A Beginner's Guide to MVS TCP/IP Socket Programming

Author:
Version: 1.2
Document Number: GG24-2561-00
Build Date: 06/22/95 20:29:51
Copyright Date: Copyright IBM Corp. 199

Processed by boo2pdf (http://www.kev009.com/wp/boo2pdf)
A Beginner's Guide to MVS TCP/IP Socket Programming

This publication provides basic TCP/IP socket programming information to MVS program developers who plan to use the socket programming interfaces of IBM TCP/IP Version 3 Release 1 for MVS. The main focus is the Sockets Extended, REXX sockets, IMS sockets and CICS sockets programming interfaces of IBM TCP/IP Version 3 Release 1 for MVS. The reader is not required to be familiar with C programming syntax. Code samples are provided in COBOL, PL/I, assembler and REXX. No prerequisite socket programming knowledge is required, but the reader is supposed to be familiar with the MVS environment and its subsystems, including IMS and CICS, and the related application program development tools and techniques.

(355 pages)
A Beginner's Guide to MVS TCP/IP Socket Programming

3.1 TCP/IP Protocol Layers
3.2 Addresses
3.2.1 IP Addresses
3.2.2 Ports
3.3 Sockets
3.4 Socket Types
3.5 Encapsulation
3.6 Addressing Families
3.6.1 Integrated Sockets
3.7 General Socket Program Structure
3.7.1 Iterative Server
3.7.2 Concurrent Server
3.7.3 Socket Program Categories

4.0 Chapter 4. The IBM TCP/IP for MVS Socket APIs
4.1 API Relationship
4.2 IBM TCP/IP for MVS C-Sockets
4.3 Sockets Extended Call Interface
4.3.1 PL/I Programs
4.3.2 User Abend 4093
4.4 Sockets Extended Assembler Macro Interface
4.5 REXX Sockets
4.6 Pascal API
4.7 Inter-User Communication Vehicle (IUCV) Sockets

5.0 Chapter 5. Your First Socket Program
5.1 Type Conversion Between Programming Languages
5.2 Iterative Server Program Structure
5.3 Initialize the Socket API
5.3.1 Initializing a C-socket Program
5.3.2 Getclientid
5.4 Create a Socket
5.5 Bind a Socket to a Specific Port Number
5.6 Listen for Client Connection Requests
5.7 Accepting Connection Requests from Clients
5.8 Transferring Data Over a Stream Socket
5.8.1 Streams and Messages
5.8.2 Reading and Writing Data From and To a Socket
5.8.3 Data Representation
5.9 Closing a Connection
5.9.1 Half Close
5.9.2 The Linger Option
5.10 Blocking, Non-blocking and Asynchronous Socket Calls
5.11 Socket Programs and MVS Security
5.11.1 User or Client Authentication
5.11.2 Authorizing Access to MVS Resources

6.0 Chapter 6. Native MVS Concurrent Server Program
6.1 Concurrent Servers in the Native MVS Environment
6.2 MVS Subtasking Considerations
6.2.1 Access to Shared Storage Areas
6.2.2 Data Set Access
6.2.3 Task and Workload Management
6.2.4 Security Considerations
6.2.5 Reentrant Code
6.3 Program Structure
6.4 Initializing the Concurrent Server Program
6.5 Select Processing
6.6 Accepting Connection Requests from Clients
6.6.1 Give Socket to Subtask
6.6.2 Take Socket from Main Process

7.0 Chapter 7. Socket Client Programs
7.1 General REXX Subroutine for Socket Calls
7.2 Initializing the Socket API
7.2.1 Getclientid
Connecting a Client to a Server

Accessing a Host Entry Structure with EZACIC08

Closing the Socket

Terminating the REXX Socket API

Chapter 8. Datagram Socket Programs

Datagram Socket Characteristics

Datagram Socket Program Structure

Use of Connect on a Datagram Socket

Transferring Data Over a Datagram Socket

Chapter 9. IMS Sockets

IMS and TCP/IP Networks

Overview of IMS Sockets

Concurrent Server in an IMS Environment

IMS Listener Security Exit

Remote Client Design Considerations

Explicit-mode Server Program

Implicit-mode Server Program

Dual-purpose IMS Programs

IMS Recovery Considerations

Chapter 10. CICS Sockets

CICS and TCP/IP Networks

Overview of CICS Sockets

Concurrent Server in a CICS Environment

Link Editing CICS Socket Programs

Chapter 11. Debugging and Tracing Socket Programs

Exception Handling

Application Trace Facilities

TCP/IP Packet Trace

IUCV Socket API Trace Function

Appendix A. Sample Datagram Socket Programs

Datagram Socket COBOL Server Program

Datagram Socket COBOL Client Program

Datagram Socket C Server Program

Datagram Socket C Client Program

Appendix B. Sample Stream Socket Programs

Sample Stream Socket COBOL Server

Sample Stream Socket COBOL Client

Sample Stream Socket C Server

Sample Stream Socket C Client

Appendix C. Sample IMS Socket Programs

Dual Purpose Implicit Mode IMS Server Program

C Client Program to Test Dual Purpose IMS Server

Explicit Mode IMS Server Program

IMS Listener Security Exit

Appendix D. Sample CICS Socket Program

Stream Socket COBOL Program for CICS

C Version of EZACICSC

Appendix E. Sample REXX Socket Programs

REXX Client

REXX Server

NetView NETSTAT Client REXX

NETSTAT Server REXX

Appendix F. Sample PLI Socket Programs

PL/I Server

Appendix G. Socket Utilities for Sockets Extended Programs

TPICLNID Obtain Values for TCP/IP Client ID

TPINTOA Convert IP Address to Character String

TPIIADDR Convert IP Address Character String to Full-word

TIPIOCTL Convert IOCTL Command Name to Command

TPIWAIT Place Calling Process in Wait

TPIRACF Interface to RACROUTE REQUEST=VERIFY User SVC
A Beginner's Guide to MVS TCP/IP Socket Programming

G.7 User SVC for RACROUTE REQUEST=VERIFY
G.8 TPIAUTH Issue RACROUTE REQUEST=AUTH for FACILITY Class
H.0 Appendix H. Sample MVS Concurrent Server
H.1 TPI Concurrent MVS Server
H.1.1 TPIMAIN Concurrent Server Main Process
H.1.2 TPILOGWT Logwriter Data Services Task
H.1.3 TPISERV Concurrent Server Subtask
H.1.4 TPISERVD Concurrent Server DB2 Access
H.1.5 TPISEND Send Data Over a Stream Socket
H.1.6 TPIRECV Receive Data Over a Stream Socket
H.1.7 TPIMCB Macro Main Task Control Block
H.1.8 TPISCB Macro Subtask Control Block
H.1.9 TPILOG Macro Issue Logwriter Request
H.1.10 TPIITRC Macro Issue Trace Request
H.1.11 TPIIMASK Macro Set and Test Bits in Select Mask
H.1.12 TPIREC Macro DB2 Row Layout
H.1.13 TPIMSO Macro Socket Descriptor Table
H.2 TPI REXX Client Application
H.2.1 TPI REXX Client
H.2.2 TPI REXX Client ISPF Panel Definition
H.2.3 TPI REXX Client ISPF Message Definitions
H.3 TPI DB2 Table Definition
H.4 Sample Log from TPI Server Execution
I.0 Appendix I. Sample Compile and Link JCL Procedures
I.1 Assemble JCL Procedure
I.2 COBOL Compile JCL Procedure
I.3 C/370 Compile JCL Procedure
I.4 Link/Edit JCL Procedure
ABBREVIATIONS List of Abbreviations
INDEX Index
COMMENTS ITSO Technical Bulletin Evaluation

FIGURES

1. Sockets in TCP/IP 1.1
2. TCP/IP Layers 1.1
3. The Application Model - Where Do We Split the Application? 1.2.1
4. Peer-to-Peer Distribution Model 1.2.2
5. Client/Server Distribution Model 1.2.2
6. Conversational Communications Model 1.2.3
7. Remote Procedure Call Communications Model 1.2.3
8. Message Queuing Communications Model 1.2.3
9. Socket Programming Interface 2.2
10. RPC Programming Interface and Protocol Layers 2.3
11. X-Windows Client and Server Hosts 2.4
12. X/Open Transport Layer Programming Interface 2.5
13. The TCP/IP Protocol Stack 3.1
14. IP Address Classes 3.2.1
15. Multihomed IP Host 3.2.1
16. The Port Concept 3.2.2
17. Port Number Assignment 3.2.2
18. The Socket Concept 3.3
19. Socket Calls for a Connection Oriented Protocol 3.4
20. Socket Calls for a Connectionless-Oriented Protocol 3.4
21. TCP/IP Encapsulation 3.5
22. Integrated Sockets in OpenEdition/MVS 3.6.1
24. Iterative Server Main Logic 3.7.1
25. Concurrent Server Main Logic 3.7.2
26. Socket API Relationship to TCP/IP Protocol Layers 4.1
27. Sockets Extended Macro Interface Storage Areas 4.4
28. Iterative Server Main Logic 5.2
A Beginner’s Guide to MVS TCP/IP Socket Programming

29. Identifying Your TCP/IP Address Space via TCPNAME 5.3
30. Identifying Your Own Program with a Client ID 5.3
31. The Client ID Structure 5.3
32. MVS TCP/IP Socket Descriptor Table 5.4
33. The TCP Buffer Flush Technique 5.8.1
34. Big or Little Endian Byte Order for a 2-Byte Integer 5.8.3
35. Closing Sockets 5.9
36. Serialize Access to a Shared Storage Area 6.2.1
37. Synchronize Use of a Common Service Task 6.2.1
38. Concurrent Server in an MVS Address Space 6.3
39. Host Entry Structure 7.1.1
40. Datagram Server Program Structure 8.2
41. IMS Sockets Structural Overview 9.2
42. Explicit-mode Server Program Initiation 9.3.3
43. Implicit-mode Server Program Initiation 9.3.4
44. IMS Assist Module Process Flow 9.3.4
45. Dual-purpose IMS Program Input/Output Flow 9.4
46. CICS Socket Application Overview 10.2
47. CICS Sockets Infrastructure 10.2
48. Concurrent Server in CICS 10.3
49. CICS Listener Transaction Request Message Format 10.3
50. Sample Packet Trace Output 11.3
51. Packet Trace of TCP Connection: SYN Segment 11.3
52. Packet Trace of TCP Connection: SYN + ACK Segment 11.3
53. Packet Trace of TCP Connection: ACK Segment 11.3
54. Socket API Trace: INITAPI Call 11.4
55. Socket API Trace: GETCLIENTID Call 11.4
56. Socket API Trace: SOCKET Call 11.4
57. Socket API Trace: BIND Call 11.4
58. Socket API Trace: LISTEN Call 11.4
59. Socket API Trace: ACCEPT Call 11.4
60. Socket API Trace: RECEIVE Peek Call 11.4
61. Socket API Trace: RECEIVE Call 11.4
62. TPI Application Components H.0
63. TPI Server Address Space Logic H.1

