICS330.TCPIP.SDUMP,DISP=SHR
SYSABEND DD DSN=CICS330.TCPIP.SDUMP,DISP=SHR
DFHSNAP DD SYSOUT=A
PLIDUMP DD SYSOUT=A
...

Figure 117. SYSUDUMP and SYSABEND Statements in CICS Startup JCL

You will then be able to use the dump data set as an input to IPCS to allow you to format and examine the information. The formatted dump will help if you ever need to contact the support center, as it will enable you to give more information about the problem.

You can find useful information about the socket tasks in the CICS region in the RMI (Resource Manager Interface) control blocks in the formatted dump. Format the dump data set using IPCS, with the following command:

```shell
VERBX DFHPD330 UEH=3
```

where DFHPD330 is the supplied IPCS verbexit (in this case for CICS/ESA 3.3.0).

Search for the characters GWA.EZACIC01 in the formatted dump. This is the heading used for the formatted global work area for the TRUE. The individual task interface element control blocks for CICS socket tasks can be found by searching for ‘TIE.EZACIC01’. The Sockets Interface uses the user-extension of the TIE, which begins at offset +X’64’ into the TIE. Figure 118 on page 119 shows an extract from a formatted output.
GWA.EZACIC01 00062CA4 EPB GLOBAL WORK AREA

<table>
<thead>
<tr>
<th>Offset</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>C5E9C1C3 C93C7E6 C5E9C1C3 C93C7E6</td>
<td><em>EZACICGMEZACIC00</em></td>
</tr>
<tr>
<td>0010</td>
<td>E5F1D9F1 D4F0F0F8 61F3F061 F9F2F1F3</td>
<td><em>VIR1M008/30/9213</em></td>
</tr>
<tr>
<td>0020</td>
<td>4BF3F240 4040C5E9 C1C3C9C3 F0F1E5F1</td>
<td><em>.32 EZACICD1V1</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0160</td>
<td>C5E9C1C3 C93C7E6</td>
<td>*EZACIC03TCP/IP *</td>
</tr>
<tr>
<td>0170</td>
<td>00000000 0000030C 0000015C</td>
<td><em>0008C500</em></td>
</tr>
<tr>
<td>0180</td>
<td>00000016 F0F3F0F0 0000030C</td>
<td><em>.0330....</em></td>
</tr>
</tbody>
</table>

TIE.EZACIC01 0006C030 Task Interface Element

<table>
<thead>
<tr>
<th>Offset</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00000000 00000000 00000000 E3C9C548</td>
<td><em>...........TIE.</em></td>
</tr>
<tr>
<td>0010</td>
<td>1F85924 0000030C</td>
<td><em>.8.EZAC</em></td>
</tr>
<tr>
<td>0020</td>
<td>C93C7E6F1 00000000 00000000 A743184E</td>
<td><em>IC01....X...</em></td>
</tr>
<tr>
<td>0030</td>
<td>86F1B181 40404040 00000000 40000000</td>
<td><em>f..a .......</em></td>
</tr>
<tr>
<td>0040</td>
<td>00000000 80000000 00000000 00000000</td>
<td><em>.000000000...</em></td>
</tr>
<tr>
<td>0050</td>
<td>04F07078 00000104 0007C030 0003F15C</td>
<td><em>.0.........1</em>*</td>
</tr>
<tr>
<td>0060</td>
<td>01BC01DE C5E9C1C3 C93C7E6C9 C5E9C1C3</td>
<td>*.<em>EZACICTIEZAC</em></td>
</tr>
<tr>
<td>0070</td>
<td>C93C7E6F1 E5F1D9F1 D4F0F0F8 61F3F061</td>
<td><em>IC01V1R1M008/30/</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01A0</td>
<td>00000000 00000000 00000000 0006C524</td>
<td><em>...........E.</em></td>
</tr>
<tr>
<td>01B0</td>
<td>0062CA4 0000C166 E2C3D4C3 C93C7E6C1</td>
<td><em>..u.AWSOMCICSA</em></td>
</tr>
<tr>
<td>01C0</td>
<td>00000000 0000C166 E2D2D300 00030C3E</td>
<td><em>.CSH...T</em></td>
</tr>
<tr>
<td>01D0</td>
<td>04C5F0F3 61F3F061 F9F3F1F3 7AF2F77A</td>
<td><em>E003/30/9313:27:</em></td>
</tr>
<tr>
<td>01E0</td>
<td>F5F60000 00172420 00000000 00171757</td>
<td><em>56.........</em></td>
</tr>
<tr>
<td>01F0</td>
<td>00000000 00171753 308C8258 008C00A0</td>
<td><em>.000000000...</em></td>
</tr>
<tr>
<td>0200</td>
<td>008F15D8 005473B8 00A93770 0054B400</td>
<td><em>...Q....z...</em></td>
</tr>
<tr>
<td>0210</td>
<td>00028000 0054DC00 00084000 00000000</td>
<td><em>.000000000...</em></td>
</tr>
<tr>
<td>0220</td>
<td>00060000 0000F0F0 00F3F0D3 00107000</td>
<td><em>.000000000...</em></td>
</tr>
<tr>
<td>0230</td>
<td>00000000 00000000 00000000 00000000</td>
<td><em>.000000000...</em></td>
</tr>
<tr>
<td>0240</td>
<td>B598</td>
<td><em>.q</em></td>
</tr>
</tbody>
</table>

Figure 118. GWA and TIE Extracts from a Formatted CICS System Dump

- 1 shows the number of port at which the Listener is bound (at offset +X ’17C’). At offset +X’17E’ we have the TRUE status indicator for the Sockets Interface. This field can have the following values:
  - **E** - The TRUE is enabled.
  - **I** - An immediate disable is in operation.
  - **Q** - A quiescent disable is in operation.
  - **S** - The Listener has detected that CICS is in shutdown.

- 2 shows two status fields in the TIE for a CICS sockets task. At offset +X’1B6’ we have the TRUE status for the task. This field can have the following values:
  - **A** - The TRUE is available for this task.
  - **D** - The subtask is detached and the TRUE is disabled for this task.
  - **E** - The subtask has detected errors and is detached, and the TRUE is disabled for this task.
  - **D** - The subtask has abended and the TRUE is disabled for this task.

At offset +X’1B7’ we have the application program status. This field can have the following values:
10.2.2 Problem Determination Methods for the TCP/IP Network

In this section we describe some of the problem determination methods available for TCP/IP for MVS that are applicable to the Sockets Interface. These methods enable you to check the status of the TCP/IP resources being used by your applications.

10.2.2.1 Ensuring TCPIP on MVS is Active

If you have enabled the Sockets Interface, but you get an error message from the Listener indicating that the BIND call failed, you may have a problem with the TCPIP address space. If you have specified the correct TCPIPUSERID and PORT number, the BIND parameters will have been correct, so the problem lies outside the Sockets Interface. From a TSO session, use the SDSF Display Active utility to verify that TCPIP is active in the MVS region. If TCPIP does not appear in the active list, then you must start it with:

```
/s tcpip
```

If TCPIP is active, check its log for error messages.

To verify that the TCPIP address space is working correctly, try issuing a TCPIP command such as HOMETEST. This will verify the TCPIP installation and configuration. Any errors should be checked using the TCP/IP Version 2 Release 2.1 for MVS: Planning and Customization manual.

10.2.2.2 Checking the Route to the Destination Host

To check the route to a destination host try issuing a PING command. This command sends packets to the specified destination host, which echoes them back. This command is a simple way of testing a route between two hosts. Figure 119 shows the PING command to test the route to the host at IP address 9.20.8.111. In this case we contacted the remote host.

```
PING 9.20.8.111
...
TCPNG015I Ping V2R2.1: Pinging host 9.20.8.111. Use PA1 to interrupt.
TCPNG020I PING: Ping #1 response took 0.605 seconds. Successes so far 1.
***
```

Figure 119. PING Command and Responses - Route Active

If anything is wrong with TCP/IP on either host, with connections to the network, or with the network itself, then you will receive a negative response from PING, as shown in Figure 120 on page 121.
PING 9.33.33.33
***
TCPPING015I Ping V2R2.1: Pinging host 9.33.33.33. Use PA1 to interrupt.
TCPPING021I PING: Ping #1 timed out
***

Figure 120. PING Command and Responses - Route Inactive

10.2.2.3 Using NETSTAT to Check TCPIP Resources
The NETSTAT command can be used to check the status of various TCPIP resources on MVS. A common use is to check current sockets, which is done by issuing the following command:

```
NETSTAT SOCKETS
```

This command displays a list of socket applications that are known to TCPIP, together with any sockets they have. The CICS sockets applications can be distinguished from other socket applications by the subtask naming convention. We recommend that subtasks be named by using the EIBTASKN of the associated CICS task as the first seven characters, followed by an alphabetic character. The Listener’s subtask can be identified because its last letter is L. The subtasks for all Listener-started server transactions end in T. This convention is enforced by the Sockets Interface for Listener-started servers. We advise you to adhere to this convention when naming your own subtasks in application programs, using a suffix of C for clients and S for servers.
Appendix A. The Supplied Sample Applications

In this appendix we describe the sample applications provided with this document and give instructions for using them. We list each of the sample programs, and briefly describe its function. We also describe how to install and execute the samples.

The topics we cover are:
• Sample application programs
• Loading the sample applications from the diskette
• Preparing the sample distributed message system
• Using the sample distributed message system
• Preparing the sample record retrieval system
• Using the sample record retrieval system.

A.1 Sample Application Programs

The two sample applications contain many programs. We chose a naming convention to help distinguish between client and server programs. In this section we describe the naming convention, and then list the programs that make up the two sample applications.

A.1.1 Naming Convention for the Sample Applications

Figure 121 shows the naming convention used.

```
Sltxxxxl : where SI = Socket Interface
t - type : C for Client
          S for Server
          M for Mapset
          U for Utility
xxxx - unique name
l - language : C for COBOL
       D for C
```

Figure 121. Naming Convention for the Sample Applications

A.1.2 The Messaging System

SICCCLTC Part of the CICS sockets client. This program provides the BMS menu panel for the CICS client. The BMS maps used in the CICS client are the following:
• SIMMENU—Initial menu panel
• SIMINQU—Panel to show a list of messages
• SIMREAD—Panel for a single message.
SICCCL1C  The CICS sockets client. This is the CICS version of the client program for this system. It allows the user to write, read and delete messages, and to inquire on and delete message logs.

SISSLOGC  A CICS sockets child server. This server is used to check the server’s log when the client receives an error on the connection during commit processing. It checks the log and returns the status to the client.

SISSRR1C  A CICS sockets child server. This server reads a single message from a user’s mail-log. The client passes the USERID and message number, and the server sends back the appropriate message, or an error if that message did not exist.

SISSRI1C  A CICS sockets child server. This server queries the messages in a mail-log. The client passes the USERID, and the server sends back the message number, subject, and sender for each message in that user’s mail-log.

SISSRD1C  A CICS sockets child server. This server deletes a message from a user’s mail-log. The client passes the USERID and message number, and the server attempts to delete the appropriate message. It returns an error if the message did not exist.

SISSRD2C  A CICS sockets child server. This server deletes an entire mail-log. The client passes the USERID, and the server attempts to delete all the messages for that user. It returns an error if the user’s message log could not be found.

SISSRW1C  A CICS sockets child server. This server writes a message into the user’s mail-log. The client passes the USERIDs of the sender and the intended recipient, the subject and the message body; the server writes the message to the recipient’s message log.

SIULSECC  A CICS utility program. This program is used to set up the contents of the VSAM files used by the Security Exit. It is included to allow you to easily change the contents of these files to match your installation.

EZACICSE  The CICS Sockets Interface Security Exit used with the sample application. It is invoked for each connection request to the Listener. It checks the client’s host, the client’s user, and the requested server. It uses CICS user data tables to hold security information.

SICOS2AD  An OS/2 sockets client program. This is the OS/2 version of the client program for this system. It offers the user access to the server transactions running on the CICS host. It provides functions for writing and reading messages, for inquiring on messages, and for deleting individual messages or entire message logs.

SICAIXAD  An AIX sockets client program. This is the AIX version of the client program for this system. It offers the user access to the server transactions running on the CICS host. It provides functions for writing and reading messages, inquiring on messages, and deleting individual messages or entire message logs.
A.1.3 The Record Retrieval System

The following is a list of the programs that are part of this sample application:

**SICUCLTC**  
A CICS sockets client. This is used both with the sample iterative server (SISUSRVC), and the sample concurrent server (SISUCSVC) and its child server (SISUSV2C). SICUCLTC uses the server to read a record from the CICS sample FILEA VSAM file. This demonstrates simple CICS to CICS communication over TCP/IP.

**SISUSRVC**  
A CICS sockets iterative server. This server is independent of the IBM-supplied Listener transaction. It is started from a terminal; the user specifies a port number for it to bind to. It services requests from client program SICUCLTC.

**SISUCSVC**  
A CICS sockets concurrent server. This concurrent server receives connection requests from the client program SICUCLTC, and then starts transaction USSV (child server program SISUSV2C) to service the request.

**SISUSV2C**  
A CICS sockets child server. It is started by concurrent server SISUCSVC. It services requests from the client program SICUCLTC.

**SICUCCLD**  
A C/370 CICS sockets client. This is the C equivalent of SICUCLTC. It uses the server SISUCS1D to read a specific record from the CICS sample FILEA VSAM file.

**SISUCS1D**  
A C/370 CICS sockets iterative server. This is the C equivalent of SISUSRVC. It handles requests from SICUCCLD to read the CICS sample FILEA.
A.2 Loading the Sample Applications from the Diskette

In this section we describe how to move the sample programs and definitions from the diskette onto MVS, OS/2 and AIX. The record retrieval system runs entirely on CICS. The server part of the messaging system also runs on CICS, but we have provided clients that run on CICS, OS/2 and AIX.

A.2.1 Contents of the Diskette

The sample programs are provided on the supplied diskette. They have been placed in two directories, named MESSAGE and RETRIEVE.

- MESSAGE—contains the programs for the sample distributed message system:
  - SICCCLTC
  - SICCCL1C
  - SISSLOGC
  - SISSRR1C
  - SISSRD1C
  - SISSRD2C
  - SISSRW1C
  - SISSRI1C
  - SIULSECC
  - EZACICSE
  - SIMMENU
  - SIMINQU
  - SIMREAD
  - DEFUSER
  - DEFSEC1
  - DEFSEC2
  - DEFSEC3
  - TCPTST
  - TCPTCT
  - TCPDCT
  - CSDUPB
  - WORKSTN—subdirectory containing the workstation source code:
    - SICOS2AD.C
    - SICAIXAD.C

- RETRIEVE—contains the programs for the sample record retrieval system:
  - SICUCLTC
  - SISUSRVC
  - SISUCSVC
  - SISUSV2C
  - SICUCLD
  - SISUCS1D
  - CSDUPA

A.2.2 Transferring the Sample Applications from the Diskette

Part of the contents of the diskette need to be transferred to your CICS system; other components will need to be moved onto your PS/2 or your RISC System/6000.
A.2.2.1 Transferring CICS Components to MVS

If you have TCP/IP installed on a PS/2, use FTP to transfer the code from the diskette to MVS. If you are using OS/2, you should follow a dialog similar to that shown in Figure 122, which shows the process to transfer the files required for the record retrieval system. You should repeat this process for the messaging system, using the following command to switch to the correct directory on the diskette:

```bash
lcd a:\message
```

In this case the diskette is in drive A:

```
[C:\]ftp 9.12.15.5
Connected to 9.12.15.5
220--FTPSEX IBM MVS V2R2 at WSCMA.ITSCPOK.IBM.COM, 17:12:03 on 04/07/93
220 Connection will close if idle for more than 5 minutes.
Name (9.12.15.5): cicsrs1
331 Send password please.
Password:
230 CICSRS1 is logged on.
ftp> lcd a:\retrieve
1 Local directory now A:\RETRIEVE
ftp> cd cicsrs1.jcl.data
257 +CICSRS1.JCL.DATA+ partitioned data set is working directory.
ftp> prompt
Interactive mode off.
ftp> mput *
200 Port request OK.
125 Storing data set CICSRS1.JCL.DATA(CSDUPA)
250 Transfer completed successfully.
local: csdupa remote: csdupa
3741 bytes sent in 0.34 seconds (10 Kbytes/s)
... ftp> close
221 Quit command received. Goodbye.
ftp> bye
[C:\]
```

Figure 122. Transferring Files to MVS Using FTP

The important points to note are the following:

1. You must change the local directory to the required directory on the diskette.
2. You must change the remote directory to be a partitioned data set (PDS), so that the files you send to MVS are stored as members of that PDS.
3. You can use a multiple-put (MPUT) to transfer all of the files in the local directory. If you turn off 'Interactive mode', then you are not prompted to confirm the transfer request for each file.

A.2.2.2 Transferring Workstation Components to OS/2

To transfer the workstation client source code to OS/2, issue an OS/2 COPY command for SICOS2AD.C from the WORKSTN directory to your working directory, as shown in the following example:

```
[C:\]copy a:\message\workstn\sicos2ad.c c:\work
```
A.2.2.3 Transferring Workstation Components to AIX

Use FTP to transfer SICAIXAD.C from the WORKSTN directory to an AIX workstation.

A.3 Preparing the Sample Distributed Message System

Our sample application consists of two parts: the code for the CICS server, and the code for the clients (CICS, OS/2 and AIX). Figure 123 on page 129 shows an example of one configuration that it is possible to implement using the supplied code.

All of the CICS server transactions are started through the Listener. The server which queries the messages in a user’s mail-log, SRI1, is started by means of the Listener’s transient data mechanism. All other server transactions are started through the Listener’s task control option. The Security Exit is used in the server CICS region to restrict access to the server transactions.

All of the CICS source code is in COBOL; the source code for the AIX and OS/2 clients is in C.

A.3.1 Defining the Sample Message System to CICS

To create the necessary CSD definitions for the CICS resources you must run the CSDUPB job provided. The CSDUPB job creates the CSD definitions for the CICS clients and the CICS servers, so it must be run against the CSDs of all CICS regions involved. Once you have run CSDUPB you should install the CSD group with the required program, transaction and file definitions (this is group TCPIPU if you use the definitions we give in CSDUPB). Remember to install the definitions in both the client (if you want to use the CICS client sample) and the server regions.

Add the definitions provided in TCPTST, TCPTCT and TCPDCT to your own TST, TCT and DCT definitions. Reassemble these tables.
A.3.2 Preparing the CICS Server Region

To prepare the CICS server region for the messaging system, perform the following steps:

1. Run the DEFUSER, DEFSEC1, DEFSEC2 and DEFSEC3 jobs to create the VSAM files needed for the server and the Security Exit.

2. Compile and link-edit SISSLOGC, SISSRD1C, SISSRD2C, SISSRI1C, SISSRR1C and SISSRW1C as CICS COBOL sockets applications.

3. Modify the source of SIULSECC to match your installation. You must alter the IP current addresses defined in CONN-REC, as shown in Figure 124 on page 130, to match those of your host machines. You may also want to modify the USERIDs which are defined in USER-REC.
Figure 124. Data Needing Modification in Program SIULSECC

4. Compile and link-edit EZACICSE and SIULSECC as normal CICS COBOL programs.

5. Execute the LSEC transaction to set up the contents of the VSAM files used by the Security Exit.

6. Use the CECI transaction to set up the AUXQUEUE temporary storage queue:

```cobol
CECI WRITEQ TS QUEUE(AUXQUEUE) FROM(00)
```

This TS queue holds the current number of users in the TCPUSER file. Because there are no users at the start, we initialize this queue to 0.

7. Enable the Sockets Interface on the server’s CICS region.

### A.3.3 Preparing the CICS Client

To prepare the CICS client region for the messaging system, perform the following steps:

1. Assemble the BMS maps SIMMENU, SIMINQU and SIMREAD.

2. Compile and link-edit SICCCLTC as a normal CICS COBOL application program. Ensure that you include the data set containing the COBOL copybooks for the BMS maps in the SYSLIB concatenation for the COBOL compilation.

3. Compile and link-edit SICCCL1C as a CICS COBOL sockets application program. Ensure that you include the data set containing the COBOL copybooks for the BMS maps in the SYSLIB concatenation for the COBOL compilation.

When the CICS client is invoked a BMS map is displayed. The default IP address (hexadecimal format) and port number (decimal format) of the server system are displayed in the top right corner of the map. You can alter these default values by finding and changing the SENDMAP-ROUTINE in program SICCCLTC. Figure 125 shows the default values in SENDMAP-ROUTINE for our system.

```
SENDMAP-ROUTINE.
  MOVE &90C0F05 TO CONNIPI.
  MOVE &3000 TO PORTNBI.
```

Figure 125. Default Values in SENDMAP-ROUTINE
If you do alter the default values in SICCCLTC, remember to recompile the program and use the CEMT SET PROGRAM(SICCCLTC) NEWCOPY command to refresh the working version.

A.3.4 Preparing the OS/2 Client

To prepare the OS/2 client for the messaging system, perform the following steps:

1. Compile and link-edit program SICOS2AD.C using a suitable OS/2 C compiler. We used the Microsoft C6 compiler, using the commands shown in Figure 126.

   ```
   cl /dos2 /c /al sicos2ad.c
   link sicos2ad,sicos2ad.exe,,tcip.lib,/st:8192
   ```

   *ERROR* a - EDC0322 Type of the parameter name cannot conflict with previous declaration of function connect.
   *INFORMATIONAL* a - EDC0142 Previous declaration has type _Seg16 pointer to void .
   *INFORMATIONAL* a - EDC0145 Redeclaration has type pointer to structure sockaddr.

   You should find that the program as supplied will compile cleanly if your TCP/IP system under OS/2 is using the 16-bit include libraries. If you have applied the latest corrective service diskette to TCP/IP, then the libraries will have been updated to support 32-bit applications. In this case you may see the compilation messages shown in Figure 127.

   Figure 127. Error Compiling Supplied OS/2 Client Program

   You should find that the compilation will complete; our testing showed that the program will still execute correctly.

2. Ensure TCP/IP is active under OS/2.

A.3.5 Preparing the AIX Client Program

To prepare the AIX client for the messaging system, perform the following steps:

1. Compile and link-edit program SICAIXAD.C. For example, you may use the command shown in Figure 128.

   ```
   cc -o sicaixad -I /usr/include sicaixad.c
   ```

   You should find that the compilation will complete; our testing showed that the program will still execute correctly.

2. Ensure TCP/IP is active under AIX.

A.4 Using the Sample Distributed Message System

In this section we describe how to invoke each of the client programs, and then give examples of using these clients to create and retrieve messages.
A.4.1 Starting the CICS Client
To start the CICS client enter:

```
CCLT-FRED
```

at a terminal connected to the client CICS region. FRED is any authorized USERID (maximum of eight characters) that has been defined in USER-REC, as shown in Figure 124 on page 130. A BMS map will be displayed. The default IP address (hexadecimal format) and port number (decimal format) of the server system are displayed in the top right corner of the map. Type over these fields to change them if you did not change the default values.

A.4.2 Starting the OS/2 Client
To start the OS/2 client enter:

```
sicos2ad 9.12.15.5 3000 FRED
```
in a window under OS/2. 9.12.15.5 is the IP address of the server’s host, 3000 is the port number at which the server’s Listener transaction is bound, and FRED is any authorized USERID (maximum of eight characters).

A.4.3 Starting the AIX Client
To the AIX client enter:

```
sicaixad 9.12.15.5 3000 FRED
```
where 9.12.15.5 is the IP address of the server’s host, 3000 is the port number at which the server’s Listener transaction is bound, and FRED is any authorized USERID (maximum of eight characters).

A.4.4 Example of Using the Sample Clients
In this section we follow a sample execution of the distributed message system. We cover all of the supported platforms: CICS/ESA, OS/2 and AIX. We assume that all of the required programs and resources have been installed on the required systems, and that the Sockets Interface is enabled on the client and server CICS regions.

The function of the messaging system is described in Section 6.1.1, “Function of the Messaging System” on page 51. These are the action codes that you should use on each client to invoke the server function that you want:

- **W or w** Write one message to another user’s mail-log
- **R or r** Read one message from a mail-log
- **I or i** Inquire on a mail-log
- **D or d** Delete one message
- **A or a** Delete a mail-log.
We start by using the CICS client transaction.

CCLT-FRED

The panel shown in Figure 129 is displayed.

```
OPERATOR INSTRUCTIONS CONN-IP-ADDR: 090C0F05
PORT NUMBER.: 3000
MESSAGE INQUIRY  - ENTER  I
MESSAGE READ     - ENTER  R AND NUMBER
MESSAGE WRITE    - ENTER  W TO-NAME SUBJECT AND TEXT
MESSAGE DELETE ONE - ENTER  D AND NUMBER
MESSAGE DELETE ALL - ENTER  A

PRESS CLEAR TO EXIT

ENTER FUNCTION: NUMBER.: TO-NAME..: SUBJECT:

TEXT......:

INFORMATION MESSAGE.: CONNECTION MESSAGE..:
```

Figure 129. CICS Client: Initial Panel
We send a message to BARNEY. To do this we enter W as the function, BARNEY as the TO-NAME, and then give a subject and text for our message, as shown in Figure 130.

```
OPERATOR INSTRUCTIONS

CONN-IP-ADDR: 090C0F05
PORT NUMBER.: 3000

MESSAGE INQUIRY    - ENTER I
MESSAGE READ        - ENTER R AND NUMBER
MESSAGE WRITE       - ENTER W TO-NAME SUBJECT AND TEXT
MESSAGE DELETE ONE  - ENTER D AND NUMBER
MESSAGE DELETE ALL  - ENTER A

PRESS CLEAR TO EXIT

ENTER FUNCTION: W
NUMBER.: 
TO-NAME.: BARNEY
SUBJECT: SUBJECT 1
TEXT......: THIS IS THE FIRST MESSAGE FROM FRED...

INFORMATION MESSAGE.: 
CONNECTION MESSAGE.: 
```

Figure 130. CICS Client: Write a Message to BARNEY

Now we start a client on OS/2, and enter function I to inquire on BARNEY's mail-log, as shown in Figure 131.

```
[C:\SOURCE\REDBOOK]sicos2ad 9.12.15.5 3000 BARNEY
Hello <BARNEY >
Which function ? : I

-------> Connection opened.

SRI1,BARNEY ,TD
Length - 16
287 bytes received
Output : 0001FRED SUBJECT 1

No more messages...

-------> Connection closed.

Which function ? :
```

Figure 131. OS/2 Client: Query BARNEY's Mail-log
We read the message from FRED by entering R, as shown in Figure 132. The client prompts us for the message number we want to read, and we enter that number. Note that our clients expect a four-digit number. You cannot get message 0001 by just entering 1.

```
Which function ? : R
Message # : 0001

--------> Connection opened.

SRR1,BARNEY 0001
Length - 18
287 bytes received

From : FRED
To : BARNEY
Subject : SUBJECT 1
Message : THIS IS THE FIRST MESSAGE FROM FRED...

--------> Connection closed.
```

Figure 132. OS/2 Client: Read a Message from BARNEY’s Mail-log

We send a message to WILMA by entering W, as shown in Figure 133. The client prompts us to enter the USERID of the person we want to send a message to, the subject of the message, and the message text.

```
Which function ? : W
To : WILMA
Subject : SUBJECT 2
Message : THIS IS A MESSAGE FROM BARNEY TO WILMA...

--------> Connection opened.

SRW1,BARNEY
Length - 13
2 bytes received
Output : OK
80 bytes of data sent
21 bytes received
Output : 0206PREPARE-TO-COMMIT
sending ::prepare-ok;
The update was successfully committed

--------> Connection closed.
```

Figure 133. OS/2 Client: Write a Message to WILMA
Now we use the AIX client, and inquire on WILMA's mail-log, as shown in Figure 134.

```
$ sicaixad 9.12.15.5 3000 WILMA
Hello <WILMA>
Which function ? : I

 ------> Connection opened.

SRI1,WILMA ,TD
Length - 16
90 bytes received
Output : 0001BARNEY SUBJECT 2

No more messages...

 ------> Connection closed.

Which function ? :
```

Figure 134. AIX Client: Query WILMA's Mail-log

We send a message to FRED by entering W, as shown in Figure 135.

```
Which function ? : W
To : FRED
Subject : SUBJECT 3
Message : THIS IS A NEW MESSAGE FOR FRED

 ------> Connection opened.

SRW1,WILMA
Length - 13
2 bytes received
Output : OK
66 bytes of data sent
21 bytes received
Output : 0208PREPARE-TO-COMMIT
sending $prepare-ok$
The update was successfully committed

 ------> Connection successfully committed.

Which function ? :
```

Figure 135. AIX Client: Write a Message to FRED
Returning to the CICS client, we inquire on FRED’s mail-log. The server tells us that there is a message from WILMA, as shown in Figure 136.