TABLES

1. IBM TCP/IP Version 3 Release 1 for MVS Socket Libraries and MVS Environments 2.2
2. Which Socket Type to Use 3.4
3. Network Interface and Typical MTU Values 3.5
4. Addressing Families and Programming Interfaces 3.6
5. Functional Comparison of the TCP/IP Socket APIs 4.1
6. Language Type Definition Conversion 5.1
7. Effect of Shutdown Socket Call 5.9.1
8. Effect of Blocking or Non-blocking Mode 5.10
9. Important IUCV Socket Trace Entry Fields 11.4

FRONT_1

Special Notices

This publication is intended to help application developers to develop MVS TCP/IP socket based client/server programs. The information in this publication is not intended as the specification of any programming interfaces that are provided by IBM TCP/IP Version 3 Release 1 for MVS. See the PUBLICATIONS section of the IBM Programming Announcement for IBM TCP/IP Version 3 Release 1 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 book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

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

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

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

ACF/VTAM
Advanced Peer-to-Peer Networking
AIX/6000
Application Development
AS/400
C Set ++
C/370
CICS OS/2
CICS/MVS
COBOL/2
CUA
DatagLANce
DFSMS
Distributed Relational Database Architecture
Enterprise Systems Architecture/370
Enterprise System/3090
Enterprise Systems Connection Architecture
ES/9000
ESA/390
IBM
IMS CS/2
MQSeries
OpenEdition
OS/2
PS/2
RACF
S/390
SMP/E
System/390
VTAM

AD/Cycle
AIX
AnyNet
APPN
BookManager
C/2
CICS
CICS/ESA
CICS/6000
COBOL/370
DATABASE 2
DB2
DFSMS/MVS
DRDA
Enterprise Systems Architecture/390
Enterprise System/9000
ES/390
ESA/370
ESCON
IMS Client Server/2
IMS/ESA
MVS/ESA
Operating System/2
Presentation Manager
QMF
S/370
SAA
System/370
Systems Application Architecture
3090
The following terms are trademarks of other companies:

Windows is a trademark of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

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

C-bus is a trademark of Corollary, Inc.

Other trademarks are trademarks of their respective companies.

**PREFACE Preface**

This book is for newcomers in the world of IBM TCP/IP for MVS socket programming.

We do not write for the experts who have written complicated C-socket code for the last 20 years or so. On the contrary, our focus is on those of you who have written MVS application programs for CICS, IMS or batch during the last many years, but have never written a socket program. We do not expect that you are familiar with C, and we do not expect you to learn C in order to develop socket programs. This book will illustrate most of the coding examples in COBOL, PL/I, assembler and REXX, which we believe is what most of you are familiar with. Some samples will also be shown in C.

We will focus on the Sockets Extended programming interfaces that are supplied with IBM TCP/IP Version 3 Release 1 for MVS. Emphasis will be on stream sockets, as the major part of all TCP/IP applications are stream socket applications.

Readers who already know sockets but who have little experience in IBM TCP/IP for MVS sockets may also benefit from this book.

**PREFACE.1 How This Document is Organized**

**PREFACE.2 Related Publications**

**PREFACE.3 Additional Publications**

**PREFACE.4 International Technical Support Organization Publications**

**PREFACE.5 Acknowledgments**

**PREFACE.1 How This Document is Organized**

The document is organized as follows:

- **Chapter 1, "Cooperative Applications"**

  In this introductory chapter we will present some of the fundamental design considerations you have to make before you decide on a specific application design.

- **Chapter 2, "Introduction to TCP/IP Programming Interfaces"**

  The socket programming interface is just one of the programming interfaces that are used in a TCP/IP-based network. In this chapter we give you a short introduction to each of the programming interfaces that are delivered with IBM TCP/IP Version 3 Release 1 for MVS.
Chapter 3, "TCP/IP Concepts for Socket Programmers"

We do not expect that you are an expert in TCP/IP protocols, but a few basic concepts must be understood in order to develop good socket programs. In this chapter we will explain these basic concepts, without going into too much detail. Other books are devoted solely to the purpose of explaining the TCP/IP protocols, and we will refer you to some of these books for more detail.

Chapter 4, "The IBM TCP/IP for MVS Socket APIs"

In this chapter we will introduce you to each of the individual socket programming interfaces that are delivered with IBM TCP/IP Version 3 Release 1 for MVS. These include C-sockets, Sockets Extended (both the call instruction API and the assembler macro API), REXX sockets, Pascal sockets and some of the older socket programming interfaces that were used with previous versions of TCP/IP for MVS. The main focus of this book is on the Sockets Extended programming interfaces.

Chapter 5, "Your First Socket Program"

This chapter includes all the basic socket programming techniques you need to develop socket programs in MVS. We will guide you through the development of a Sockets Extended iterative COBOL server program, and we will explain how you work with distinct messages in a stream protocol like the Transmission Control Protocol (TCP).

Chapter 6, "Native MVS Concurrent Server Program"

For those of you who have the requirement to develop high performance server applications in MVS, this chapter will give you information on how you develop a concurrent server in a native MVS address space. The concurrent server in this chapter is based on the Sockets Extended assembler macro interface.

Chapter 7, "Socket Client Programs"

It is expected that the majority of socket applications in an MVS environment is server applications, but you will from time to time also have to develop socket client programs in MVS. In this chapter we will add client specific information to what you learned about socket programs in the two previous chapters. The client issues are illustrated by a sample client that uses REXX sockets.

Chapter 8, "Datagram Socket Programs"

In the three previous chapters we confined ourself to stream sockets based on the Transmission Control Protocol (TCP). The majority of socket applications use stream sockets, but occasionally you may have the need for a datagram socket application. This chapter explains the specific characteristics of datagram socket applications based on the User Datagram Protocol (UDP).

Chapter 9, "IMS Sockets"

This chapter explains how you implement socket programs in an IMS dependent region. It also includes guidelines on how you can use the same IMS Message Processing Program (MPP) from both IBM 3270 terminals and socket clients.

Chapter 10, "CICS Sockets"
If your requirement is to implement socket programs as CICS transaction programs, this chapter will give you information on how you do that.

Chapter 11, "Debugging and Tracing Socket Programs"

From time to time it may be necessary to debug a socket program that does not behave as you intended. In this chapter we give you some guidance on how you handle socket return codes, and how you can trace the Internet Protocol (IP) packets that are forwarded over the IP network as a result of your socket calls.

This book includes a fairly extensive appendix, where we have placed a number of sample programs we developed as part of this redbook project. The samples are of no use by themselves; they only serve an educational purpose. We do not guarantee that the samples show the best or only way of implementing socket programs; they show how we implemented socket programs. We will refer you to specific samples throughout the text and we will encourage you to study them, as they might give you that extra clue on how you could proceed with your own socket programs. But remember: the best way to learn is to do it yourself.

All the COBOL samples are developed with the COBOL ANSI85 standard, which allows lowercase keywords and identifiers.

Appendix A, "Sample Datagram Socket Programs"

Sample datagram socket programs in COBOL and in C.

Appendix B, "Sample Stream Socket Programs"

Sample stream socket programs written in COBOL and in C.

The COBOL socket server is the sample iterative server program we refer to in Chapter 5, "Your First Socket Program" in topic 5.0.

Appendix C, "Sample IMS Socket Programs"

In this appendix you find sample explicit and implicit mode IMS socket programs written in COBOL, and sample remote clients used to test the IMS socket programs with, written in COBOL and C.

Appendix D, "Sample CICS Socket Program"

These are sample CICS socket programs. You will find a COBOL stream socket program that acts as an echo server in CICS, and you will find a C implementation of the EZACICSC sample program that is distributed with IBM TCP/IP Version 3 Release 1 for MVS.

Appendix E, "Sample REXX Socket Programs"

Here you find a couple of sample REXX socket programs. One of these samples is an implementation of a NETSTAT function in NetView. This function is based on a REXX client that runs in the NetView address space and communicates with a REXX server that runs in a batch TSO address space.

Appendix F, "Sample PLI Socket Programs"

This appendix contains sample PL/I socket programs that use the Sockets Extended API.
Appendix G, "Socket Utilities for Sockets Extended Programs"

This appendix contains a number of useful utility programs we developed for use from Sockets Extended programs.

Appendix H, "Sample MVS Concurrent Server"

This is an Sockets Extended assembler macro sample application that is implemented as a multitasking concurrent server in MVS.

Appendix I, "Sample Compile and Link JCL Procedures"

Here you find the compile and link edit JCL procedures that were used to compile and link the sample programs shown throughout this book.

**PREFACE.2 Related Publications**

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


IBM TCP/IP for MVS: Customization and Administration Guide, SC31-7134-00

IBM TCP/IP for MVS: Offloading TCP/IP Processing, SC31-7133

IBM TCP/IP for MVS: Programmer's Reference, SC31-7135

Performance Tuning Guide, SC31-7188

IBM TCP/IP for MVS: CICS TCP/IP Socket Interface Guide and Reference, SC31-7131

IBM TCP/IP for MVS: IMS TCP/IP Application Development Guide and Reference, SC31-7186

IBM TCP/IP for MVS: Application Programming Interface Reference, SC31-7187

IBM TCP/IP for MVS: Quick Reference, SX75-0095

**PREFACE.3 Additional Publications**


International Technical Support Organization Publications

TCP/IP Tutorial and Technical Overview, GG24-3376
CICS/ESA and TCP/IP for MVS Sockets Interface, GG24-4026
Client/Server Computing with IMS/ESA Using APPC, GG24-3981
IBM TCP/IP V3R1 for MVS Implementation Guide, GG24-3687

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

International Technical Support Organization Bibliography of Redbooks, GG24-3070.

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

TOOLS SENDTO WTSCPOK TOOLS REDBOOKS GET REDBOOKS CATALOG

How to Order ITSO Technical Bulletins (Redbooks)

| IBM employees in the USA may order ITSO books and CD-ROMs using PUBORDER. Customers in the USA may order by calling 1-800-879-2755 or by faxing 1-800-284-4721. Visa and Master Cards are accepted. Outside the USA, customers should contact their IBM branch office. |
| Customers may order hardcopy redbooks individually or in customized sets, called GBOFs, which relate to specific functions of interest. IBM employees and customers may also order redbooks in online format on CD-ROM collections, which contain the redbooks for multiple products. |

Acknowledgments

The advisor for this project was:
Alfred Bundgaard Christensen
International Technical Support Organization, Raleigh Center

The authors of this document are:
Alfred Bundgaard Christensen
International Technical Support Organization, Raleigh Center

Reinier B. Bakels
IBM Netherlands

This publication is the result of a residency conducted at the International Technical Support Organization, Raleigh Center.

Thanks to the following people for the invaluable advice and guidance provided in the production of this document:
Carla Sadtler
International Technical Support Organization, Raleigh Center
1.0 Chapter 1. Cooperative Applications

In the development of communication applications, such as TCP/IP applications, there are a number of common design topics that apply regardless of the actual technology being used. In this chapter we will discuss these topics in order to provide a common ground for the following chapters that are more implementation oriented.

TCP/IP programming is basically a question of creating cooperative applications based on the TCP/IP transport protocols and the programming interfaces associated with these protocols.

1.1 The Basic Socket Concept

Communications applications receive data from a network and send data to a network. In a way, communicating with a network is similar to reading from and writing to any other device.

The socket concept essentially builds a communications application program interface (API) on this similarity. The socket API is said to be based on an open/close/read/write paradigm, as the most important socket routine calls are similar to the calls that are used to access flat files.

For a positioning of sockets, we can extend the analogy with flat files. For many applications, flat files offer exactly the functions required by the application; while in other cases more functionality is required, and the application requirements are best met by a database system instead of a flat file. Under the covers, databases eventually are implemented as a set of flat files. Databases can be accessed through specialized APIs from user-written application programs. Also, databases can be accessed using tools such as query programs that may even make it unnecessary to write any application program yourself.

Similarly, TCP/IP provides application program interfaces beyond the socket interface. In addition, TCP/IP provides standardized tools: system applications, that require no user-written application at all (just as you can access a database with a query tool). Yet, internally most of these facilities are based on sockets, as is shown in Figure 1.
If you have decided that you are going to write TCP/IP applications, you have the choice to either use the socket interface directly, or use it through one of the other TCP/IP-provided interfaces, such as RPC (more about RPC in "Remote Procedure Call Programming Interfaces" in topic 2.3).

Finally, TCP/IP products provide built-in system applications for various purposes such as file transfer (FTP), terminal emulation (TELNET) and remote file access (Network File System).

Figure 2 shows the various TCP/IP facilities in perspective. It shows how user-written socket programs in fact use an interface that is also used by many TCP/IP system applications and facilities. Not all components shown in Figure 2 will be discussed in this book. For a comprehensive introduction we would recommend another redbook, TCP/IP Tutorial and Technical Overview, GG24-3376.

In most situations, it is the application requirements that dictate which solution is best. Sometimes the choice is obvious from an application point of view. At other times it is not obvious, and it might be useful to step back at some point in time (preferably early in a project) in order to critically review the requirements leading to a particular choice.
Occasionally you need a facility to access someone else's application over a network, or you create a service that can be accessed by someone else's application. In either case, you implement just one side of the communication, and the protocols and other application related specifications may be fully defined in advance.

On other occasions, you are in a position to design, from scratch, an application that partly runs on one machine and partly on another machine. Typically this is the case with intelligent workstations providing a graphical user interface to applications that run on other machines in the network where, for example, the database is located. Often this is called client/server computing, but actually client/server computing is just one of the models for cooperative application design, as we will discuss in the following sections.

1.2 Cooperative Application Design Models

In a cooperative application, one or more programs cooperate to implement the full application. When you design a cooperative application, you must decide where the application is split into separate programs, what the role of each program is, and how the programs communicate with each other in order to implement the appearance of a full cohesive application.

Cooperative application design can be very complex; however, viewed from a high-level perspective, the design considerations can be grouped into three major categories, which are known as:

1. The application model - where do you split your application into separate application parts?
2. The distribution model - what is the role of each application part?
3. The communication model - what happens between the application parts?

1.2.1 Application Model
1.2.2 Distribution Model
1.2.3 Communications Model

1.2.1 Application Model

The application model describes where your application is split, so one part of the application is executing on one system and other parts of the application are executing on other systems.
Figure 3. The Application Model – Where Do We Split the Application?

Three basic application models exist as follows:

1. Distributed Presentation

   The main body of the application, including data access and business logic, is in one system, while the user dialog is distributed to another system. For a distributed presentation application, you may consider using some standardized protocols. These may include a simple 3270 data stream front-ended with screen-scrapers of various kinds, distributed to a more sophisticated distributed dialog software like ISPF Version 4. In a TCP/IP environment, you have an excellent choice for distributed presentation applications, which is the X-Windows programming interfaces.

2. Distributed Function

   In a distributed function application, there is no easy way to characterize the split, which is located somewhere in the middle of your business logic. You have parts of the business logic in one system and other parts of the business logic on other systems. If you use this model, you have to design and implement your own application protocol. This would entail message formats, state switching, and exception handling, just to mention the most important aspects of such a design.

   Most of your TCP/IP socket based applications will probably be located within this category.

3. Distributed Data Access
With distributed data access, you have your business logic and presentation logic in one system, and your data or parts of it on another system. In this area, you will find more very powerful standardized protocols ranging from Structured Query Language (SQL) using Distributed Relation Data Architecture (DRDA) to simpler remote file access protocols like Network File System (NFS) in a TCP/IP environment or Distributed Data Management (DDM) in an SNA environment.

You may also use the Network DataBase (NDB) component of IBM TCP/IP for MVS for distributed access to DB2/MVS databases, but be aware that the NDB protocols do not implement any two-phase commit functions, so your local data updates are not synchronized with your DB2/MVS updates.

Distributed data access is often an attractive solution for implementing cooperative applications because your programs, with a correct implementation, are more or less unaware of the fact that the data is being accessed across a network. It makes it easy to implement programs and relatively easy to port them to other platforms.

Distributed data access may be very easy to implement, but there are situations, not only where application characteristics can lead to unacceptable poor performance in a distributed data access environment, but also where a distributed function design might prove to be a more feasible solution, at least from a performance point of view.

1.2.2 Distribution Model

The distribution model describes how the two parts of your application are distributed and how they interact with each other. The following are the three basic distribution models:

1. A Peer-to-Peer model

In this model, no single part of the application is by definition slave or master, both parts are peers and both parts can initiate or terminate a conversation with the other.

This model may well be flexible, but it is often difficult to implement in real life. It can in many situations be substituted by two applications that are based on the following client/server distribution model instead.

```
|Process 1| <_______> |Process 2|
    |________|           |________|
```

Figure 4. Peer-to-Peer Distribution Model

2. A Client/Server model
This is the most common distribution model. One part of your application (the client side) makes requests of the other part (the server side) which performs some service and sends back a reply.

The roles of the client and the server are specialized and constrained to a certain predefined type of interaction and function.

---

### Figure 5. Client/Server Distribution Model

This model is relatively simple to implement, and most of your cooperative applications will most likely use the client/server distribution model.

The socket programming interface offers you a set of function calls that are very useful in making it easy for you to write cooperative applications that are based on the client/server distribution model.

3. A Processor Pool model

A processor pool model is very useful for parallel processing, as its basic idea is to have a coordinator process break a given job into small pieces, parcel these pieces out to a number of parallel work processes and finally assemble the individual result pieces into a final result.

This distribution model will be used for high performance specialized applications. It is difficult to implement.

### 1.2.3 Communications Model

The third and final design model category describes what happens between the two parts of your application. How is the communication flow?

Basically communication models can be grouped into three major groups:

1. A Conversational model

In a conversation, the two application parts implement a half-duplex, flip/flop application protocol. This has nothing to do with the transport protocol, which may very well be a full-duplex protocol like a TCP protocol. Here we focus on how the two application parts control their conversation. The conversation is synchronous; one side sends data, and the other side receives data. All components of the network must be available for a conversation to take place. Application programs must include logic to deal with network failures, which will break the conversation at unpredictable points. A well-implemented conversational protocol may include elements for coordinating synchronization points. If your distributed application
relies on synchronized updates of data at both involved end points, you will probably use a conversational model.

SNA LU6.2 protocols, with Common Programming Interface - Communications (CPI-C) used as a programming interface, is an excellent choice for an application that requires a conversational model.

---

**Figure 6. Conversational Communications Model**

If you are using the socket programming interfaces, you have to build your own conversational protocol control (state control and state transition logic) into your application, because the socket programming interface does not enforce conversation states.

2. Remote Procedure Call model

This is a well-known communications model in the TCP/IP community where you will find more implementations based on this model. The basic concept of this model is call subroutine and return results.

Communication between the parts is synchronous; the requestor calls a subroutine and blocks until the subroutine returns. In an RPC implementation, the subroutine is not part of the calling program but may be located on another system to which the call parameters are passed. The routine is executed and the return parameters are returned to the originating system where they are passed back to the calling program.

The Remote Procedure Call model is a simple and easy-to-use communication model.
IBM TCP/IP for MVS implements both SUN's Open Network Computing / Remote Procedure Call (ONC/RPC) and Appolo's Network Computing System / Remote Procedure Call (NCS/RPC).

In Open Software Foundation / Distributed Computing Environment (OSF/DCE), you will find a third remote procedure call implementation, which is called OSF/DCE Remote Procedure Call (DCE/RPC). In an MVS environment, DCE/RPC is implemented by MVS/ESA OpenEdition Distributed Computing Environment.

The CICS function called Distributed Program Link (DPL) between, for example, CICS-OS/2 and CICS/ESA is also an implementation of a remote procedure call model.

3. Message Queuing model.

The first two communication models were synchronous in the sense that the two parts were in direct interaction with each other. In a message queuing model, the requester queues a request for the receiver to process at some time later. The requesting and the receiving processes are fully asynchronous in nature. No connection exists between the two.

Message queuing inside one operating system has been known for many years, but it is not until recently that message queuing between heterogeneous environments has been implemented in commercial products.

To implement distributed applications based on a message queuing model, you may use the IBM MQSeries products, which implement recoverable store-and-forward queues and a uniform message queuing programming interface (MQI) across a range of operating system platforms.

Figure 7. Remote Procedure Call Communications Model
1.3 Cooperative Design Summary

In your design of a cooperative application, you have to consider the aspects of each of the models just described. Just to recap a few of these aspects:

Where can you split your application logic?
How well will alternative implementations perform?
How much effort must you put into implementing each of your alternatives?
How do you synchronize updates if such synchronization is required?

It really does not matter if you are able to refer your application to a specific set of models or not; what matters is that you consider the aspects of each model when you lay out the design of your cooperative application.

2.0 Chapter 2. Introduction to TCP/IP Programming Interfaces

In this chapter we will give you a short introduction to each of the programming interfaces that are delivered with IBM TCP/IP Version 3 Release 1 for MVS.