---

**MESSAGE INQUIRY**

THE FOLLOWING MESSAGES ARE DESTINED TO: FRED

<table>
<thead>
<tr>
<th>NUMBER</th>
<th>FROM</th>
<th>SUBJECT</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>WILMA</td>
<td>SUBJECT 3</td>
<td></td>
</tr>
</tbody>
</table>

PRESS CLEAR TO EXIT

INFORMATION MESSAGE.: NO MORE MESSAGE

CONNECTION MESSAGE.: CLOSE   ERNO=00000000 RCODE=00000000

---

*Figure 136. CICS Client: Query FRED’s Mail-log*

We read the message from WILMA. The server displays the message, as shown in Figure 137.

---

**MESSAGE DISPLAY**

<table>
<thead>
<tr>
<th>TO..........: FRED</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER........: 0001</td>
</tr>
</tbody>
</table>

MESSAGE FROM....: WILMA

SUBJECT.........: SUBJECT 3

TEXT...........: THIS A NEW MESSAGE FOR FRED

PRESS CLEAR TO EXIT

INFORMATION MESSAGE.: |

CONNECTION MESSAGE.: CLOSE   ERNO=00000000 RCODE=00000000

---

*Figure 137. CICS Client: Read the Message from WILMA*
The error detection code in our examples does not cover every possible eventuality. For example, the CICS client will terminate if you try to read a message that does not exist.

### A.5 Preparing the Sample Record Retrieval System

This section explains how to prepare the six programs that make up the record retrieval system. These sample programs demonstrate an iterative server and client, as shown in Figure 138, and a concurrent server and client, as shown in Figure 139.

![Figure 138. Iterative Server](image-url)

![Figure 139. Concurrent and Child Servers](image-url)

We provide samples for both cases in COBOL. We also supply a sample written in C for the case shown in Figure 138.

You need to ensure that the CICS sample VSAM file FILEA is installed in the server CICS region. The Installation Guide for your release of CICS tells you how to install FILEA if you have not already done this.
A.5.1 Defining the Record Retrieval System to CICS

To create the necessary CSD definitions for the CICS resources you must run the CSDUPA job provided. The CSDUPA job creates the CSD definitions for the CICS clients and the CICS servers. It must be run against the CSDs of all the CICS regions you are using to run the sample application.

After you have run the CSDUPA job you should install the CSD group with the required program and transaction definitions (group TCPIPU if you use the definitions we give in CSDUPA). Remember to do this in the client region as well as in the server region if you are using two regions. You can execute these samples on a single CICS region.

A.5.2 Preparing the CICS Programs

To prepare the CICS servers for the record retrieval system, perform the following steps:

1. Compile and link-edit the sample COBOL programs as CICS sockets applications. The sample COBOL programs are the following:
   - SICUCLTC (client)
   - SISUSRVC (iterative server)
   - SISUCSVC (concurrent server)
   - SISUSV2C (child server).

2. Modify the source of programs SICUCLTD and SISUCS1D and replace
   \[
   \text{char ip_addr}[9]=\{9.12.15.5\};
   \]
   with the IP address of the host on which the server CICS region is running.

3. Compile and link-edit the sample C/370 programs as CICS sockets applications. The sample C/370 programs are the following:
   - SICUCLLD (client)
   - SISUCS1D (iterative server).

4. Install the CSD group (TCPIPU) with the required program and transaction definitions.

5. Enable the Sockets Interface on the CICS server region.

6. Enable the Sockets Interface on the CICS client region, unless the clients and servers are running in the same CICS region.

A.6 Using the Sample Record Retrieval System

It is possible to have the COBOL concurrent server, the COBOL iterative server and the C/370 iterative server active concurrently. We recommend that you either run one at a time, or make a careful note of which server is active at which port.

A.6.1 Starting the COBOL Iterative Server

To start the COBOL iterative server enter:

```
USRV-4000
```

at a terminal connected to the server CICS region. 4000 is the port number (decimal format) to which the server will bind.
The iterative server has been designed so that once it has been started it executes in a loop. This loop just accepts client requests, performs the function, sends the result, closes the socket and then goes back to accept another client connection request. At no stage does it check to see if the Sockets Interface is being closed down. In order to stop this transaction you must purge it.

A.6.2 Starting the C/370 Iterative Server

To start the C/370 iterative server enter:

```
UCS1-4050
```

at a terminal connected to the server CICS region. 4050 is the port number (decimal format) that the server will bind to.

There is no checking in the C server for invalid record IDs.

The C/370 iterative server, like the COBOL iterative server, has been designed so that once it has been started it executes in a loop. To stop this transaction you must purge it.

A.6.3 Starting the COBOL Concurrent Server

To start the COBOL concurrent server enter:

```
UCSV-4000
```

at a terminal connected to the server CICS region. 4000 is the port number (decimal format) to which the concurrent server will bind.

The sample concurrent server, unlike the iterative servers, checks to see whether the Sockets Interface is being quiesced. It performs this check, as shown in Figure 140, whenever it exits from a SELECT call.

```
EXEC CICS INQUIRE TASK LIST
LISTSIZE(NUM-TASKS)
SET(ADDRESS OF TASK-AREA)
SETTRANSID(ADDRESS OF TRAN-AREA)
END-EXEC.
MOVE 0 TO CTR.
INSPECT TRAN-AREA TALLYING CTR
FOR ALL DISABLE-TRAN.

IF CTR NOT EQUAL 0 THEN
   GO TO LEAVE-SERVER.
```

Figure 140. Check whether Sockets Interface Is Being Disabled

We check all transactions running in the system, and look for CSKD. If we find it, then we assume the Sockets Interface is quiescing, and we close all sockets and end the transaction.

Note: You can only use this mechanism in socket transactions that use the SELECT call with the timeout facility. The SELECT call allows you periodically give control back to your program to enable it to check the status of the CICS region (for example, to see if CSKD is running) regardless of whether or not there is any socket activity.
A.6.4 Starting the C/370 Client

The C/370 client can only be used with the C/370 iterative server. It will not communicate with either of the COBOL servers in this sample system.

To start the C/370 client enter:

UCCL-4050-000111

at a terminal connected to the client CICS region. 4050 is the port number (decimal format) at which the server is bound, and 000111 is the record ID that you wish to read from FILEA. Remember that for this sample you do not have the opportunity to specify the host IP address when you invoke the client, and therefore you need to modify the source of programs SICUCLD and SISUCS1D and replace

char ip_addr[9]=(9.12.15.5);

with the IP address of the host on which the server CICS region is running. The client CICS region may be the same as the server CICS region.

Both client and server output response codes to the CSMT destination. If you find any problems in executing the sample, you should look at the messages on CSMT.

The client will display the contents of the record with the specified record ID. The client ends at this stage. To retrieve a second record, you need to re-enter the entire command.

A.6.5 Starting the COBOL Client

The COBOL client can only be used with the COBOL servers. It will not communicate with the C/370 iterative server in this sample system.

To start the COBOL client enter:

UCLT-090C0F05-4000-000111

at a terminal connected to the client CICS region. The client CICS region may be the same as the server CICS region. 090C0F05 is a hexadecimal representation of the IP address of host on which the server is running (9.12.15.5 in this example); 4000 is the port number (decimal format) on that host at which the server is bound; and 000111 is the record ID that you wish to read from FILEA.

The client should display either REC NOT-FND if an invalid record ID was given, or the record contents if a valid record ID was given.

The client ends at this stage. To retrieve a second record, you need to re-enter the entire command.
Appendix B. The Sockets API for COBOL and Assembler Language Programs

In this appendix we give more detailed information on the sockets API for COBOL and Assembler language programs provided with the Sockets Interface. It expands upon the information given in the Sockets Interface for CICS - Using TCP/IP Version 2 Release 2 for MVS: User's Guide, to which you should refer for information on the format of C language calls.

We list each call in alphabetical order, explain its function and parameters, and give an example of how to use it. At the end of the appendix, we list the three utility modules; EZACIC04, EZACIC05 and EZACIC06, explain their function and parameters, and give examples of how to use them.

TCP/IP Version 2 Release 2.1 for MVS provides two kinds of socket interfaces: C sockets, which provides a standard socket API for MVS C/370 application programs, and the IUCV interface into TCPIP. This interface is language-independent, allowing you to write socket applications in a language other than C.

The sockets API for COBOL and Assembler is based on the IUCV Sockets Interface. The Sockets Interface is a set of routines which map onto the IUCV socket calls. Applications that use the Sockets Interface invoke these routines through a call-interface.

Note: The C API provided with the CICS to TCP/IP Sockets Interface is based on the COBOL call-interface, and hence the IUCV interface. It is not based on the C socket interface supplied with TCP/IP Version 2 Release 2.1 for MVS. The C header stub (EZACIC07) supplied with the Sockets Interface is a set of C functions that map the standard C socket parameters onto the parameters required for the call-interface.

All sockets are stream (TCP) sockets. Datagram (UDP) sockets and connections are not supported in the Sockets Interface. The addressing family used for all Sockets Interface sockets is AF_INET (2).
B.1 Socket Calls in COBOL

The following COBOL declaration must be included for each call:

```
01 STANDARD-PARMS.
   02 TOKEN PIC X(16) VALUE 'TCPIPIUCVSTREAMS'.
   02 COMMANDA PIC 9(4) COMP.
   02 SOCKETD PIC 9(4) COMP.
```

**TOKEN**
The TOKEN parameter is required by the Sockets Interface stub (EZACICAL) to allow it to correctly identify socket calls.

**COMMANDA**
The COMMANDA parameter defines which socket call is being requested. The number of parameters expected on the call is dependent on this field.

**SOCKETD**
The SOCKETD parameter, or socket descriptor, represents the current socket. It is assumed that this variable is set to the required socket before a call is made.

Although we have not shown this data declaration in the individual examples, you must remember to include it in your application.

**Note:** Because the sockets API for COBOL and assembler language programs is based on the IUCV API, we must ensure that all parameters are at the correct offsets in the parameter list of each call. Some extra fields are included for this reason (see ACCEPT-ZERO in B.1.2, “ACCEPT” on page 146). The fields are either half-word full-word or double-word in length. Their contents are ignored and are used merely to preserve the alignment of the parameters on the call to IUCV.

The number of parameters on each call is also important. The Sockets Interface expects a certain number of parameters for each call. Altering this number in a call will cause an AEY9 abend in your application.

B.1.1 Ordered List of Socket Calls

The following is a list of the socket calls, in order of the COMMANDA parameter value.

- 00 - INITAPI
- 01 - ACCEPT
- 02 - BIND
- 03 - CLOSE
- 04 - CONNECT
- 05 - FCNTL
- 07 - GETHOSTID
- 08 - GETHOSTNAME
- 09 - GETPEERNAME
- 10 - GETSOCKNAME
- 11 - GETSOCKOPT
- 12 - IOCTL
- 13 - LISTEN
• 14 - READ
• 16 - RECVFROM
• 19 - SELECT
• 20 - SEND
• 22 - SENDTO
• 23 - SETSOCKOPT
• 24 - SHUTDOWN
• 25 - SOCKET
• 26 - WRITE
• 30 - GETCLIENTID
• 31 - GIVESOCKET
• 32 - TAKESOCKET
B.1.2 ACCEPT

Description: The ACCEPT call is used by a server to accept connection requests from a client. The call accepts the first connection on the queue of pending connections, defined by the LISTEN call. It creates a new socket and returns it to the calling program. This new socket can then be used by the server to communicate with the client. If there are no outstanding connection requests, and the socket is in blocking mode, then the ACCEPT call blocks the calling program until a connection request arrives. If the socket is in non-blocking mode, then the ACCEPT call returns -1 in RETCODE, and EWOULDBLOCK (35) in ERRNO. The blocking mode of the socket can be controlled using the FCNTL call, described in B.1.6, "FCNTL" on page 152.