IBM TCP/IP Version 3 Release 1 for MVS gives you a broad range of programming interfaces that you can use to develop application programs that interact with the TCP/IP protocol layers and services at various levels.

Some of the programming interfaces are general use interfaces like the socket and Remote Procedure Call interfaces. Others are special purpose programming interface, like the SNMP DPI interface, which you will only use if you have to develop an SNMP subagent.

2.1 Choosing an API
2.2 Socket Application Programming Interfaces
2.3 Remote Procedure Call Programming Interfaces
2.4 X-Windows Programming Interfaces
2.5 X/Open Transport Interface (XTI)
2.6 SNMP Agent Distributed Programming Interface (DPI)
2.7 Kerberos Programming Interface

2.1 Choosing an API

The sockets application programming interface is often referred to as low-level, as opposed to interfaces such as RPC that are considered to be high-level.
What should you choose for your particular application? Will it always be a good idea to use a high-level programming interface? Well, it depends. Typically, a high-level interface offers more ease of use at the expense of flexibility. Sometimes, you would need flexibility. On other occasions, the standardized functionality of a high-level interface is exactly what you want.

Exploiting the facilities of IBM TCP/IP Version 3 for MVS is important for maximum application development productivity.

This book will help you decide whether to use sockets or not. On several occasions, we will tell you explicitly that sockets require you to decide yourself on certain design aspects. What this means is that sockets give you the freedom to decide on those aspects yourself, which may be really what you want.

Whether you are going to use the basic programming interfaces or you are going to use one of the higher-level interfaces for your application depends on many factors.

- What functions do you require in the programming interface?
- What are the operating system platforms you have to support with your application?
- Which transport protocols do the operating systems support?
- If a higher-level programming interface suits your needs, is the supporting program products available on all the operating systems where you want to implement your application?
- Can your application justify the cost of purchasing the supporting program products if it is available but not implemented?
- What are your programming skills, do you need extra training in order to use a specific programming interface and can you get that training in the right time before your development project starts?

Other factors, like company policy, may of course influence the choice of programming interface.

For some of your applications, you will probably end up with a choice of basic TCP/IP socket programming in MVS.

2.2 Socket Application Programming Interfaces

The socket programming interface has been implemented in more variations, but all implementations are in some way or another based on the original Berkeley Software Distribution (BSD) socket implementation, which has its roots in the UNIX environment. As we explained in the introduction, socket programming is a generalized file access mechanism where your socket programs interact with a socket in much the same way they would interact with a file. You open and close a socket. You read and write data from and to a socket. In a C program, you will actually use the same system calls to access a socket and a file.

You must understand that, when we talk about sockets, we are talking about a programming interface, not a protocol. The socket programming interface is a programming interface to the TCP/IP protocol layers (mainly the Transport Control Protocol (TCP) and User Datagram Protocol (UDP) layers).
But the socket programming interface also supplies you with interfaces to the Internet Protocol (IP) layer directly, if you want to develop special purpose network control applications. See Figure 9 for an overview of the relationship between the socket interface and the protocol layers.

---

|  ______    ________  |  ______  |  ______  |
| |Stream|  |Datagram| programming|  |Raw   | |
| |socket|  |socket  |  |socket| |
| |_ ^ __|  |__ ^ ___|             |____|_____|
|____|__________|_______           |
|___ V __    __ V ___   |  ______  |
|        |  |        |  | |Raw   | |
|  TCP   |  |  UDP   |  | |_ ^ __| |
|        |  |        |  |____|_____|
|_______________________|___________________|__________________|

Figure 9. Socket Programming Interface

The socket programming interface is a general-use, wide spread and very flexible programming interface. It can be used for almost any application type you want to implement. On the other hand, this is also the weakness of the socket programming interface. The socket programming interface does, for example, not include functions to establish and to control conversation states between two applications that exchange data over a socket connection. If conversation states make sense in the application, then the application designer must design a conversation protocol based on both logic in the application programs and state data transmitted as part of the user data. If data is exchanged over a socket connection between programs that execute on different hardware platforms, then the programs must include logic to convert data from one data representation to the other.

In an MVS environment, applications run either natively on MVS (batch, TSO or started task), or exploit specific subsystems such as CICS or IMS. All of these subsystems provide several alternatives for TCP/IP applications.

---

Table 1. IBM TCP/IP Version 3 Release 1 for MVS Socket Libraries and MVS Environments

<table>
<thead>
<tr>
<th>IBM TCP/IP Version 3 Release 1 for MVS Socket Libraries</th>
<th>Native MVS or TSO</th>
<th>CICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-sockets</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>REXX Sockets</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PASCAL Sockets</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

---
TCP/IP for MVS Version 3 offers you the opportunity to develop socket programs in both the C language and other high-level languages, as is shown in Table 1.

In addition to the listed IBM TCP/IP Version 3 Release 1 for MVS socket libraries, you can in an MVS environment use OpenEdition/MVS sockets and AnyNet/MVS sockets. We will in "Integrated Sockets" in topic 3.6.1, return to a discussion of the relationship between these different socket libraries.

C-sockets.

This programming interface is based on the original BSD socket definitions and is widely used in the UNIX world. A C program using this interface can be ported between MVS and most UNIX environments with relative ease, if the program does not use any other MVS specific services.

C-socket applications can be implemented in normal MVS address spaces, CICS, and IMS transaction programs.

Sockets Extended - call interface.

This is a generalized call-based, high-level language interface to socket programming. The functions implemented in this call interface resembles the C-socket implementation, with some minor deviations. The Sockets Extended call interface is available to COBOL, PL/I or assembler programmers.

Sockets Extended call based applications can be implemented in normal MVS address spaces, CICS, and IMS transaction programs.

Sockets Extended - assembler macro interface.

This programming interface is fundamentally the same as the Sockets Extended call interface, but it is implemented as assembler macros, which adds some extra features like multitasking support and support for asynchronous socket calls.

This programming interface can only be used to implement socket applications in normal MVS address spaces (batch, TSO or started task).

REXX sockets.

This programming interface implements facilities for socket communication directly from REXX programs via an address socket function.

REXX socket programs can execute in TSO (either TSO online or TSO batch) and in NetView.
Pascal sockets.

This is a Pascal socket interface allowing programmers to develop socket applications in Pascal language.

Environments supported are normal MVS address spaces.

While this API conceptually provides the same sockets interface, the actual implementation in routines is fairly different.

IUCV and VMCF sockets.

These are assembler macro based interfaces, which are relatively low-level and complex to use. These APIs are primarily included in IBM TCP/IP Version 3 Release 1 for MVS for compatibility reasons.

Reference information for all the IBM TCP/IP for MVS socket programming interfaces can be found in *IBM TCP/IP for MVS: Application Programming Interface Reference*, SC31-7187.

2.3 Remote Procedure Call Programming Interfaces

A Remote Procedure Call programming interface is located at a higher level in the protocol stack than the socket based programming interfaces. Somewhere down underneath the RPC interface the socket programming interface is used, but the details of the socket interface are hidden for the application programmer that uses the RPC programming interface.

The RPC programming interfaces offer more ease of use to the application programmer than do the socket programming interfaces which makes the network programming job somewhat easier to accomplish. The RPC
programming interfaces generally deal with things like different data representation and some kind of state control over the dialog. On the other hand this also implies some restrictions; a dialog is normally limited to one procedure call. Each remote procedure call is stateless and independent of either preceding or succeeding calls. If an RPC client program requires more interactions with the server program, the state data has to be carried back and forth as user data in the parameters passed on each remote procedure call, or the server program has to implement some kind of Scratch Pad Area (SPA) implementation where state data per client is saved from call to call.

If you develop RPC programs, your only programming language choice is C.

In IBM TCP/IP for MVS you have two RPC implementations:

Sun Microsystems Open Network Computing / Remote Procedure Call (ONC/RPC).

Hewlet Packard Remote Procedure Call implementation, which is called Apollo Network Computing System / Remote Procedure Call (NCS/RPC).

Please refer to IBM TCP/IP for MVS: Programmer's Reference, SC31-7135, for reference information on both ONC/RPC and NCS/RPC.

2.4 X-Windows Programming Interfaces

If you want to develop distributed presentation programs, where your application program is running in MVS and the user interface is implemented on an X-Windows server in your IP network, you can use the X-Windows application programming interfaces that are supplied with IBM TCP/IP for MVS to develop X-Windows MVS client programs.

In an X-Windows environment, the term server is applied to the host where the display shows up, and the term client is applied to the host where the application program is executing.
One X server may be connected to many X clients thus sharing the physical display and input devices among many application programs. The clients may be located on different hosts.

MVS is only able to act as an X-Windows client, not as an X-Windows server which means you can execute X-Windows applications in MVS that communicate with X-Windows servers in TCP/IP workstations.

The X-Windows programming interfaces are, like the RPC programming interfaces, a higher level programming interface to the socket interface. But unlike the RPC programming interface, which is a general use interface, the X-Windows interface is a specialized programming interface that deals only with distributed presentation.

The X-Windows programming interface in IBM TCP/IP Version 3 Release 1 for MVS is based on the X11.4 specification. The X11.4 programming interface is extremely detailed and gives you a high number of low-level functions. On top of the basic X11.4 programming interface, you find some toolkits that implement generally used X-Windows functions (also called intrinsic functions). You can use the toolkits to develop X-Windows applications without the detailed coding you would have to use if you only had the X11.4 interface.

At an even higher level than the X-Windows toolkits, you find what is termed X-Windows widget sets. A widget set is a collection of procedures or functions that you use to create commonly used X-Windows objects. Examples of such objects are the following:

- Push buttons
- Scroll bars
- Dialog boxes
- Text boxes
- Pull-down menus

The widget sets that are supplied with IBM TCP/IP for MVS are:

- The Athena Widget set from Massachusetts Institute of Technology (MIT).
- The OSF/Motif Widget set release 1.1 from the Open Software Foundation (OSF).

You can only develop X-Windows programs in C.


2.5 X/Open Transport Interface (XTI)

IBM TCP/IP for MVS implements an XTI programming interface in C that allows you to use XTI programs in a TCP/IP environment.

XTI is defined by X/Open and is a superset of UNIX System V Transport Layer Interface (TLI), which is a programming interface introduced in UNIX System V.
The IBM TCP/IP for MVS implementation of XTI includes a mapping component which maps between XTI calls and TCP socket calls. The mapping component is based on RFC1006.

The XTI system calls are implemented as C function calls, so you must develop your XTI application in C.

For details about the XTI programming interface, please study CAE Specifications: X/Open Transport Interface (XTI) and IBM TCP/IP for MVS: Application Programming Interface Reference, SC31-7187.

2.6 SNMP Agent Distributed Programming Interface (DPI)

This is a special purpose programming interface that you can use if you want to implement dynamic Management Information Base (MIB) variables. In an SNMP environment, the MIB variables are defined in the tcpip.v3r1.MIBDESC.DATA data set. If you want to dynamically add, replace or delete MIB variables, you can develop an SNMP subagent program that uses the DPI programming interface to interact with the SNMP agent address space (SNMPD) to perform such functions.

If you develop an SNMP subagent, you can define your own MIB variables and SNMP traps.

The connection between the subagent address space and the SNMP agent is established as a TCP socket connection, so the DPI programming interface is again a higher level programming interface to the socket interface.

It is outside the scope of this book to explain the DPI programming interface in detail. For the socket based parts of an SNMP subagent you
2.7 Kerberos Programming Interface

Kerberos is an authentication system that you can use to identify clients and authenticate connection requests. Authentication depends on both client and server programs to include specific system calls to the Kerberos Authentication Server (KAS) and Ticket Granting Server (TGS).

The Kerberos calls are implemented in C, so you can only use the Kerberos authentication features if you develop your programs in C.

This book does not include details about the Kerberos program calls., You can find call reference information in IBM TCP/IP for MVS: Programmer’s Reference, SC31-7135.

3.0 Chapter 3. TCP/IP Concepts for Socket Programmers

This chapter explains some very basic TCP/IP concepts which are required in order to understand the remaining parts of this book. We will not indulge into too much detail, but we will focus on the concepts where an explanation may ease your understanding of the TCP/IP socket programming issues that are presented in the succeeding chapters.

For a more thorough explanation of TCP/IP concepts, we refer you to TCP/IP Tutorial and Technical Overview, GG24-3376.

3.1 TCP/IP Protocol Layers
3.2 Addresses
3.3 Sockets
3.4 Socket Types
3.5 Encapsulation
3.6 Addressing Families
3.7 General Socket Program Structure

3.1 TCP/IP Protocol Layers

The TCP/IP protocol stack consists conceptually of four layers, each layer consisting of more protocols.

We will define a protocol as a set of rules or standards that two entities must follow to allow each other to receive and interpret messages sent to them. The entities could, for example, be two application programs, in which case we talk about an application protocol. The entities could also be two TCP protocol layers in two different IP hosts, in which case we talk about the TCP protocol.

<table>
<thead>
<tr>
<th>Process</th>
<th>User</th>
<th>User</th>
<th>User</th>
<th>User</th>
<th>OSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer</td>
<td>Process</td>
<td></td>
<td>Process</td>
<td></td>
<td>Process</td>
</tr>
<tr>
<td></td>
<td>______</td>
<td>_______</td>
<td>_______</td>
<td>_______</td>
<td></td>
</tr>
</tbody>
</table>
Your programs are located at the process layer, where they may interface either to the two transport layer protocols (TCP and UDP) or directly to the network layer protocols ICMP and IP.

**TCP**

Transmission Control Protocol

TCP is a connection-oriented transport protocol that provides a reliable, full-duplex byte stream. By far the majority of TCP/IP applications use the TCP transport protocol. It is estimated that between 80% and 90% of all TCP/IP applications are based on TCP, which is the reason why this book devotes most of the pages to explaining how to create TCP based applications.

**UDP**

User Datagram Protocol

UDP is a connectionless protocol that provides datagram services. There are no guarantee that a UDP datagram ever reaches its intended destination, or that it reaches its destination only once and in the same shape as it was passed to the sending UDP layer by a UDP application.

**ICMP**

Internet Control Message Protocol

ICMP is used to handle error and control information at the IP layer. ICMP is mostly used by network control applications that are part of the TCP/IP software product itself, but ICMP may be used by authorized user processes as well. PING and TRACEROUTE are examples of network control applications that use the ICMP protocol.

**IP**

Internet Protocol

The IP layer provides the packet delivery services for TCP, UDP and ICMP. The IP layer protocol is in itself an unreliable and so-called best-effort protocol. There is no guarantee that IP packets will arrive to the destination or that they will arrive only once and error-free. Such reliability features are built
into the TCP protocol, but not into the UDP protocol. If you want a reliable transport between two UDP applications, the reliability functions must be built into the UDP applications.

**ARP**  
Address Resolution Protocol

This protocol is used by the networking layer to map an IP address into a hardware address. On a local area network, such an address would be a Media Access Control (MAC) address.

**RARP**  
Reverse Address Resolution Protocol

As the name suggests, this protocol is used to do the reverse operation of the ARP protocol: map a hardware address into an IP address.

Please note that both ARP packets and RARP packets are not forwarded in IP packets, but are media level packets themselves. ARP and RARP are not used on all network types as some networks do not need these protocols.

### 3.2 Addresses

One of the most basic requirements for network programming is the ability to find your communication partner by address or name.

From the perspective of an application program, the identity of a TCP/IP communication partner is defined in two steps:

1. The first step is the address of the machine where the partner application is running. In an IP network, this is the IP address.

2. The second step is to identify the specific application on that machine. This is done through the port number.

#### 3.2.1 IP Addresses

#### 3.2.2 Ports

### 3.2 / IP Addresses

In the current version of the IP protocol (version 4), an IP address occupies 32 bits. These 32 bits are divided into a network part and a host part.

The split between the network part and the host part is determined by the *address class*. The first bits in an IP address identify the address class.

<table>
<thead>
<tr>
<th>Class A</th>
<th>0</th>
<th>netid</th>
<th>hostid</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7-bit netid and 24-bit hostid: 0.0.0.0 to 127.255.255.255</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class B</th>
<th>10</th>
<th>netid</th>
<th>hostid</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Beginner's Guide to MVS TCP/IP Socket Programming

14-bit netid and 16-bit hostid: 128.0.0.0 to 191.255.255.255

Class C  |110| netid | hostid |
        |   |       |       |
21-bit netid and 8-bit hostid: 192.0.0.0 to 223.255.255.255

Figure 14. IP Address Classes

In addition to the three IP address classes presented in Figure 14, class D and E also exist. Class D addresses are called multicast addresses, and class E addresses is a group of addresses that is currently reserved for future use.

Every IP datagram contains the full 32-bit source IP address and the full 32-bit destination IP address in the 20-byte IP header. IP routers on the path, between the source and the destination IP host, only need to look at the IP addresses of an IP datagram in order to determine where to forward the IP datagram.

IP addresses in the form of numbers are hard to remember. And, as they are related to the internal structure of your IP network, they tend to change. A person may move to a different office and get a different IP address; but, of course, remains the same person.

For those reasons, TCP/IP provides facilities to assign a symbolic name to an IP address. Such a name is known as a host name.

The translation between host names and IP addresses is performed by a component called the name resolver. This component is part of every TCP/IP product. The name resolver finds its information in either some local host tables or it queries a special server called a name server. The socket programming interface includes calls you can use to translate a host name to an IP address or an IP address to a host name. These calls are gethostbyname and gethostbyaddr.

In a TCP/IP network, there are several other types of addresses, such as physical (LAN) addresses. TCP/IP software handles all of these addresses, so your application should not be concerned with those.

An IP address is by tradition expressed externally in dotted decimal form and internally in a 32 bit wide field. In a C-program you can use two library routines to convert an IP address from one format to the other:

inet_addr Converts a null-terminated character string to a full-word IP address
inet_ntoa Converts a full-word IP address to a null-terminated character string

In the C programming language a variable length character string is terminated with a hexadecimal zero (X'00'). This is the reason why such a string is called a null-terminated string.
If you are writing your socket programs in languages other than C, you have to develop similar routines. You will find, in the appendix of this book, examples of two assembler routines that implement similar functions. These routines can be called from any high-level language that supports an assembler call interface. See "TP1INTOA Convert IP Address to Character String" in topic G.2, and "TP1IADDR Convert IP Address Character String to Full-word" in topic G.3.

An IP host may have more IP addresses. Such a host is, in IP terms, called a multihomed host. Actually an IP address does not identify an IP host but rather an IP network interface on an IP host.

Any multihomed host may act as an IP router. MVS TCP/IP allows the system programmer to disable IP routing for a multihomed MVS host by setting the NOFWD option in the tcpip.v3r1.TCPIP.PROFILE configuration data set.

If your MVS TCP/IP host has two IP network interfaces, for example, a token-ring interface and an Ethernet interface, it will have two distinct IP addresses; one for each network interface.

You have to consider the multihomed aspects of a host when you write socket programs. A server program can specify if it will accept client requests from all available network interfaces, or only from a specific network interface. A client program that sends requests to a server on a multihomed host may use any of the supplied network interfaces of the server host, if the server accepts requests on all the network interfaces.

In Figure 15, the MVS system with a host name of mymvs has two IP addresses (one for each physical network interface).
for a host name, the name resolver will return not one IP address, but a list of IP addresses: one for each registered network interface for the host in question. It is a good programming practice to take the full list of IP addresses into consideration when you write your client program. If a connect to the first IP address in the returned list does not respond, your program should include code to pick up the next IP address in the returned list and try to connect to that one. If you write your client programs this way, you build into the code dynamic backup options for failed network interfaces on the server host.

In the example in Figure 15, the application that runs on host myaix will receive both IP address 10.0.1.1 and 10.0.2.1 on a gethostbyname call for the host name mymvs. If, for example, the Token-ring interface on mymvs is down, the client application on myaix will not be able to connect to address 10.0.1.1; but it will be able to successfully connect to address 10.0.2.1, which is on the Ethernet LAN.

3.2.2 Ports

A socket program in an IP host identifies itself to the underlying TCP/IP protocol layers by a port number.

A port is a 16-bit integer ranging from 0 to 65534. A port uniquely identifies this application to the underlying protocol (TCP, UDP or IP) in this TCP/IP host. Other applications in the TCP/IP network may contact this application via reference to the port number on this specific IP host.

```
<table>
<thead>
<tr>
<th>Application</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process no. 1</td>
<td>Process no. 2</td>
</tr>
<tr>
<td>Port</td>
<td>Port</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>TCP or UDP</td>
<td></td>
</tr>
<tr>
<td>IP and ICMP</td>
<td></td>
</tr>
<tr>
<td>Network hardware interface</td>
<td>Hardware address</td>
</tr>
</tbody>
</table>
```

Figure 16. The Port Concept

Both server applications and client applications have port numbers. A server application will use a specific port number that uniquely identifies this server application. The port number can be reserved for this particular server so no other process ever uses it. In an IBM TCP/IP for MVS environment, you can do so via the PORT statement in the tcpip.v3r1.PROFILE.TCPIP configuration data set. When the server application initializes, it will, via the bind socket call, instruct the underlying protocol layers what its port number is. A client application must know the port number of a server application in order to be able to contact it.