Example:

```
01 ACCEPT-PARS.
  02 ACCEPT-ZERO PIC 9(8) COMP.
  02 ACCEPT-NEW PIC 9(8) COMP.
  02 ACCEPT-NAME.
    03 ACCEPT-AF-INET PIC 9(4) COMP.
    03 ACCEPT-PORT PIC 9(4) COMP.
    03 ACCEPT-IP-ADDR PIC 9(8) COMP.
    03 FILLER PIC X(8).
  02 ACCEPT-ERR PIC 9(8) COMP.
  02 ACCEPT-RET PIC 9(8) COMP.

ACCEPT-Routine.
  *
  * TO ACCEPT A CONNECT REQUEST FROM A CLIENT AND CREATE A NEW
  * SOCKET FOR THAT CONNECTION
  *
    MOVE 1 TO COMMANDA.
    MOVE 1 TO ACCEPT-NEW.
    CALL 'EZACICAL' USING TOKEN COMMANDA SOCKETD
      ACCEPT-ZERO
      ACCEPT-NEW
      ACCEPT-NAME
      ACCEPT-ERR
      ACCEPT-RET.

ACCEPT-Routine-exit.
  EXIT.
```

Input Parameters:
- COMMANDA: Must be set to 01
- SOCKETD: The socket on which the accept is done
- ACCEPT-NEW: Number for the new socket to be created during the accept call

Output Parameters:
- ACCEPT-AF-INET: Addressing family, should return 2 (AF_INET)
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCEPT-PORT</td>
<td>Port address of the client</td>
</tr>
<tr>
<td>ACCEPT-IP-ADDR</td>
<td>IP address of the client’s host</td>
</tr>
<tr>
<td>ACCEPT-ERR</td>
<td>ERRNO field</td>
</tr>
<tr>
<td>ACCEPT-RET</td>
<td>RETCODE field. If it is a positive number, this is the number of the new socket. If it is negative, it indicates an error.</td>
</tr>
</tbody>
</table>
B.1.3 BIND

Description: The BIND call binds a unique local name (port address and IP address) to the specified socket. The BIND call is not always necessary. For example, the CONNECT call can be used to complete the bind processing, if desired.

Example:

```assembly
01 BIND-PARMS.
  02 BIND-NAME.
    03 BIND-AF-INET PIC 9(4) COMP VALUE 2.
    03 BIND-PORT PIC 9(4) COMP.
    03 BIND-IP-ADDR PIC 9(8) COMP.
    03 FILLER PIC X(8).
  02 BIND-ERR PIC 9(8) COMP.
  02 BIND-RET PIC 9(8) COMP.

BIND-Routine.
* *
* TO BIND A SOCKET TO PORT 3000 ON THE LOCAL HOST *
*
  MOVE 2 TO COMMANDA.
  MOVE 3000 TO BIND-PORT.
  MOVE 0 TO BIND-IP-ADDR.

  CALL ´EZACICAL´ USING TOKEN COMMANDA SOCKETD
      BIND-NAME
      BIND-ERR
      BIND-RET.

BIND-Routine-Exit.
EXIT.
```

Input Parameters:

- **COMMANDA**: Must be set to 02
- **SOCKETD**: The socket to be bound to the local name
- **BIND-AF-INET**: Addressing family, must be set to 2 (AF_INET)
- **BIND-PORT**: Port address to bind to
- **BIND-IP-ADDR**: Local IP address to bind to. If this is zero, then the default is to bind to the local host.

Output Parameters:

- **BIND-ERR**: ERRNO field
- **BIND-RET**: RETCODE field. Zero is successful; a negative value indicates an error.
B.1.4 CLOSE

**Description:** The CLOSE call shuts down the specified socket. It frees all resources allocated to the socket. If the specified socket refers to an open connection, that connection is closed.

**Example:**

```cobol
01 CLOSE-PARMS.
   02 CLOSE-ZERO PIC X(8).
   02 CLOSE-ERR PIC 9(8) COMP.
   02 CLOSE-RET PIC 9(8) COMP.

CLOSE-Routine.
*
* TO CLOSE A SOCKET AND FREE ITS RESOURCES
*
   MOVE 3 TO COMMANDA.

   CALL ¢EZACICAL¢ USING TOKEN COMMANDA SOCKETD
      CLOSE-ZERO
      CLOSE-ERR
      CLOSE-RET.

CLOSE-Routine-Exit.
EXIT.
```

**Input Parameters:**

- **COMMANDA**
  Must be set to 03

- **SOCKETD**
  The socket to be closed

**Output Parameters:**

- **CLOSE-ERR**
  ERRNO field

- **CLOSE-RET**
  RETCODE field. Zero is successful; a negative value indicates an error.
B.1.5 CONNECT

_Description:_ The CONNECT call attempts to establish a connection between two sockets. The first socket is the one specified in the call, and the second is the socket that is bound to the port address and IP address specified in the call. The CONNECT call is used by a client to connect to a server. If the client has not performed a BIND, then the CONNECT call will also complete any necessary binding.

*Example:*

```plaintext
01 CONN-PARMS.
  02 CONN-NAME.
    03 CONN-AF-INET PIC 9(4) COMP VALUE 2.
    03 CONN-PORT PIC 9(4) COMP.
    03 CONN-IP-ADDR PIC 9(8) COMP.
    03 FILLER PIC X(8).
  02 CONN-ERR PIC 9(8) COMP.
  02 CONN-RET PIC 9(8) COMP.

77 TEMP PIC X(4).

CONNECT-Routine.
  *
  * TO CONNECT TO A SERVER AT PORT 3000 ON A REMOTE HOST WITH
  * IP ADDRESS 9.12.15.5
  *
  MOVE 4 TO COMMANDA.
  MOVE 3000 TO CONN-PORT.
  MOVE X'090C0F05' TO TEMP.
  MOVE TEMP TO CONN-IP-ADDR.

  CALL ¢EZACICAL¢ USING TOKEN COMMANDA SOCKETD
  CONN-NAME
  CONN-ERR
  CONN-RET.

CONNECT-Routine-Exit.
  EXIT.
```

_Input Parameters:_

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMANDA</td>
<td>Must be set to 04</td>
</tr>
<tr>
<td>SOCKETD</td>
<td>The local socket to be connected to the remote server</td>
</tr>
<tr>
<td>CONN-AF-INET</td>
<td>Addressing family, must be set to 2 (AF_INET)</td>
</tr>
<tr>
<td>CONN-PORT</td>
<td>Remote port address to connect to</td>
</tr>
<tr>
<td>CONN-IP-ADDR</td>
<td>Remote IP address to connect to. If this is zero, the default is to bind to the local host</td>
</tr>
</tbody>
</table>

_Output Parameters:_

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONN-ERR</td>
<td>ERRNO field</td>
</tr>
</tbody>
</table>
CONN-RET RETCODE field. Zero is successful; a negative value indicates an error.
B.1.6 FCNTL

**Description:** The FCNTL call can be used to control the blocking mode of a socket. Two commands are available:

- F_GETFL (3)—query the FNDELAY flag, placing the result in the RETCODE field
- F_SETFL (4)—set the FNDELAY flag, using the FCNTL-ARG parameter

If the FNDELAY flag is set for a socket, then that socket is in non-blocking mode. If a socket is in non-blocking mode, and there is no data on that socket, then calls such as RECVFROM, READ and ACCEPT will return -1 in RETCODE, and EWOULDBLOCK (35) in ERRNO.

**Example:**

```plaintext
01 FCNTL-PARMS.
   02 FCNTL-CMD PIC 9(8) COMP.
   02 FCNTL-ARG PIC 9(8) COMP.
   02 FCNTL-ERR PIC 9(8) COMP.
   02 FCNTL-RET PIC 9(8) COMP.

FCNTL-Routine.
* * TO SET THE CURRENT SOCKET TO NONBLOCKING MODE
* *
   MOVE 5 TO COMMANDA.
   MOVE 4 TO FCNTL-CMD.
   MOVE 4 TO FCNTL-ARG.

   CALL *EZACICAL* USING TOKEN COMMANDA SOCKETD
      FCNTL-CMD
      FCNTL-ARG
      FCNTL-ERR
      FCNTL-RET.

FCNTL-Routine-Exit.
   EXIT.
```

**Input Parameters:**

- **COMMANDA**
  - Must be set to 05
- **SOCKETD**
  - The socket in question
- **FCNTL-CMD**
  - The command, either F_GETFL (3) to query the FNDELAY flag, or F_SETFL (4) to set the FNDELAY flag
- **FCNTL-ARG**
  - The argument. To set the FNDELAY flag, changing the socket to non-blocking mode, set FCNTL-CMD to 4, and FCNTL-ARG to 4. To clear FNDELAY, and change the socket to blocking mode, set FCNTL-CMD to 4, and FCNTL-ARG to 0. This parameter is ignored when FCNTL-CMD is set to 3.
Output Parameters:

| FCNTL-ERR        | ERRNO field
|------------------|-------------
| FCNTL-RET        | RETCODE. For F_GETFL, a positive number returns the current setting of FNDELAY (0 or 4). For F_SETFL, zero indicates a successful call. For both commands, a negative value indicates an error. |
B.1.7 GETCLIENTID

**Description:** The GETCLIENTID call returns the identifier by which the calling application is known to the TCPIP address space. For a calling application under CICS, the returned ID always consists of the jobname of the CICS region and the name of the MVS subtask associated with the calling CICS transaction. This call is generally used before issuing a GIVESOCKET call, in order to gather the required information to pass to the child server. See the sample concurrent server SISUCSVC for an example of how this call is used. This call is independent of any other TCP/IP conversations, so we do not need to specify a socket.

**Example:**

```plaintext
01 GETCLIENT-PARMS.
   02 GETCLIENT-HZERO PIC 9(4) COMP.
   02 GETCLIENT-ZERO PIC X(8).
   02 GETCLIENT-ID.
      03 GETCLIENT-DOMAIN PIC 9(8) COMP.
      03 GETCLIENT-NAME PIC X(8).
      03 GETCLIENT-TASK PIC X(8).
      03 FILLER PIC X(8).
   02 GETCLIENT-ERR PIC 9(8) COMP.
   02 GETCLIENT-RET PIC 9(8) COMP.

GETCLIENT-ID-ROUTINE.
   *
   * TO FIND THE CURRENT CICS JOBNAME AND MVS SUBTASK NAME
   *
   MOVE 30 TO COMMANDA.

   CALL 'EZACICAL' USING TOKEN COMMANDA
      GETCLIENT-HZERO
      GETCLIENT-ZERO
      GETCLIENT-ID
      GETCLIENT-NAME
      GETCLIENT-TASK
      GETCLIENT-ERR
      GETCLIENT-RET.

GETCLIENT-ID-ROUTE-EXIT.
   EXIT.
```

**Input-Parameter:**

- **COMANDA** Must be set to 30

**Output Parameters:**

- **GETCLIENT-DOMAIN** Addressing family; should return 2 (AF_INET)
- **GETCLIENT-NAME** CICS jobname
- **GETCLIENT-TASK** Subtask name
- **GETCLIENT-ERR** ERRNO field
- **GETCLIENT-RET** RETCODE field. Zero is successful; a negative value indicates an error.
B.1.8 GETHOSTID

Description: The GETHOSTID call returns the unique 32-bit IP address of the local MVS host. This call is independent of any other TCP/IP conversations, so we do not need to specify a socket.

Example:

```cobol
01 GETHOST-ID-PARMS.
   02 GETHOST-ID-ZERO-H PIC 9(4) COMP.
   02 GETHOST-ID-ZERO-D PIC X(8).
   02 GETHOST-ID-ERR PIC 9(8) COMP.
   02 GETHOST-ID-ID PIC 9(8) COMP.

GETHOST-ID-ROUTINE.
   *
   * TO FIND THE IP ADDRESS OF THE LOCAL MVS HOST
   *
   MOVE 7 TO COMMANDA.

   CALL &EZACICAL USING TOKEN COMMANDA
   GETHOST-ID-ZERO-H
   GETHOST-ID-ZERO-D
   GETHOST-ID-ERR
   GETHOST-ID-ID.

GETHOST-ID-ROUTINE-EXIT.
   EXIT.
```

**Input-Parameter:**

COMMANDA  Must be set to 07

**Output Parameters:**

GETHOST-ID-ERR  ERRNO field
GETHOST-ID-ID  If this is a positive number, it holds the IP address of the local host; a negative value indicates an error.
B.1.9 GETHOSTNAME

**Description:** The GETHOSTNAME call returns the name of the local MVS host. This call is independent of any other TCP/IP conversations, so we do not need to specify a socket.

**Example:**

```
01 GETHOST-NAME-PARMS.
   02 GETHOST-NAME-HZERO PIC 9(4) COMP.
   02 GETHOST-NAME-DZERO PIC X(8).
   02 GETHOST-NAME-LEN PIC 9(8) COMP.
   02 GETHOST-NAME PIC X(255).
   02 GETHOST-NAME-ERR PIC 9(8) COMP.
   02 GETHOST-NAME-RET PIC 9(8) COMP.

GETHOST-NAME-ROUTINE.
  *
  * TO FIND THE NAME OF THE CURRENT MVS HOST
  *
    MOVE 8 TO COMMANDA.
    MOVE 30 TO GETHOST-NAME-LEN.

    CALL ¢EZACICAL€ USING TOKEN COMMANDA
    GETHOST-NAME-HZERO
    GETHOST-NAME-DZERO
    GETHOST-NAME-LEN
    GETHOST-NAME
    GETHOST-NAME-ERR
    GETHOST-NAME-RET.

GETHOST-NAME-ROUTINE-EXIT.
  EXIT.
```

**Input Parameters:**

- **COMMANDA**
  - Must be set to 08

- **GETHOST-NAME-LEN**
  - Length of the expected host name; must be greater than zero, and less than 256.