Normally, no one needs to have advance knowledge of the port number of a
client, so a client leaves it often to TCP/IP to assign a free port number when the client issues the `connect` socket call to connect to a server. Such a port number is called an ephemeral port number, which means it is a port number with a short life. The selected port number is assigned to the client for the duration of the connection and is then made available for other processes to use. It is the responsibility of the TCP/IP software to ensure that a port number is only assigned to one process at a time.

Some application processes are themselves standardized protocols, such as FTP, SMTP, or TELNET. Such standardized applications will use the same port number on all TCP/IP hosts. These port numbers are called well-known ports and they represent well-known services. Well-known official Internet port numbers are all in the range from 0 to 255. You can find a list of these port numbers in Assigned Numbers, RFC1700. In addition, port numbers in the range 256 to 1023 are reserved for other well-known services. Port numbers in the range from 1024 to 5000 are used by TCP/IP when TCP/IP automatically assigns port numbers to client programs that do not use a specific port number. Your server applications should use port numbers above 5000.

<table>
<thead>
<tr>
<th>Port Numbers</th>
<th>0 - 255</th>
<th>256 - 1023</th>
<th>1024 - 4999</th>
<th>5000 - 65534</th>
</tr>
</thead>
<tbody>
<tr>
<td>Official</td>
<td></td>
<td>Ephemeral</td>
<td>Your well-known</td>
<td></td>
</tr>
<tr>
<td>Internet</td>
<td>Well-known ports</td>
<td>server ports</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Services</td>
<td>Services</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 17. Port Number Assignment

Before you select a port number for your server application, you should consult the tcpip.v3r1.ETC.SERVICES data set. This data set is used to assign port numbers to server applications. The server application can use the `getservbyname` socket call to retrieve the port number assigned to a given server name. You may add the names of your server applications to this data set and use the `getservbyname` call. Using this technique, you avoid hard coding the port number into your server program. The client program must know the port number of the server on the server host. There is no socket call to obtain that information from the server host. One way to handle this could be to synchronize the contents of the ETC.SERVICES data sets on all TCP/IP hosts in your network. Your client application could then use the `getservbyname` socket call to query its local ETC.SERVICES data set for the port number of the server. Using this technique, you develop your own locally well-known services.

3.3 Sockets

A port represents an application process on a TCP/IP host, but the port number itself does not indicate what protocol is being used: either TCP, UDP or IP. The application process may use the same port number for all three protocols. To uniquely identify the destination of an IP packet that arrives over the network, we have to extend the port principle with information about the protocol used and the IP address of the network interface; this union is called a socket.

A socket is made up of 3 components:
A socket uniquely identifies the endpoint of a communication link between two application ports.

---

<table>
<thead>
<tr>
<th>Application</th>
<th>CONNECTION</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process A</td>
<td>&lt;_________&gt;</td>
<td>Process B</td>
</tr>
<tr>
<td>Port 1028</td>
<td>Port 2034</td>
<td></td>
</tr>
<tr>
<td>TCP</td>
<td>TCP</td>
<td></td>
</tr>
<tr>
<td>IP 9.67.38.96</td>
<td>9.67.38.92</td>
<td></td>
</tr>
<tr>
<td>Network intf.</td>
<td>Network intf.</td>
<td></td>
</tr>
</tbody>
</table>

Socket A = {TCP, 9.67.38.96, 1028}
Socket B = {TCP, 9.67.38.92, 2034}

---

The term association is used to completely specify the two processes that comprise a connection:

\{protocol, local-address, local-port, foreign-address, foreign-port\}

A socket is also called a half association or a transport address.

If you have knowledge about SNA, some of these terms may seem familiar to you. The network part of an IP address resembles the SNA network name. The host part of the IP address resembles a System Services Control Point (SSCP) in an SNA subarea network, while the port number resembles a Logical Unit (LU) that is owned by that SSCP. A socket resembles a half-session, and the association resembles an SNA session.

The terms socket and port are sometimes used as synonyms, but please note that the terms port number and socket address are not synonymous. A port number is one of the three parts in a socket address. A port number can be represented by a single number; for example, 1028 and a socket address can be represented by \{tcp,myhostname,1028\}.

A socket descriptor (or sometimes referred to as a socket number) is a binary half-word (2 byte integer) that acts as an index into a table of sockets currently allocated to a given process. A socket descriptor represents a socket but is not the socket by itself.

### 3.4 Socket Types

When you write socket programs, you have to select what kind of service you require from the transport protocol layer.
Three different socket types are defined as follows:

**Stream socket** - a stream socket is characterized by:

- Connection-oriented, which means that the transport layer representing the two sockets establish a logical connection before they begin to exchange data.
- Full-duplex, which means data can be transmitted in both directions simultaneous.
- Reliable, which means that error-free data delivery is guaranteed in right order and without duplication.
- Byte stream - no boundaries are imposed on the data. The data being transmitted can be of virtually unlimited size.
- Flow control, which guarantees that the sender does not send data faster than the network and the receiver is able to manage.

The default protocol for such a service in a TCP/IP network is the TCP protocol. FTP is an example of an application that uses stream sockets.

**Datagram socket** - a datagram socket is characterized by:

- Connectionless, which means that datagrams are transmitted over the network without first establishing a connection between the two sockets. Each datagram must contain the full set of addressing information required for its delivery.
- No reliability guaranteed, which means that data may be duplicated, out of order, corrupted or never sent.
- No flow control, which means that a sender may monopolize the network and send datagrams faster than the receiver can manage.
- Messages have a maximum size. If you want to send more data than the amount you can send in a single datagram, you must send more independent datagrams.

The default protocol for such a service in a TCP/IP network is the UDP protocol. NFS is an example of an application that uses datagram sockets.

**Raw socket** - a raw socket can be characterized by:

- Access to lower-level protocols (IP and ICMP).
- Connectionless.
- Reliability not guaranteed.
- Messages have a maximum size.

PING is an example of an application that uses raw sockets.

Normally, you would not use raw sockets unless you intend to develop TCP/IP system software functions yourself.
Study Table 2 for an overview of the three socket types and a guide on when to use which one.

<table>
<thead>
<tr>
<th>Table 2. Which Socket Type to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Stream</td>
</tr>
<tr>
<td>Reliable?</td>
</tr>
<tr>
<td>Performance?</td>
</tr>
<tr>
<td>Data size</td>
</tr>
<tr>
<td>Protocol used?</td>
</tr>
</tbody>
</table>

A stream socket represents a connection-oriented protocol, while a datagram socket represents a connectionless-oriented protocol.

Figure 19 illustrates the typical socket calls that are used for a connection-oriented protocol. Other calls exist, but those shown here are the typical calls you will use.

Any number of send and receive calls from either side is possible. The figure primarily illustrates connection initiation and termination procedures.

The listen/accept sequence by definition characterizes a connection-oriented server, whereas the client is characterized by the connect call.

---

SERVER

<table>
<thead>
<tr>
<th>socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bind</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>listen</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>accept</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
</tr>
</tbody>
</table>

blocks until connection establishment <---------- connect |

<table>
<thead>
<tr>
<th>recv</th>
</tr>
</thead>
</table>

| SOCKET |

CLIENT

<table>
<thead>
<tr>
<th>socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>connect</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data (request to server)</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>send</th>
</tr>
</thead>
</table>

---
Figure 19. Socket Calls for a Connection Oriented Protocol

Figure 20 illustrates the typical socket calls for a connectionless protocol. Other calls exist, but those shown are the typical calls you will use.

The important thing to note here is that there is no connection establishment.

Both the connectionless server and client will use a bind call. The client does it in order to ensure it has a unique address, so the server may be able to send a response back to it. You can compare it to placing a valid return address on an envelope.

Normally a connectionless application will not use a connect call, but it may do so. In that case, no connection is established, but the connect call just stores the peer address of the partner application. Anything sent on the socket goes to that address, and only data from that peer address is returned to the application that issued the connect call.
3.5 Encapsulation

When your program, which is located at the process layer in the TCP/IP protocol stack, passes data to the underlying protocol layers, each protocol layer will add some extra bytes in front of your data and, for certain protocols, also extra bytes following your data. This process is called encapsulation and is the source of most of the confusion about datagram sizes, IP packet sizes and MTU (Maximum Transmission Unit) sizes.

See Figure 21 for an overview of the encapsulation process.

The MTU size dictates the maximum size of an IP datagram that can be transmitted over a given interface, given the physical characteristics of that interface.

See Table 3 for an overview of typical MTU sizes.

<table>
<thead>
<tr>
<th>Network Interface</th>
<th>MTU size in bytes</th>
</tr>
</thead>
</table>
If the size of an IP datagram exceeds the MTU value of your network interface, the IP layer will fragment the IP datagram and transmit the TCP segment as a number of IP datagram fragments.

From a performance point of view, it is normally advisable to prevent fragmentation. The TCP protocol works with a unit that is called a TCP segment. When a TCP segment is passed to the IP layer, a 20-byte IP header is added to the TCP segment to form an IP datagram. The size of a TCP segment is determined by the two TCP protocol layers that are involved in setting up a TCP connection.

Your application program cannot influence the decision made by TCP in determining the segment size. Your application passes a stream of bytes to the TCP layer. It is up to the TCP layer to chop your data up into TCP segments and send these segments over the IP network. The receiving TCP layer assembles the TCP segments into the right order and passes them to your application as a stream of bytes without any apparent boundaries.

If you use UDP protocols instead of TCP, your data is placed into a UDP datagram preceded with an 8 byte UDP header. If you pass 8192 bytes of data to the UDP layer, a UDP datagram of 8200 bytes is handed over to the IP layer. Sending that size UDP datagrams will almost always result in IP datagram fragmentation. Many UDP applications restrict themselves to sending UDP datagrams that do not exceed 512 bytes in order to reduce the risk of fragmentation.

### 3.6 Addressing Families

Until now we have more or less let you believe that socket programming only was used with TCP/IP transport protocols; but that is not the full truth.

The socket programming interface is not limited to TCP/IP. Sockets can also be used for interprocess communication within a computer without any network involvement or between computers using network protocols other than TCP/IP. Generally speaking, sockets can be used for interprocess communication using a whole range of protocol suites.

A socket is the endpoint of a communication path; it identifies the address of a specific process at a specific computer using a specific transport protocol. The exact syntax of a socket address depends on the protocol being used; on its addressing family. When you obtain a socket
via the `socket` system call, you pass a parameter that tells the socket library to which addressing family the socket should belong. All socket addresses within one addressing family use the same syntax to identify sockets; in other words, they belong to the same family.