**Output Parameters:**

- **GETHOST-NAME**
  - Name of the local MVS host. This is the name specified in the HOSTNAME statement in the tcpip.TCPIP.DATA data set.

- **GETHOST-NAME-ERR**
  - ERRNO field

- **GETHOST-NAME-RET**
  - RETCODE field. Zero is successful; a negative value indicates an error.
B.1.10 GETPEERNAME

Description: The GETPEERNAME call returns the name of the calling applications peer, which comprises the internet family, port number and IP address. The peer must be connected to the caller on the specified socket.

Example:

```cobol
01 GETPEER-NAME-PARMS.
  02 GETPEER-NAME-ZERO PIC X(8).
  02 GETPEER-NAME.
    03 GETPEER-AF-INET PIC 9(4) COMP.
    03 GETPEER-PORT PIC 9(4) COMP.
    03 GETPEER-IP-ADDR PIC 9(8) COMP.
    03 FILLER PIC X(8).
  02 GETPEER-NAME-ERR PIC 9(8) COMP.
  02 GETPEER-NAME-RET PIC 9(8) COMP.

GETPEER-NAME-ROUTINE.

* TO FIND THE NAME OF THE PEER APPLICATION
* 
MOVE 9 TO COMMANDA.

CALL $EZACICAL USING TOKEN COMMANDA SOCKETD GETPEER-NAME-ZERO GETPEER-NAME GETPEER-NAME-ERR GETPEER-NAME-RET.

GETPEER-NAME-ROUTINE-EXIT.
EXIT.
```

Input Parameters:
- **COMMANDA**: Must be set to 09
- **SOCKETD**: The socket on which the peer is connected

Output Parameters:
- **GETPEER-AF-INET**: Addressing family; should return 2 (AF_INET).
- **GETPEER-PORT**: Remote port number at which the peer is bound
- **GETPEER-IP-ADDR**: Remote IP address of the peer’s host
- **GETPEER-NAME-ERR**: ERRNO field
- **GETPEER-NAME-RET**: RETCODE field. Zero is successful; a negative value indicates an error.
B.1.11 GETSOCKNAME

Description: The GETSOCKNAME call returns the local port address and IP address to which a socket is bound. If the socket is not yet bound, then the call returns the addressing family, which will always be 2 (AF_INET) for stream sockets.

Example:

```plaintext
01 GETSOCK-NAME-PARMS.
  02 GETSOCK-NAME-ZERO PIC X(8).
  02 GETSOCK-NAME.
    03 GETSOCK-AF-INET PIC 9(4) COMP.
    03 GETSOCK-PORT PIC 9(4) COMP.
    03 GETSOCK-IP-ADDR PIC 9(8) COMP.
    03 FILLER PIC X(8).
  02 GETSOCK-NAME-ERR PIC 9(8) COMP.
  02 GETSOCK-NAME-RET PIC 9(8) COMP.

GETSOCK-NAME-ROUTINE.

* TO FIND THE LOCAL PORT ADDRESS AND IP ADDRESS FOR A SOCKET
*  MOVE 10 TO COMMANDA.

CALL ¢EZACICAL¢ USING TOKEN COMMANDA SOCKETD
  GETSOCK-NAME-ZERO
  GETSOCK-NAME
  GETSOCK-NAME-ERR
  GETSOCK-NAME-RET.

GETSOCK-NAME-ROUTINE-EXIT.

EXIT.
```

Input Parameters:

- COMMANDA: Must be set to 10
- SOCKETD: The socket whose IP address and port number we need

Output Parameters:

- GETSOCK-AF-INET: Addressing family; should return 2 (AF_INET)
- GETSOCK-PORT: Local port number at which the socket is bound
- GETSOCK-IP-ADDR: Local IP address of the caller’s host
- GETSOCK-NAME-ERR: ERRNO field
- GETSOCK-NAME-RET: RETCODE field. Zero is successful; a negative value indicates an error.
B.1.12 GETSOCKOPT

**Description:** The GETSOCKOPT call gets options associated with a socket. All options must be manipulated at the socket level, therefore the GETSOCK-OPT-LEVEL parameter must be set to SOL_SOCKET ('X'0000FFFF').

The options available for GETSOCKOPT are the following:

- **SO_ERROR (X'00001007') (OPT-LEN = 4)**—Returns any pending error on the socket, and also clears the error status.
- **SO_LINGER (X'00000080') (OPT-LEN = 8)**—Lingers on the CLOSE call if there is data present. If this option is set, then the CLOSE call will block until the data is sent. The data returned is two full-words:
  - **ONOFF**—Indicates whether or not the option is set
  - **LINGER**—The time interval for the linger if the option is set
- **SO_OOBINLINE (X'00000100') (OPT-LEN = 4)**—Indicates whether or not out-of-band data is placed in the normal data stream as it is sent or received on the specified socket.
- **SO_REUSEADDR (X'00000004') (OPT-LEN = 4)**—Indicates whether or not an address can be reused. When this option is set, the socket can be bound to a local port that is already in use, or still in a TIME-WAIT state after a CLOSE call.
- **SO_SNDBUF (X'00001001') (OPT-LEN = 4)**—Returns the size of the send-buffer.
- **SO_TYPE (X'00001008') (OPT-LEN = 4)**—Returns the type of the socket. For the CICS to TCP/IP Sockets Interface, this will be 1 (SOCK-STREAM).

All socket options except SO_LINGER return a full-word value. A non-zero value indicates that the option is set. A value of 0 indicates that the option is not set. SO_LINGER returns an indicator (first full-word) and a time-interval (second full-word).
**Example:**

```
01 GETSOCK-OPT-PARMS.
   02 GETSOCK-OPT-LEVEL PIC X(4).
   02 GETSOCK-OPT-NAME PIC X(4).
   02 GETSOCK-OPT-LEN PIC 9(4) COMP.
   02 GETSOCK-OPT-VAL-DW.
      03 GETSOCK-OPT-VAL-1 PIC X(4).
      03 GETSOCK-OPT-VAL-FW PIC X(4).
   02 GETSOCK-OPT-ERR PIC 9(8) COMP.
   02 GETSOCK-OPT-RET PIC 9(8) COMP.

GETSOCKOPT-ROUTINE.

*    TO QUERY THE SO_REUSEADDR OPTION FOR A SOCKET *
    MOVE 11 TO COMMANDA.
    MOVE X'0000FFFF' TO GETSOCK-OPT-LEVEL.
    MOVE 4 TO GETSOCK-OPT-NAME.
    MOVE 4 TO GETSOCK-OPT-LEN.

    CALL <EZACICAL> USING TOKEN COMMANDA SOCKETD
       GETSOCK-OPT-LEVEL
       GETSOCK-OPT-NAME
       GETSOCK-OPT-LEN
       GETSOCK-OPT-VAL-FW
       GETSOCK-OPT-ERR
       GETSOCK-OPT-RET.

GETSOCKOPT-ROUTINE-EXIT.
    EXIT.
```

**Input Parameters:**

- **COMMANDA**: Must be set to 11
- **SOCKETD**: The socket in question
- **GETSOCK-OPT-LEVEL**: Must be set to SOL_SOCKET (X’0000FFFF’)
- **GETSOCK-OPT-NAME**: The name of the option (see list)
- **GETSOCK-OPT-LEN**: Length of the data for the option

**Output Parameters:**

- **GETSOCK-OPT-VAL-FW**: Full-word (4 bytes) value. This is used for socket options with GETSOCK-OPT-VAL = 4
- **GETSOCK-OPT-VAL-DW**: Double-word (8 bytes) value. This is used for socket options with GETSOCK-OPT-VAL = 8 (SO_LINGER)
- **GETSOCK-OPT-ERR**: ERRNO field
- **GETSOCK-OPT-RET**: RETCODE field. Zero is successful; a negative value indicates an error.
B.1.13 GIVESOCKET

**Description:** The GIVESOCKET call tells TCP/IP to make the specified socket available to a TAKESOCKET call issued by another program. The Listener uses this call to pass sockets to child server programs. The concurrent server (for example, the Listener) should issue a SELECT call to wait for the child server program to issue the TAKESOCKET call. Once the concurrent server has been notified that the TAKESOCKET call has been issued, it can CLOSE the socket, freeing its resources and allowing it to be reused by the concurrent server. The concurrent server issues the GIVESOCKET call, specifying the client ID of the server program. It should then issue a GETCLIENTID call to get its own client ID information, which it should then pass to the child server program in its startup parameter list.

**Example:**

```cobol
01 GIVESOCKET-PARMS.
   02 GIVESOCK-SOCKET PIC 9(4) COMP.
   02 GIVESOCK-ID.
      03 GIVESOCK-DOMAIN PIC 9(8) COMP VALUE 2.
      03 GIVESOCK-NAME PIC X(8).
      03 GIVESOCK-TASK PIC X(8) VALUE SPACES.
      03 FILLER PIC X(20).
   02 GIVESOCK-ERR PIC 9(8) COMP.
   02 GIVESOCK-RET PIC 9(8) COMP.

GIVESOCKET-ROUTINE.
   *
   * TO PASS SOCKET 1 TO A STARTED TASK. THE PASS IS COMPLETED WHEN
   * THE STARTED TASK ISSUES A TAKESOCKET
   *
   MOVE 31 TO COMMANDA.
   MOVE 1 TO GIVESOCK-SOCKET.
   MOVE GETCLIENT-NAME TO GIVESOCK-NAME.
   CALL $EZACICAL USING TOKEN COMMANDA
      GIVESOCK-SOCKET
      GIVESOCK-ID
      GIVESOCK-ERR
      GIVESOCK-RET.

GIVESOCKET-ROUTINE-EXIT.
   EXIT.
```

**Input Parameters:**

- **COMMANDA**
  Must be set to 31

- **GIVESOCK-SOCKET**
  The socket to be given

- **GIVESOCK-DOMAIN**
  Addressing family. Must be set to 2 (AF_INET)

- **GIVESOCK-NAME**
  Address space ID of the region which is going to do the TAKESOCKET call. If we specify spaces here, the CICS sockets task that is going to issue the TAKESOCKET call is in the same CICS region as the task that issues the GIVESOCKET call.
GIVESOCK-TASK Specify blanks here

*Output Parameters:*

GIVESOCK-ERR ERRNO field

GIVESOCK-RET RETCODE field. Zero is successful; a negative value indicates an error.
B.1.14 INITAPI

**Description:** The INITAPI call performs IUCV connect and send-initial calls to establish a path to the TCPIP address space. It must be the first socket call issued by an application, unless the application is a child server, in which case the TAKESOCKET is the first call. On this call, we specify the maximum number of sockets we need in the calling application, and also a unique name for the associated MVS subtask. The INITAPI call returns the highest socket number available to the application. The recommended method for uniquely naming subtasks is to use the CICS tasks EIBTASKN for the first seven digits, suffixed by a single character.

**Example:**

```
01 INITAPI-PARMS.
  02 INITAPI-IDENT PIC X(8) VALUE "IUCVAPI ".
  02 INITAPI-MAXSOC PIC 9(4) COMP.
  02 INITAPI-MAXAPI PIC 9(4) COMP VALUE 2.
  02 INITAPI-SUBTSK.
    03 SUBTSK-NO PIC X(7).
    03 SUBTSK-IDENT PIC X VALUE "X".
  02 INITAPI-ZERO PIC 9(8) COMP.
  02 INITAPI-MAXSNO PIC 9(8) COMP.
  02 INITAPI-RET PIC 9(8) COMP.

INITAPI-ROUTINE.
  *
  * TO SET UP A PATH TO IUCV, REQUESTING 75 SOCKETS, AND NAMING THE
  * SUBTASK AS .......X
  *
  MOVE 0 TO COMMANDA.
  MOVE 75 TO INITAPI-MAXSOC.
  MOVE EIBTASKN TO SUBTSK-NO.

  CALL &EZACICAL USING TOKEN COMMANDA
    INITAPI-IDENT
    INITAPI-MAXSOC
    INITAPI-MAXAPI
    INITAPI-SUBTSK
    INITAPI-ZERO
    INITAPI-MAXSNO
    INITAPI-RET.

  INITAPI-ROUTINE-EXIT.
  EXIT.
```

**Input Parameters:**

- **COMMANDA** Must be set to 00
- **INITAPI-IDENT** Must be set to "IUCVAPI ." (Note the space)
- **INITAPI-MAXSOC** The maximum number of sockets this application will use. The maximum allowed is 5000; the minimum is 50.
INITAPI-MAXAPI Must be set to 2 (AF_INET)

INITAPI-SUBTSK Unique name for the associated subtask

*Output Parameters:*

INITAPI-MAXSNO The highest socket number available to this application. This is the ERRNO field if RETCODE is negative.

INITAPI-RET RETCODE. Zero is successful; a negative value indicates an error.
**B.1.15 IOCTL**

**Description:** The IOCTL call controls the operating characteristics of a socket. The operation to be controlled is specified by the IOCTL-CMD parameter. IOCTL is a very powerful call, allowing the application to add and delete routing table entries, and alter various other network parameters. The only option described here is given as a simple example. For more complete information on the IOCTL call, refer to *TCP/IP Version 2 Release 2.1 for MVS: Programmer’s Reference*.

One of the commands available to IOCTL is:

- **FIONBIO** (`X’8004A77E’`)—Sets or clears non-blocking input-output for a socket. If IOCTL-REQARG is zero, non-blocking input-output on the socket is cleared (the socket can block). If IOCTL-REQARG is non-zero, then non-blocking input-output is set for that socket.

The other commands are the following:

- **FIONREAD** gets the number of immediately readable bytes for the socket
- **SIOCADDRT** adds a routing table entry
- **SIOCATMARK** queries whether the current location in the data input is pointing to out-of-band data
- **SIOCDELRT** deletes a routing table entry
- **SIOCGIFADDR** gets the network interface address
- **SIOCGIFBRDADDR** gets the network interface broadcast address
- **SIOCGIFCONF** gets the network interface configuration
- **SIOCGIFDSTADDR** gets the network interface destination address
- **SIOCGIFFLAGS** gets the network interface flags
- **SIOCGIFMETRIC** gets the network interface routing metric
- **SIOCGIFNETMASK** gets the network interface network mask
- **SIOCSIFNETMASK** sets the network interface network mask.
Example:

```
01 IOCTL-PARMS.
   02 IOCTL-CMD PIC X(4).
   02 IOCTL-REQARG PIC X(4).
   02 IOCTL-RETLEN PIC 9(8) COMP.
   02 IOCTL-RETARG PIC X(4).
   02 IOCTL-ERR PIC 9(8) COMP.
   02 IOCTL-RET PIC 9(8) COMP.

IOCTL-ROUTINE.
  *
  * TO SET NONBLOCKING I/O FOR A SOCKET
  *
    MOVE 12 TO COMMANDA.
    MOVE X'8004A77E' TO IOCTL-CMD.
    MOVE 1 TO IOCTL-REQARG.

    CALL &EZACICAL: USING TOKEN COMMANDA SOCKETD
    IOCTL-CMD
    IOCTL-REQARG
    IOCTL-RETLEN
    IOCTL-RETARG
    IOCTL-ERR
    IOCTL-RET.

IOCTL-ROUTINE-EXIT.
  EXIT.
```

Input Parameters:

- **COMMANDA**
  - Must be set to 12
- **SOCKETD**
  - The socket in question
- **IOCTL-CMD**
  - The command indicating which operation to perform (see list)
- **IOCTL-REQARG**
  - The request argument. This is specific to the command being used

Output Parameters:

- **IOCTL-RETLEN**
  - The length of the return argument
- **IOCTL-RETARG**
  - The return argument
- **IOCTL-ERR**
  - ERRNO field
- **IOCTL-RET**
  - RETCODE. Zero is successful; a negative value indicates an error.
B.1.16 LISTEN

**Description:** The LISTEN call creates a queue for incoming connection requests. The length of this queue is specified in the call. Once this queue is full, all subsequent CONNECT calls from clients will be refused. The LISTEN call also indicates a readiness to accept client connection requests. The socket specified on the call becomes a passive socket, which means that it can no longer be used to initiate connection requests.

**Example:**

```
01 LISTEN-PARMS.
   02 LISTEN-ZERO PIC 9(8) COMP.
   02 LISTEN-BACKLOG PIC 9(8) COMP.
   02 LISTEN-ERR PIC 9(8) COMP.
   02 LISTEN-RET PIC 9(8) COMP.

LISTEN-ROUTINE.
   *
   TO INDICATE READINESS TO ACCEPT CONNECTION REQUESTS ON A SOCKET,
   SPECIFYING A BACKLOG QUEUE OF 5 REQUESTS
   *
   MOVE 13 TO COMMANDA.
   MOVE 5 TO LISTEN-BACKLOG.

   CALL &EZACICAL USING TOKEN COMMANDA SOCKETD
      LISTEN-ZERO
      LISTEN-BACKLOG
      LISTEN-ERR
      LISTEN-RET.

LISTEN-ROUTINE-EXIT.
   EXIT.
```

**Input Parameters:**

- **COMMANDA**
  Must be set to 13

- **SOCKETD**
  The socket to be listened on

- **LISTEN-BACKLOG**
  The maximum number of incoming CONNECT requests that will be queued. If it is less than zero, it is set to zero. If it is greater than SOMAXCONN, it is set to SOMAXCONN (SOMAXCONN is defined in tcpip.SEZACMAC(SOCKET))

**Output Parameters:**

- **LISTEN-ERR**
  ERRNO field

- **LISTEN-RET**
  RETCODE field. Zero is successful; a negative value indicates an error.
B.1.17 READ

**Description:** The READ call reads data on a socket and stores it in a specified buffer. If there are fewer bytes available than were requested, the READ call returns the number that were read. If no data is available on the socket, and the socket is in blocking mode, the READ call will block the caller until data arrives. If no data is available on the socket, and the socket is in non-blocking mode, the READ call returns -1 in RETCODE, and EWOULDBLOCK (35) in ERRNO.

**Example:**

```
01 READ-PARMS.
   02 READ-ZERO PIC X(8).
   02 READ-NBYTE PIC 9(8) COMP.
   02 READ-ZERO2 PIC X(16).
   02 READ-BUFF PIC X(100).
   02 READ-ERR PIC 9(8) COMP.
   02 READ-RET PIC 9(8) COMP.

READ-Routine.
*  
* TO READ UP TO 100 BYTES ON A SOCKET
*  
   MOVE 14 TO COMMANDA.
   MOVE 100 TO READ-NBYTE.

   CALL $EZACICAL USING TOKEN COMMANDA SOCKETD
      READ-ZERO
      READ-NBYTE
      READ-ZERO2
      READ-BUFF
      READ-ERR
      READ-RET.

READ-Routine-Exit.
EXIT.
```

**Input Parameters:**

- **COMMANDA**
  - Must be set 14
- **SOCKETD**
  - The socket to read the data on
- **READ-NBYTE**
  - Number of bytes to attempt to read on the socket

**Output Parameters:**

- **READ-BUFF**
  - The read-buffer. Must be at least READ-NBYTE in size
- **READ-ERR**
  - ERRNO field
- **READ-RET**
  - RETCODE field. If it is a positive number, it indicates how many bytes were read on the call; a negative value indicates an error.
B.1.18 RECVFROM

Description: The RECVFROM call receives data on a socket and stores it in a specified buffer. This call differs from the READ call in that it also returns the source address of the data.

If there are fewer bytes available than were requested, the RECVFROM call returns the number that were read. If no data is available on the socket, and the socket is in blocking mode, the RECVFROM call will block the caller until data arrives. If no data is available on the socket, and the socket is in non-blocking mode, the RECVFROM call returns -1 in RETCODE, and EWOULDBLOCK (35) in ERRNO.

The flags on the RECVFROM call can be used to specify that out-of-band (exception) data is received (MSG_OOB (1)), or that the receive does not destroy the data in the TCP/IP buffer, and it remains for a subsequent receive operation (MSG_PEEK (2)). Both options can be selected by setting RECVFROM-FLAGS to 3.

Example:

```
01 RECVFROM-PARMS.
  02 RECVFROM-ZERO PIC 9(8) COMP.
  02 RECVFROM-FLAGS PIC 9(8) COMP.
  02 RECVFROM-NBYTE PIC 9(8) COMP.
  02 RECVFROM-FROM.
    03 RECVFROM-AF-INET PIC 9(4) COMP.
    03 RECVFROM-PORT PIC 9(4) COMP.
    03 RECVFROM-IP-ADDR PIC 9(8) COMP.
    03 FILLER PIC X(8).
  02 RECVFROM-BUFF PIC X(100).
  02 RECVFROM-ERR PIC 9(8) COMP.
  02 RECVFROM-RET PIC 9(8) COMP.

RECVFROM-ROUTINE.
  * 
  * TO RECEIVE UP TO 100 BYTES ON A SOCKET 
  * 
    MOVE 16 TO COMMANDA.
    MOVE 0 TO FLAGS.
    MOVE 100 TO RECVFROM-NBYTE.

    CALL &EZACICAL USING TOKEN COMMANDA SOCKETID
       RECVFROM-ZERO
       RECVFROM-FLAGS
       RECVFROM-NBYTE
       RECVFROM-FROM
       RECVFROM-BUFF
       RECVFROM-ERR
       RECVFROM-RET.

RECVFROM-ROUTINE-EXIT.
  EXIT.
```

Input Parameters:
COMMANDA Must be set to 16
SOCKETD The socket to receive the data on
RECVFROM-FLAGS Either set it to zero, 1 (MSG_OOB), or 2 (MSG_PEEK), or 3 (MSG_OOB and MSG_PEEK)
RECVFROM-NBYTE Number of bytes to attempt to receive on the socket

Output Parameters:
RECVFROM-AF-INET Addressing family. Should return 2 (AF_INET)
RECVFROM-PORT Port number from where the data was received
RECVFROM-IP-ADDR Remote IP address of the source of the data
RECVFROM-BUFF Receive-buffer. Must be at least RECVFROM-NBYTE in size
RECVFROM-ERR ERRNO field
RECVFROM-RET RETCODE field. If it is a positive number, it indicates how many bytes were received on the call; a negative value indicates an error.
B.1.19 SELECT

**Description:** The SELECT call monitors activity on a set of sockets to see if any of the sockets are ready for reading, writing, or have an exception condition pending. You can use the SELECT call in two ways:

- SELECT immediately returns the current status of a set of sockets. The user must set the SELECT-TIMESW parameter to zero. This is the *immediate* mode.

- SELECT waits for either activity on a set of socket, or a timeout interval. The user can set SELECT-TIMESW to 1, and specify the required TIMEOUT interval. This ensures that the SELECT will always return after a given length of time, even if there is no socket activity. This is the *timeout* mode.

Use the immediate call if your application can usefully do other work even if there is no activity on the socket. Otherwise, use a timeout call, so that your application waits for a period of time, while there is no socket activity, before becoming active again.

You can use the SELECT call in a server program to provide a non-blocking ACCEPT function. For example, if an ACCEPT call is made and there are no CONNECT requests from clients, the ACCEPT call will block the calling application. To avoid being blocked use the SELECT call to check for activity on the bound socket before doing the ACCEPT. This will inform us of any CONNECT calls, when the socket is ready for reading.

We also use the SELECT call in concurrent servers before issuing a CLOSE for a socket that has been passed on a GIVESOCKET call. When the started server issues the TAKESOCKET, this generates an exception on the socket, which the SELECT call registers. The socket can then be safely closed.
Example:

```assembler
01 SELECT-PARMS.
   02 SELECT-LOM PIC 9(4) COMP.
   02 SELECT-NUMFDS PIC 9(8) COMP.
   02 SELECT-TIMESW PIC 9(8) COMP VALUE 1.
   02 SELECT-RDSW PIC 9(8) COMP VALUE 1.
   02 SELECT-WRSW PIC 9(8) COMP VALUE 0.
   02 SELECT-EXSW PIC 9(8) COMP VALUE 0.
   02 SELECT-TIMEOUT.
      03 SELECT-SECS PIC 9(8) COMP VALUE 30.
      03 SELECT-MILLSECS PIC 9(8) COMP VALUE 0.
   02 SELECT-RD-MASK PIC 9(16) COMP.
   02 SELECT-WR-MASK PIC 9(16) COMP.
   02 SELECT-EX-MASK PIC 9(16) COMP.
   02 SELECT-DZERO PIC 9(16) COMP.
   02 SELECT-RR-MASK PIC 9(16) COMP.
   02 SELECT-RW-MASK PIC 9(16) COMP.
   02 SELECT-RE-MASK PIC 9(16) COMP.
   02 SELECT-ERR PIC 9(8) COMP.
   02 SELECT-RET PIC 9(8) COMP.

SELECT-Routine.
*
* TO MONITOR THE READ-ACTIVITY ON THE SOCKETS SPECIFIED IN BIT-MASK
* (GENERATED BY EZACIC06), OR TIMEOUT AFTER 30 SECONDS
*
   MOVE BIT-MASK TO SELECT-RD-MASK.
   MOVE 8 TO SELECT-LOM.
   MOVE 51 TO SELECT-NUMFDS.
   MOVE 19 TO COMMANDA.

   CALL &EZACICAL& USING TOKEN COMMANDA
   SELECT-LOM
   SELECT-NUMFDS
   SELECT-TIMESW
   SELECT-RDSW
   SELECT-WRSW
   SELECT-EXSW
   SELECT-TIMEOUT
   SELECT-RD-MASK
   SELECT-WR-MASK
   SELECT-EX-MASK
   SELECT-DZERO
   SELECT-RR-MASK
   SELECT-RW-MASK
   SELECT-RE-MASK
   SELECT-ERR
   SELECT-RET.

SELECT-Routine-Exit.
EXIT.
```
**Input Parameters:**

**COMMANDA**
Must be set to 19

**SELECT-LOM**
Length of the bit-mask. This is calculated using integer arithmetic: \(((\text{SELECT-NUMFDS} + 31) \div 32) \times 4\). It determines the number of bytes required to represent the bit-mask.

**SELECT-NUMFDS**
The number of socket descriptors to be monitored by the SELECT call. It should be set the largest number of sockets being used by the application, plus 1.

**SELECT-TIMESW**
Time-switch. Set it to zero to specify no timeout value (a poll). Set it to 1 to specify a timeout. The length of the timeout is set by SELECT-TIMEOUT.

**SELECT-RDSW**
Read-switch. Set it to zero to prevent checking for read interrupts. Set it to 1 to check for read interrupts.

**SELECT-WRSW**
Write-switch. Set it to zero to prevent checking for write interrupts. Set it to 1 to check for write interrupts.

**SELECT-EXSW**
Exception-switch. Set it to zero to prevent checking for exception interrupts. Set it to 1 to check for exception interrupts.