In an MVS environment, you are able to use the following addressing families:

<table>
<thead>
<tr>
<th>Family</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AF_INET</td>
<td>Addressing family Internet - also referred to as the Internet domain.</td>
</tr>
</tbody>
</table>

This addressing family is used within the TCP/IP domain to identify sockets on IP hosts. A socket address in AF_INET consists of the following:

- **Family**: Half-word binary with a value of 2, which identifies the socket address as belonging to the AF_INET addressing family.
- **Port**: Half-word binary with port number (see "Ports" in topic 3.2.2) that identifies the process.
- **IP address**: Full-word binary with IP address of IP host in network byte order format.
- **Reserved**: 8 reserved bytes.

The following is an example of an AF_INET address that represents the telnet server (port number 23) on an IP host with the IP address of 9.24.104.74:

**AF_INET 23 9.24.104.74**

<table>
<thead>
<tr>
<th>Family</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AF_IUCV</td>
<td>Addressing family IUCV (Inter User Communication Vehicle).</td>
</tr>
</tbody>
</table>

This addressing family is unique to IBM TCP/IP for MVS and is only used within MVS. It can be used in C programs to implement a form of interprocess communication between processes in the same MVS system, or what is also termed as local sockets. You can use AF_INET for the same purpose, but AF_IUCV has some performance advantages over AF_INET, as AF_IUCV communication takes place directly between two MVS address spaces without involving the TCP/IP address space. In a UNIX or in an OpenEdition/MVS environment, you would use the AF_UNIX addressing family for the same purpose. The syntax of an AF_IUCV address is as the following:

- **Family**: Half-word binary with a value of 17, which identifies the socket address as belonging to the AF_IUCV addressing family.
- **Port**: Half-word binary. Reserved for future use. Must be set to zero.
- **Address**: Full-word binary. Reserved for future use. Must be set to zero.
- **Node ID**: 8 characters. Reserved for future use. Must be set to space.
User ID
8 characters set to the address space name of the application that binds the socket to a specific process.

Name
8 characters set to a name by which the process wants to be known to other processes within the AF_IUCV addressing family.

The following is an example of an IUCV address of a program called TESTPGM in the TESTAS MVS address space:

```
AF_IUCV 0 0 <space> TESTAS TESTPGM
```

AF_UNIX
Addressing family UNIX – also referred to as the UNIX domain.

This addressing family is not, as the name might suggest, restricted to UNIX environments. It just has its roots in the UNIX environment where it can be used for socket based interprocess communication between processes within one UNIX operating system. IBM TCP/IP for MVS does not support AF_UNIX sockets. You can use AF_UNIX with OpenEdition/MVS sockets, where this addressing family is used for interprocess communication between OpenEdition/MVS processes within one MVS operating system. The syntax of an AF_UNIX address is as the following:

```
Family
Half-word binary with a value of 1, which identifies the socket address as belonging to the AF_UNIX addressing family

Path
108 characters holding a pathname (similar to a hierarchical file system path name) by which this local process wants to be known by other local processes.
```

The following is an example of an address in the AF_UNIX addressing family:

```
AF_UNIX /u/xyz/testsrv
```

Other addressing families exist, and new families may be added in the future; but these three are the families you will meet in an MVS environment today. The two most important are the AF_INET and the AF_UNIX addressing families.

See Table 4 for an overview of which addressing families are supported by which socket library in MVS.

In the coming chapters, we will restrict our discussion to mostly TCP/IP sockets, which are sockets that belong to the AF_INET addressing family.

<table>
<thead>
<tr>
<th>Table 4. Addressing Families and Programming Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket Library Support</td>
</tr>
<tr>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>Open/MVS with integrated socket support (1)</td>
</tr>
</tbody>
</table>

A Beginner's Guide to MVS TCP/IP Socket Programming
### 3.6.1 Integrated Sockets

Integrated sockets is a concept introduced with OpenEdition/MVS in MVS/ESA SP 5.1, and it deserves some explanation in this context.

An OpenEdition/MVS application uses the same system calls, for example, to read and write data to and from local sockets and to and from files in the OpenEdition/MVS hierarchical file system. In an OpenEdition/MVS application a descriptor that is used in, for example, a `read` system call may either be a socket descriptor or a file descriptor. Descriptor four is, for example, a socket descriptor representing an AF_UNIX socket, and descriptor five might be a file descriptor used to read data from a file in the hierarchical file system. On, for example, a `read` system call an OpenEdition/MVS application will pass a descriptor. If the descriptor represents a socket, data will be read from the socket. If it represents a file, data will be read from the file.

In the MVS/ESA 4.3 OpenEdition/MVS implementation, descriptors had to be managed by different environments. The OpenEdition/MVS environment for file descriptors and local socket descriptors and the TCP/IP or AnyNet/MVS environment for network socket descriptors. This gave problems in the assignment and management of descriptor numbers across the involved environments and made it practically impossible to use both OpenEdition/MVS services and TCP/IP services from the same application program.

In the MVS/ESA SP 5.1 OpenEdition/MVS implementation, integrated socket support was introduced. This support creates an OpenEdition/MVS socket...
OpenEdition/MVS integrated sockets handles the following:

- Descriptor assignment and management
- Socket inheritance when an OpenEdition/MVS application uses the fork system call
- Select processing with a mix of socket, pipes and file descriptors

IBM TCP/IP for MVS handles the following:

- Management of TCP/IP protocols and the physical network connectivity
- Name translation by means of the Domain Name System
- TCP/IP applications like FTP, TELNET, etc.

In an OpenEdition/MVS system, you have a number of C-socket libraries you can choose among. See Figure 23 for an overview of some of your options in an MVS/ESA SP 5.1 OpenEdition/MVS system.
1 The choice between TCP/IP C-sockets, OpenEdition/MVS sockets or AnyNet/MVS sockets is made when you compile and link edit your C-socket program. This is a static choice. If you want to use the same source program with both AnyNet/MVS and TCP/IP, you must compile and link it into two separate load modules.

2 If you choose OpenEdition/MVS sockets for your compile and link job, you are able to use the same program with both TCP/IP and AnyNet/MVS as AF_INET provider, but not concurrently. The choice here is made when the OpenEdition/MVS kernel address space is started. In the OpenEdition/MVS start-up parameters, you specify if the AF_INET provider is TCP/IP or AnyNet/MVS, and that is in effect for all AF_INET communication from all OpenEdition/MVS socket programs until you restart the OpenEdition/MVS kernel address space. This description applies to OpenEdition/MVS as it is implemented in MVS/ESA SP 5.1. The support for AnyNet/MVS as OpenEdition/MVS AF_INET transport provider was added with PTF UW17057.
In the MVS/ESA SP 5.2.2 OpenEdition/MVS environment you will be able to use converged sockets. Converged sockets enable an OpenEdition/MVS socket program to use both TCP/IP and AnyNet/MVS concurrently as AF_INET transport provider. This support will, for example, enable an OpenEdition/MVS socket program to listen for TCP connections from both TCP/IP and AnyNet/MVS concurrently and to use a `select` call with a mix of file descriptors, local socket descriptors, TCP/IP network socket descriptors and AnyNet/MVS network socket descriptors.

### 3.7 General Socket Program Structure

The terms client and server are very common words within the TCP/IP community. More definitions of these terms exist. Often specific machines in a network are called servers. In this context, we talk about roles of communicating programs and more specifically about the distribution aspects of a cooperative application as discussed in Chapter 1, "Cooperative Applications" in topic 1.0.

In a TCP/IP context, the terms are defined as follows:

**Server**
A process that identifies itself to the network providing one or more specific services to clients. A server process responds to client requests.

**Client**
A process that initiates a request for some service from a server.

The client/server distribution model indicates a master/slave role; the client is the master requesting some service from the server (acting as the slave) that obediently responds to the requests from the client.

The model also implies a one-to-many relationship; a server typically serves multiple clients while a client deals with a single server.

No matter which of the socket programming interfaces you select, the functions you use will be the same. The syntax may vary, but the underlying concept is the same.

While clients communicate with one server at a time, servers may serve multiple clients. Consequently, when you design a server program, you may feel a need for multiple concurrent processes. Special socket calls are available for that purpose of concurrent servers, as opposed to the more simple type of iterative servers.

#### 3.7.1 Iterative Server

An iterative server processes requests from clients in a serial manner; one connection is served and responded to before the server accepts a new client connection.

---

Client process                ________________________________

Iterative Server
Figure 24. Iterative Server Main Logic

The iterative server waits for connection requests from the IP network.

1. When a connection request arrives, it accepts the connection, and 2 receives the client data.

The iterative server processes the received data and does whatever has to be done before it builds a reply, which is sent back to the client 3.

4. The iterative server closes the socket and waits again for the next connection request from the network.

An iterative server may be implemented in more ways in MVS as follows:

- As a batch job or MVS started task that is started manually or by automation software. The job will stay active until it is closed down by some operator intervention.

- As a TSO transaction. You may start your iterative server as a TSO transaction. For a production implementation, we recommend you do so by submitting a job that executes a batch Terminal Monitor Program (TMP).

- As a long-running CICS task. The task will normally be started during CICS start-up, but it may also be started by an authorized CICS operator that types in the appropriate CICS transaction code.

- As a Batch Message Program (BMP) in IMS.

From a socket programming perspective there is no difference between an iterative server that runs in a native MVS environment (batch job, started task or TSO) and one that runs as a CICS task or BMP under IMS.

A general concern for iterative servers is how to terminate the server process. For iterative servers, that execute in traditional MVS address spaces (batch job, started task, TSO, IMS BMP), you may implement functions in the server that enables an operator to use the MVS modify command to signal the iterative server to stop: **F SERVER,STOP**. This technique cannot be used for CICS tasks. Another solution to this problem is to include a shutdown message in the application protocol. By doing so, you can develop a shutdown client program that connects to the server and sends a shutdown message. When the server receives such a shutdown message from a socket client, it terminates itself.

### 3.7.2 Concurrent Server
A concurrent server accepts a client connection, delegates the connection to a child process of some kind, and immediately signals its willingness to receive the next client connection.

---

Figure 25. Concurrent Server Main Logic

1 When a connection request arrives in the main process of a concurrent server, it will schedule a child process and forward the connection 2 to the child process.

3 The child process takes the connection, which is given to it by the main process.

4 The child process receives the client request, processes it and sends back a reply 5 to the client.

6 The connection is closed, and the child process either terminates or signals to the main process that it is available for a new connection.

You may implement a concurrent server in the following MVS environments:

- Implement it in the native MVS environment (batch job, started task or TSO). In this environment you implement concurrency by using the traditional MVS subtasking facilities. These facilities are available from assembler language programs or from high-level languages that support multi-tasking or multi-threading, such as C/370.