**SELECT-SECS**
Seconds component of the timeout value

**SELECT-MILLSECS**
Milliseconds component of the timeout value (must be < 1000)

**SELECT-RD-MASK**
Bit-mask array for reads. A 1 in position n means check read interrupts on socket (n-1). A zero prevents a check on that socket. This is generated through EZACIC06. The size of this parameter is dependent on SELECT-LOM.

**SELECT-WR-MASK**
Bit-mask array for writes. A 1 in position n means check write interrupts on socket (n-1). A zero prevents a check on that socket. This is generated through EZACIC06. The size of this parameter is dependent on SELECT-LOM.

**SELECT-EX-MASK**
Bit-mask array for exceptions. A 1 in position n means check exception interrupts on socket (n-1). A zero prevents a check on that socket. This is generated through EZACIC06. The size of this parameter is dependent on SELECT-LOM.

**Output Parameters:**

**SELECT-RR-MASK**
Returned bit-mask array for reads. A 1 in position n means a read interrupt was found on socket (n-1). A zero indicates no activity on that socket. This is generated through EZACIC06. The size of this parameter is dependent on SELECT-LOM.

**SELECT-RW-MASK**
Returned bit-mask array for writes. A 1 in position n means write interrupt was found on socket (n-1). A zero indicates no activity on that socket. This is
generated through EZACIC06. The size of this parameter is dependent on SELECT-LOM.

**SELECT-RE-MASK**
Returned bit-mask array for exceptions. A 1 in position n means an exception interrupt was found on socket (n-1). A zero indicates no activity on that socket. This is generated through EZACIC06. The size of this parameter is dependent on SELECT-LOM.

**SELECT-ERR**
ERRNO field

**SELECT-RET**
RETCODE field. If it is a positive number, it indicates a successful call; a negative value indicates an error.

**Note:** Where the masks are discussed above, we refer to the character positions in the character masks, and not the bit positions in the bit-masks. COBOL arrays have 1 as the first index, whereas the bit-mask arrays expected by the SELECT call, start on index 0. Therefore, position n in the character array, is position n-1 in the bit-mask array. See the sample concurrent server for an example of how to use the SELECT call.
**B.1.20 SEND**

*Description:* The SEND call sends data on a socket. If there is not enough buffer space to hold the data to be transmitted, and the socket is in blocking mode, the call blocks the caller until extra space is available. If the socket is in non-blocking mode, the SEND call returns -1 in RETCODE, and EWOULDBLOCK (35) in ERRNO. The send flags can be set to zero, or MSG_OOB (1) to send out-of-band data.

*Example:*

```cobol
01 SEND-PARMS.
  02 SEND-NBYTE PIC 9(8) COMP.
  02 SEND-FLAGS PIC 9(8) COMP.
  02 SEND-DZERO PIC X(8).
  02 SEND-BUF PIC X(100).
  02 SEND-ERR PIC 9(8) COMP.
  02 SEND-RET PIC 9(8) COMP.

SEND-ROUTINE.
  *
  * TO SEND 50 BYTES ON A SOCKET
  *
  MOVE 20 TO COMMANDA.
  MOVE 50 TO SEND-NBYTE.
  MOVE 0 TO SEND-FLAGS.
  MOVE DATA-BUF TO SEND-BUF.

  CALL $EZACICAL$ USING TOKEN COMMANDA SOCKETD
  SEND-NBYTE
  SEND-FLAGS
  SEND-DZERO
  SEND-BUF
  SEND-ERR
  SEND-RET.

SEND-ROUTINE-EXIT.
  EXIT.
```

*Input Parameters:*

- **COMMANDA**
  Must be set to 20

- **SOCKETD**
  The socket to send the data out on

- **SEND-NBYTE**
  The number of bytes to attempt to send

- **SEND-FLAGS**
  Either set to zero, or 1 (MSG_OOB)

- **SEND-BUF**
  The send-buffer. This must be at least SEND-NBYTE in size.

*Output Parameters:*

- **SEND-ERR**
  ERRNO field
SEND-RET: The RETCODE field. If it is a positive number, it indicates how many bytes were sent on the call; a negative value indicates an error.
B.1.21 SENDTO

Description: This call is identical to the SEND call. The SENDTO-NAME structure is ignored in this implementation, since we only support stream sockets and not datagram sockets.

Example:

```
01 SENDTO-PARMS.
  02 SENDTO-NBYTE PIC 9(8) COMP.
  02 SENDTO-FLAGS PIC 9(8) COMP.
  02 SENDTO-NAME.
    03 SENDTO-FAMILY PIC 9(4) COMP.
    03 SENDTO-PORT PIC 9(4) COMP.
    03 SENDTO-ADDRESS PIC 9(8) COMP.
    03 SENDTO-DZERO PIC X(8).
  02 SENDTO-BUF PIC X(100).
  02 SENDTO-ERR PIC 9(8) COMP.
  02 SENDTO-RET PIC 9(8) COMP.

SENDTO-ROUTINE.
  *
  * TO SEND 50 BYTES ON A SOCKET
  *
    MOVE 22 TO COMMANDA.
    MOVE 50 TO SENDTO-NBYTE.
    MOVE 0 TO SENDTO-FLAGS.
    MOVE DATA-BUF TO SENDTO-BUF.

    CALL $EZACICAL USING TOKEN COMMANDA SOCKETD
    SENDTO-NBYTE
    SENDTO-FLAGS
    SENDTO-NAME
    SENDTO-BUF
    SENDTO-ERR
    SENDTO-RET.

SENDTO-ROUTINE-EXIT.
  EXIT.
```

Input Parameters:

- **COMMANDA**: Must be set to 22
- **SOCKETD**: The socket to send the data out on
- **SENDTO-NBYTE**: The number of bytes to attempt to send
- **SENDTO-FLAGS**: Either set to zero, or 1 (MSG_OOB)
- **SENDTO-BUF**: The send-buffer. This must be at least SENDTO-NBYTE in size

Output Parameters:

- **SENDTO-ERR**: ERRNO field
| SENDTO-RET | RETCODE field. If it is a positive number, it indicates how many bytes were sent on the call; a negative value indicates an error. |
B.1.22 SETSOCKOPT

Description: The SETSOCKOPT call sets or clears options associated with a socket. All options must be manipulated at the socket level, therefore the SETSOCK-OPT-LEVEL parameter must be set to SOL_SOCKET (X'0000FFFF').

The options available for SETSOCKOPT are the following:

- **SO_LINGER (X'00000080') (OPT-LEN = 8)**—Lingers on the CLOSE call if there is data present. If this option is set, then the CLOSE call will block until the data is sent. The data required is two full-words:
  - **ONOFF**—Indicates whether or not the option is set
  - **LINGER**—The time interval for the linger if the option is set

- **SO_OOBINLINE (X'00000100') (OPT-LEN = 4)**—Indicates whether or not out-of-band data is placed in the normal data stream as it is sent or received on the specified socket

- **SO_REUSEADDR (X'00000004') (OPT-LEN = 4)**—Indicates whether or not an address can be reused. When this option is set, the socket can be bound to a local port that is already in use, or still in a TIME-WAIT state after a CLOSE call

All socket options except SO_LINGER require a full-word value. If this value is non-zero it sets the option. If it is zero, then the option is cleared. SO_LINGER requires an indicator (first full-word) and a time-interval (second full-word).
Example:

```
01 SETSOCK-OPT-PARMS.
   02 SETSOCK-OPT-LEN PIC 9(8) COMP.
   02 SETSOCK-OPT-LEVEL PIC X(4).
   02 SETSOCK-OPT-NAME PIC 9(8) COMP.
   02 SETSOCK-OPT-VAL-DW.
      03 SETSOCK-OPT-VAL-1 PIC X(4).
      03 SETSOCK-OPT-VAL-FW PIC X(4).
   02 SETSOCK-OPT-ERR PIC 9(8) COMP.
   02 SETSOCK-OPT-RET PIC 9(8) COMP.

SETSOCK-OPT-ROUTINE.
* TO SET THE SO_REUSEADDR OPTION FOR A SOCKET *
   MOVE 23 TO COMMANDA.
   MOVE 4 TO SETSOCK-OPT-LEN.
   MOVE X'0000FFFF' TO SETSOCK-OPT-LEVEL.
   MOVE 4 TO SETSOCK-OPT-NAME.
   MOVE 1 TO SETSOCK-OPT-VAL.

   CALL 'EZACICAL' USING TOKEN COMMANDA SOCKETD
   SETSOCK-OPT-LEN
   SETSOCK-OPT-LEVEL
   SETSOCK-OPT-NAME
   SETSOCK-OPT-VAL-FW
   SETSOCK-OPT-ERR
   SETSOCK-OPT-RET.

SETSOCK-OPT-ROUTINE-EXIT.
   EXIT.
```

**Input Parameters:**

- **COMMANDA**: Must be set to 23
- **SOCKETD**: The socket in question
- **SETSOCK-OPT-LEN**: Length of the data for the option
- **SETSOCK-OPT-LEVEL**: This must be set to SOL_SOCKET (X'0000FFFF')
- **SETSOCK-OPT-NAME**: The name of the option (see list)
- **SETSOCK-OPT-VAL-FW**: Full-word (4 bytes) value. This is used for socket options with OPT-VAL = 4
- **SETSOCK-OPT-VAL-DW**: Double-word (8 bytes) value. This is used for socket options with OPT-VAL = 8 (SO_LINGER)

**Output Parameters:**

- **SETSOCK-OPT-ERR**: ERRNO field
- **SETSOCK-OPT-RET**: RETCODE field. Zero is successful; a negative value indicates an error.
B.1.23 SHUTDOWN

Description: The SHUTDOWN call shuts down all or part of a duplex connection. The SHUTDOWN-HOW parameter sets the condition for shutting down the socket:

- 0 - end communication from the socket
- 1 - end communication to the socket
- 2 - end all communication both to and from the socket.

Example:

```
01 SHUTDOWN-PARMS.
   02 SHUTDOWN-FZERO PIC 9(8) COMP.
   02 SHUTDOWN-HOW PIC 9(8) COMP.
   02 SHUTDOWN-ERR PIC 9(8) COMP.
   02 SHUTDOWN-RET PIC 9(8) COMP.

SHUTDOWN-Routine.
* * TO SHUTDOWN COMMUNICATION IN BOTH DIRECTIONS ON A SOCKET *
*                             *
   MOVE 24 TO COMMANDA.
   MOVE 2 TO SHUTDOWN-HOW.

   CALL &EZACICAL USING TOKEN COMMANDA SOCKETD
      SHUTDOWN-FZERO
      SHUTDOWN-HOW
      SHUTDOWN-ERR
      SHUTDOWN-RET.

SHUTDOWN-Routine-Exit.
   EXIT.
```

Input Parameters:
- COMMANDA Must be set to 24
- SOCKETD The socket to shutdown
- SHUTDOWN-HOW Condition for shutdown - 0 (inbound), 1 (outbound) or 2 (both)

Output Parameters:
- SHUTDOWN-ERR ERRNO field
- SHUTDOWN-RET RETCODE field. A positive number indicates successful call; a negative value indicates an error.
B.1.24 SOCKET

**Description:** The SOCKET call creates a socket and allocates your chosen descriptor to it. The only kind of socket supported is of type SOCK_STREAM (1) in domain AF_INET (2). The PROTOCOL parameter should be set to zero, which allows TCP/IP to set the default protocol for the specified domain and type.

**Example:**

```plaintext
01 SOCKET-PARMS.
  02 SOCKET-HZERO PIC 9(4) COMP.
  02 SOCKET-AF PIC 9(8) COMP VALUE 2.
  02 SOCKET-TYPE PIC 9(8) COMP VALUE 1.
  02 SOCKET-PROTOCOL PIC 9(8) COMP VALUE 0.
  02 SOCKET-SOCKNO PIC 9(8) COMP.
  02 SOCKET-ERR PIC 9(8) COMP.
  02 SOCKET-RET PIC 9(8) COMP.

SOCKET-Routine.
* 
* CREATE A NEW SOCKET, AND GIVE IT THE NUMBER ◆1◆ 
* 
  MOVE 25 TO COMMANDA.
  MOVE 1 TO SOCKET-SOCKNO.

  CALL ◆EZACICAL◆ USING TOKEN COMMANDA
  SOCKET-HZERO
  SOCKET-AF
  SOCKET-TYPE
  SOCKET-PROTOCOL
  SOCKET-SOCKNO
  SOCKET-ERR
  SOCKET-RET.

SOCKET-Routine-Exit.
EXIT.
```

**Input Parameters:**
- **COMMANDA**
  Must be set to 25
- **SOCKET-AF**
  Addressing family. Must be set to 2 (AF_INET)
- **SOCKET-TYPE**
  Must be set to 1 (STREAM)
- **SOCKET-PROTOCOL**
  Protocol for the socket. Should be set to zero to allow TCP/IP to default the protocol for the socket type.
- **SOCKET-SOCKNO**
  The number for the new socket

**Output Parameters:**
- **SOCKET-ERR**
  ERRNO field
- **SOCKET-RET**
  RETCODE field. A positive number indicates successful call, and gives new socket number; a negative value indicates an error.
B.1.25 TAKESOCKET

**Description:** The TAKESOCKET call acquires a socket from another task. The task that has issued the GIVESOCKET call should pass its client ID details and socket descriptor number to the task that is to issue the TAKESOCKET.

**Example:**

```
01 TAKE-SOCKET-PARMS.
   02 TAKE-SOCKET-HZERO PIC 9(4) COMP.
   02 TAKE-SOCKET-CLIENTID.
      03 TAKE-SOCKET-DOMAIN PIC 9(8) COMP.
      03 TAKE-SOCKET-NAME PIC X(8).
      03 TAKE-SOCKET-TASK PIC X(8).
      03 TAKE-SOCKET-RESV PIC X(20).
   02 TAKE-SOCKET-LDESC PIC 9(8) COMP.
   02 TAKE-SOCKET-SOCKNO PIC 9(8) COMP.
   02 TAKE-SOCKET-ERR PIC 9(8) COMP.
   02 TAKE-SOCKET-RET PIC 9(8) COMP.

TAKESOCKET-Routine.

* TO TAKE A SOCKET WHICH WAS PASSED BY A GIVESOCKET CALL. WE
  IDENTIFY WHICH SOCKET TO TAKE BY INCLUDE INFORMATION ABOUT THE
  TASK WHICH ISSUED THE GIVESOCKET

MOVE 32 TO COMMANDA.
MOVE GIVE-CLIENTID TO TAKE-SOCKET-CLIENTID.

CALL $EZACICAL USING TOKEN COMMANDA
   TAKE-SOCKET-HZERO
   TAKE-SOCKET-CLIENTID
   TAKE-SOCKET-LDESC
   TAKE-SOCKET-SOCKNO
   TAKE-SOCKET-ERR
   TAKE-SOCKET-RET.

TAKESOCKET-Routine-Exit.
EXIT.
```

**Input Parameters:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMANDA</td>
<td>Must be set to 32</td>
</tr>
<tr>
<td>TAKE-SOCKET-DOMAIN</td>
<td>Addressing family. Must be set to 2 (AF_INET)</td>
</tr>
<tr>
<td>TAKE-SOCKET-NAME</td>
<td>Address space ID of the task that has issued the GIVESOCKET</td>
</tr>
<tr>
<td>TAKE-SOCKET-TASK</td>
<td>Subtask ID of the task that issues the GIVESOCKET</td>
</tr>
<tr>
<td>TAKE-SOCKET-LDESC</td>
<td>Number of the socket passed by the task that issued the GIVESOCKET</td>
</tr>
<tr>
<td>TAKE-SOCKET-sockno</td>
<td>Number for the new socket</td>
</tr>
</tbody>
</table>
Output Parameters:

TAKE-SOCKET-ERR    ERRNO field
TAKE-SOCKET-RET    RETCODE field. If it is positive, it is the new socket number; a negative value indicates an error.
B.1.26 WRITE

Description: The WRITE call writes data on a socket. If there is not enough buffer space to hold the data to be transmitted, and the socket is in blocking mode, the WRITE call blocks the caller until extra space is available. If the socket is in non-blocking mode, the call returns a -1 in RETCODE, and EWOULDBLOCK (35) in ERRNO.

Example:

```
01 WRITE-PARMS.
  02 WRITE-NBYTE PIC 9(8) COMP.
  02 WRITE-FZERO PIC 9(8) COMP.
  02 WRITE-SZERO PIC X(16).
  02 WRITE-BUF PIC X(100).
  02 WRITE-ERR PIC 9(8) COMP.
  02 WRITE-RET PIC 9(8) COMP.

WRITE-Routine.
  *
  * TO WRITE 50 BYTES ON A SOCKET
  *
      MOVE 26 TO COMMANDA.
      MOVE 50 TO WRITE-NBYTE.
      MOVE DATA-BUF TO WRITE-BUF.

      CALL &EZACICAL USING TOKEN COMMANDA SOCKETD
         WRITE-NBYTE
         WRITE-FZERO
         WRITE-SZERO
         WRITE-BUF
         WRITE-ERR
         WRITE-RET.

WRITE-Routine-Exit.
   EXIT.
```

Input Parameters:

- **COMMANDA**: Must be set to 26
- **SOCKETD**: The socket to write the data to
- **WRITE-NBYTE**: The number of bytes to attempt to write
- **WRITE-BUF**: The write-buffer. This must be at least WRITE-NBYTE in size

Output Parameters:

- **WRITE-ERR**: ERRNO field
- **WRITE-RET**: RETCODE field. If it is positive, it indicates how many bytes were sent on the call; a negative value indicates an error.
B.1.27 EZACIC04

Description: EZACIC04 converts EBCDIC data to ASCII data. It should be used before issuing a SEND or WRITE call from a CICS sockets transaction, to ensure that data being sent out onto the network is in ASCII format.

Example:

```
01 ETOA-TOKEN PIC X(16) VALUE 'TCPIPTOASCIIXLAT'.
01 ETOA-LEN PIC 9(8) COMP.
    77 DATA-BUF PIC X(100).
ETOA-Routine.
*
* TO CONVERT 50 BYTES OF EBCDIC SEND DATA TO ASCII
*
    MOVE 50 TO ETOA-LEN.
    CALL 'EZACIC04' USING ETOA-TOKEN
      DATA-BUF
      ETOA-LEN.
ETOA-Routine-Exit.
EXIT.
```

Parameters:

ETOA-TOKEN
Token field required by EZACIC04

DATA-BUF
A data buffer. Before the call to EZACIC04, this buffer contains the EBCDIC data. After the call, it contains the ASCII data

ETOA-LEN
The number of bytes to translate.
B.1.28 EZACIC05

Description: EZACIC05 converts ASCII data to EBCDIC data. It should be used after issuing a RECVFROM or READ call from a CICS sockets transaction, to ensure that data being read from the network converted into EBCDIC for CICS.

Example:

```
01 ATOE-TOKEN PIC X(16) VALUE 'TCPIPTOEBCDICTLIT'.
01 ATOE-LEN PIC 9(8) COMP.
77 DATA-BUF PIC X(100).

ATOE-ROUTINE.
*  *   TO CONVERT 50 BYTES OF ASCII RECEIVED DATA TO EBCDIC
*  *
   MOVE 50 TO ATOE-LEN.

   CALL 'EZACIC05' USING ATOE-TOKEN
      DATA-BUF
      ATOE-LEN.

ATOE-ROUTINE-EXIT.
EXIT.
```

Parameters:

- **ATOE-TOKEN**: Token field required by EZACIC05
- **DATA-BUF**: A data buffer. Before the call to EZACIC05, this buffer contains the ASCII data. After the call, it contains the EBCDIC data
- **ATOE-LEN**: The number of bytes to translate.
B.1.29 EZACIC06

**Description:** EZACIC06 is used in conjunction with the SELECT call in a COBOL program. It converts a COBOL array of character variables into a bit-mask array used by the SELECT call. It also converts the bit-mask array back into a COBOL character array.

**Example:** We need the following definitions for EZACIC06:

```cobol
* THESE DEFINITIONS ARE BASED ON SELECT CALL MONITORING 50 SOCKETS *
01 CHAR-MASK.
  02 CHAR-STRING PIC X(50).

01 CHAR-ARRAY REDEFINES CHAR-MASK
  02 CHAR-ENTRY-TABLE OCCURS 50 TIMES.
    03 CHAR-ENTRY PIC X.

01 BIT-MASK.
  02 BIT-ARRAY PIC 9(16) COMP.

01 BIT-FUNCTION-CODES.
  02 CTOB PIC X(4) VALUE 'CTOB'.
  02 BTOC PIC X(4) VALUE 'BTOC'.
  02 CTOB-RET PIC 9(8) COMP.
  02 BTOC-RET PIC 9(8) COMP.

01 BIT-MASK-LENGTH PIC 9(8) COMP VALUE 50.

01 BITMASK-TOKEN PIC X(16) VALUE 'TCPBITMASKCOBL'.
```
To generate a bit-mask for a SELECT call where we want to monitor sockets 0 through 3 for read and exception activity, we perform the following steps:

```
* CHAR-ENTRY(1) REPRESENTS SOCKET 0, BECAUSE COBOL ARRAYS START
* WITH INDEX=1
*
MOVE ZEROS TO CHAR-ENTRY-TABLE
MOVE $1 to CHAR-ENTRY(1)
CHAR-ENTRY(2)
CHAR-ENTRY(3)
CHAR-ENTRY(4).

CALL &EZACICO6 USING BITMASK-TOKEN
CTOB
BIT-MASK
CHAR-MASK
BIT-MASK-LENGTH
CTOB-RET.

* WE THEN MOVE THE BIT-MASK TO REQUIRED SELECT FIELD
*
MOVE BIT-MASK TO SELECT-RD-MASK
MOVE BIT-MASK TO SELECT-EX-MASK
```

To generate a COBOL character array after the SELECT call, in order to check the bit-mask for socket activity, we perform the following steps:

```
* * WE CONVERT THE BIT-MASK TO A COBOL CHARACTER ARRAY *
*
MOVE SELECT-RR-MASK TO BIT-MASK.

CALL &EZACICO6 USING BITMASK-TOKEN
BTOC
BIT-MASK
CHAR-MASK
BIT-MASK-LENGTH
CTOB-RET.

* * NOW WE CAN CHECK THE CHARACTER ARRAY *
```
**Parameters:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITMASK-TOKEN</td>
<td>Token field required by EZACIC06</td>
</tr>
<tr>
<td>CTOB</td>
<td>An indicator to EZACIC06, for performing a conversion of character array to bit-mask array</td>
</tr>
<tr>
<td>BTOC</td>
<td>An indicator to EZACIC06, for performing a conversion of bit-mask array to character array</td>
</tr>
<tr>
<td>BIT-MASK</td>
<td>A bit-mask array used by the SELECT call. This is either generated by EZACIC06, or used as input to EZACIC06, depending on the conversion being requested.</td>
</tr>
<tr>
<td>CHAR-MASK</td>
<td>A COBOL character array used by the application to check socket activity monitored by the SELECT call. This is either generated by EZACIC06, or used as input to EZACIC06, depending on the conversion being requested.</td>
</tr>
<tr>
<td>BIT-MASK-LENGTH</td>
<td>The number of entries in the character array to be used when generating the bit-mask. It is also the number of bits to be used in the bit-mask when generating the character array.</td>
</tr>
<tr>
<td>CTOB-RET</td>
<td>RETCODE field</td>
</tr>
<tr>
<td>BTOC-RET</td>
<td>RETCODE field</td>
</tr>
</tbody>
</table>

CICS/ESA and TCP/IP for MVS
Index

A
abends
  AETA  38
  AEY9  39
  BIND  30, 35, 105
  IAPI  36, 105
addressing family  143
  AF-INET  143

B
BEGINVTAM statement  31

C
C/370 language programs
  compiling  27
  CEDF  114
child server  44
COBOL programs
  compiling  23
  errors in compiling  25
  compiling a C/370 program  27
  compiling a COBOL program  23
  concurrent server  44
  CSKD  15, 17
  CSKE  15, 17
  CSKL  15

D
DATA data set (of TCP/IP)  31
debugging CICS programs
  using a dump  117
  using auxiliary trace  107
  using CEDF  114
  using GTF trace  111
  DFHEITVL  23
  DFHTC2500  35
  DFHTC2507  35
disabling the interface  36
diskette
  contents  126
  MESSAGE directory  126
  MESSAGE\WORKSTN directory  126
  moving contents to AIX  128
  moving contents to MVS  127
  moving contents to OS/2  127
  RETRIEVE directory  126

E
enabling the Sockets Interface
  automatically  34
  (continued)
  manually  33
  EZACIC00  10, 17
  defining to CICS  18
  EZACIC01  10, 17
  defining to CICS  19
  EZACIC02  10, 17
  defining to CICS  19
  EZACIC03  10, 18
  EZACIC04  10, 18
  EZACIC05  10, 18
  EZACIC06  10, 18
  EZACIC07  10, 18
  EZACICAL  10, 18
  EZACICM  17
  defining to CICS  21
  EZACICSC  18
  EZACICSE  95, 98
  EZACICSS  18

I
IBM-supplied modules
  EZACIC00  10, 17
  EZACIC01  10, 17
  EZACIC02  10, 17
  EZACIC03  10, 18
  EZACIC04  10, 18
  EZACIC05  10, 18
  EZACIC06  10, 18
  EZACIC07  10, 18
  EZACICAL  10, 18
  EZACICM  17
  EZACICSC  18
  EZACICSS  18
  IUCVMULT  18
installing the sample programs  126
iterative server  43
  IUCVMULT  18

J
JCL
  for compiling a C/370 program  28
  for compiling a COBOL program  23
  for starting CICS  14

L
Listener  10

M
messaging system
  client implementation  86
messaging system *(continued)*
function 51
installing the sample programs 126
overview 51
sample programs 123
sample server programs 59

P
PORT
 defining in PROFILE data set 30
 used when starting the Sockets Interface 34
profile data set (of TCPIP) 29
programs in the sample systems
 EZACICSE 100
 SISSRD1C 59, 124
 SISSRD2C 59, 124
 SISSRI1C 59, 124
 SISSRRI1C 59, 124
 SISSRW1C 59, 124
 SISSUCSCVC 67
 SISSIPRVC 76
PTFs to TCP/IP for MVS
 UN40677 13, 97
 UN42952 13
 UN43356 13
PTFs, effect on function 97

R
record retrieval system
 installing the sample programs 126
 overview 57
 sample programs 125
reserving a PORT 30

S
security
 checking incoming requests 95
 EZACICSE 95, 98
 implementing 95
 sample implementation 100
servers
 child 44
 concurrent 44
 iterative 43
 SEZACMAC 27
 SEZACMTX 27
 SEZALINK 14
 SEZATCP 14, 17, 26, 27
Sockets Interface
 automatic enable 34
 disabling 36
 enabling 33
 immediate disable 38
 manual enable 33
 problems enabling 35
 quiescent disable 37

T
task interface element 10
tcpip - used in examples as high-level qualifier 13
TCPIP data sets
 DATA 31
 PROFILE 29
TCPIPUSERID
 defining in DATA data set 31
 used when starting the Sockets Interface 34
TCPM transient data queue 21
TIE 10
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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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. _____________________________