- Implement it in a CICS environment, where the concurrent main process is started as a long-running CICS task that accepts connection requests from clients and initiates child processes via the EXEC CICS START command. CICS sockets includes a generic concurrent server main program called the CICS LISTENER.
Implement it in an IMS environment, where the concurrent main process is started as a BMP that accepts connection requests from clients and initiates child processes via the IMS message switch facilities. The child processes execute as IMS Message Processing Programs (MPP). IMS sockets include a generic concurrent server main program called the IMS LISTENER.

In both the iterative server and the concurrent server scenarios above, the client and server process could have exchanged a series of request/reply sequences before they decided to close down the connection. For the sake of simplicity, only a single interaction is shown in these diagrams.

3.7.3 Socket Program Categories

To distinguish between these generic program types, we will use the following terminology in the rest of this book:

Client programs for a socket program that acts as a client.

Iterative server programs for a socket program that acts as a server and that processes one client request fully before accepting a new client request.

Concurrent server main programs for that part of a concurrent server that manages child processes, accepts client connections and schedules client connections to child processes.

Concurrent server child programs for that part of a concurrent server that processes the client requests.

The term process is used for an instance of a program. In a concurrent server, the child program may be active in many parallel child processes, each processing a client request.

In an MVS environment, a process is either an MVS task, a CICS transaction, or an IMS transaction.

4.0 Chapter 4. The IBM TCP/IP for MVS Socket APIs

This chapter introduces each of the IBM TCP/IP for MVS socket APIs and gives specific usage guidelines for each API.

4.1 API Relationship
4.2 IBM TCP/IP for MVS C-Sockets
4.3 Sockets Extended Call Interface
4.4 Sockets Extended Assembler Macro Interface
4.5 REXX Sockets
4.6 Pascal API
4.7 Inter-User Communication Vehicle (IUCV) Sockets

4.1 API Relationship

Figure 26 shows how the different socket APIs are related to each other and to the TCP/IP protocol layers.
See Table 5 for an overview of the socket functions that are available in the most commonly used TCP/IP socket programming interfaces.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| htons             | Converts a host short integer to a
                        network short integer. |
| htonl             | Converts a host long integer to a
                        network long integer.   |
| givesocket        | Gets the socket number.          |
| getsockopt        | Gets the options of a socket.    |
| getsockname       | Gets the name of a socket.       |
| getpeername       | Gets the name of the connected
                        peer.                   |
| gethostent        | Gets the host entry.             |
| gethostname       | Gets the name of the host.       |
| gethostid         | Gets the host identification.    |
| getibmsockopt     | Gets the IBM socket options.     |
| gethostbyname     | Gets the name of a host.         |
| getprotobynumber  | Gets the number of a protocol.   |
| getprotobyname    | Gets the name of a protocol.     |
| getprotoent       | Gets the entry of a protocol.    |
| getservbyname     | Gets the name of a service.      |
| getservbyport     | Gets the name of a service by port. |
| getservent        | Gets the entry of a service.     |
| getservbyaddr     | Gets the name of a service by
                        address.               |
| getdomainname     | Gets the domain name.            |
| getclientid       | Gets the client identification.  |
| fcntl             | Gets the flags of a socket.      |
| inet_netof        | Gets the network offset.         |
| inet_lnaof        | Gets the local network address.  |
| inet_addr         | Gets the address of a socket.    |
| ibmflush          | Flushes the IBM socket.          |

This table lists various functions and definitions related to MVS TCP/IP socket programming.
4.2 IBM TCP/IP for MVS C-Sockets

The Berkeley socket programming interface is a C-based interface. To use it, you must develop your programs in the C programming language.

The interface is, to a large extent, compatible with the C socket interfaces available on many other system platforms. Like on most system platforms, you should observe some precautions if you port C-socket applications to MVS:

You must include an additional header file (tcperrno.h) if you want to reference all possible networking errors.

You should use the tcperror routine to print networking error messages. On other platforms, you would use the perror call instead of the tcperror call.

You must include the manifest.h header file, which is used to remap the socket function long names to 8-character names supported by MVS.

The functions ioctl, getsockopt, setsockopt and fnctl do not support all the BSD specified options.

The additional addressing family AF_IUCV is supported in C sockets. AF_IUCV allows MVS address spaces on the same host to communicate with each other using IUCV.

IBM TCP/IP for MVS C sockets use a number of socket calls that are specific to the IBM TCP/IP for MVS environment, for example, givesocket, takesocket and setibmsockopt.

Notes:

1. C-sockets as they can be used in the native MVS environment and in the IMS environment.

2. Sockets Extended including call interface and assembler macro interface. Call interface may be used in all environments including CICS and IMS, while assembler macro interface can be used in normal MVS address spaces (Batch, TSO or started task).

3. The C-socket function that can be used in the CICS environment.
For details please refer to IBM TCP/IP for MVS: Programmer’s Reference, SC31-7135.

You are able to use the C/370 multitasking facilities to create C based multitasking concurrent server applications.

We created and tested small C socket applications using the IBM C/370 Compiler Version 2 Release 1 (5688-187).

IBM TCP/IP for MVS C socket programs may be developed for both the CICS and IMS environment. Please note that not all C-socket functions are available in the CICS C-socket implementation.

The C header files are distributed in the tcpip.v3r1.SEZACMAC library, which you must concatenate to your C compiler SYSLIB DD statement. The runtime library routines are distributed in the tcpip.v3r1.SEZACMTX library, which you must concatenate to your MVS Binder SYSLIB DD statement.


4.3 Sockets Extended Call Interface

The Sockets Extended call interface supports all the basic socket functions, but it does not support all the extra data conversion or IP address manipulation functions you find in the C-sockets implementation.

Some of the extra functions are implemented in a set of EZACICxx routines; they are as follows:

- **EZACIC04**: Translate a character string from EBCDIC to ASCII.
- **EZACIC05**: Translate a character string from ASCII to EBCDIC.
- **EZACIC06**: Translate between a character array and a bit string. You can use this routine, for example, in a COBOL program to manipulate bit strings in a select mask.
- **EZACIC08**: Parse the contents of a host entry structure returned by a `gethostbyname` or `gethostbyaddr` call.

All the Sockets Extended calls use a return code parameter (RETCODE) to pass back a return code from the function you called. Most of the calls, in addition to the return code parameter, also use an error number parameter (ERRNO), which is used to pass back the specific error code that applies to the error situation that was the result of the call. If the return code parameter has been set to a negative value, the error number parameter holds the specific error code, which corresponds to the value returned to a C program on a `tcperror` function call.

It is good programming practice to include logic after each call, which tests the return code and, in case it is negative, formats an error message based on the value passed back in the error number parameter. During initial development and testing of a socket program, such a practice will prove to be very valuable.
When you call the Sockets Extended interface, you always pass a string of 16 characters as the first parameter holding the function code you want to use. We recommend you define these function code strings once and put them into a copy structure or include member.

*---------------------------------------------------------------*
* Socket interface function codes                             *
*---------------------------------------------------------------*
01 soket-functions.
  02 soket-accept pic x(16) value 'ACCEPT'.
  02 soket-bind pic x(16) value 'BIND'.
  02 soket-close pic x(16) value 'CLOSE'.
  02 soket-connect pic x(16) value 'CONNECT'.
  02 soket-fcntl pic x(16) value 'FCNTL'.
  02 soket-getclientid pic x(16) value 'GETCLIENTID'.
  02 soket-gethostbyaddr pic x(16) value 'GETHOSTBYADDR'.
  02 soket-gethostbyname pic x(16) value 'GETHOSTBYNAME'.
  02 soket-gethostid pic x(16) value 'GETHOSTID'.
  02 soket-gethostname pic x(16) value 'GETHOSTNAME'.
  02 soket-getpeername pic x(16) value 'GETPEERNAME'.
  02 soket-getsockname pic x(16) value 'GETSOCKNAME'.
  02 soket-givesocket pic x(16) value 'GIVESOCKET'.
  02 soket-initapi pic x(16) value 'INITAPI'.
  02 soket-ioctl pic x(16) value 'IOCTL'.
  02 soket-listen pic x(16) value 'LISTEN'.
  02 soket-read pic x(16) value 'READ'.
  02 soket-recv pic x(16) value 'RECV'.
  02 soket-recvfrom pic x(16) value 'RECVFROM'.
  02 soket-select pic x(16) value 'SELECT'.
  02 soket-send pic x(16) value 'SEND'.
  02 soket-sendto pic x(16) value 'SENDTO'.
  02 soket-setsockopt pic x(16) value 'SETSOCKOPT'.
  02 soket-shutdown pic x(16) value 'SHUTDOWN'.
  02 soket-socket pic x(16) value 'SOCKET'.
  02 soket-takesocket pic x(16) value 'TAKE_SOCKET'.
  02 soket-termapi pic x(16) value 'TERMAPI'.
  02 soket-write pic x(16) value 'WRITE'.

The function code must be in uppercase.

The Sockets Extended call interface does not support MVS multitasking, which means you cannot use it to develop concurrent servers that are implemented in a single MVS address space. Concurrent server implementations based on the IMS or the CICS listener, where your Sockets Extended call-based program is the child process, started either in an IMS Message Processing Region (MPR) or as a started transaction in CICS, is fully supported.

When you bind your program with the MVS binder, you must concatenate the tcpip.v3r1.SEZATCP library to your SYSLIB DD statement.

```
//SYSLIB  DD DSN=.....
//            DD DSN=TCPIP.V3R1.SEZATCP,DISP=SHR
//            DD DSN=.....
```

See "COBOL Compile JCL Procedure" in topic I.2 for a sample COBOL compile procedure and "Link/Edit JCL Procedure" in topic I.4 for a sample MVS binder procedure.

4.3.1 PL/I Programs
4.3.2 User Abend 4093

4.3.1 PL/I Programs

If you use the IBM PL/I Optimizing Compiler Version 2 (5668-910), you have to declare the Sockets Extended interface routine with:

```
DCL EZASOKET ENTRY OPTIONS(RETCODE,ASM,INTER) EXT;
```

This causes the compiler to print the following warning message:

```
IEL0983I EXTERNAL NAME 'EZASOKET' EXCEEDS 7 CHARACTERS.
EXECUTION IS UNDEFINED IF 'EZASOKET' IS THE SAME AS A COMPILER GENERATED NAME.
```

We did not experience any difficulties by ignoring this message.

When you use the new AD/CYCLE PLI compiler (5688-235) you may code:

```
DCL MYSOKET ENTRY OPTIONS(RETCODE,ASM,INTER) EXT('EZASOKET');
```

which also requires all other references to EZASOKET in the program to be replaced by references to MYSOKET. In this case you do not get a warning message. Effectively, however, the situation is not fundamentally different from the situation with PL/I Version 2.

If one is really in doubt, the interface routine may be re-link-edited under a different name:

```
//jobname JOB ............
//LKED EXEC LKED
//SYSLMOD DD DISP=SHR,DSN=load module library - for PL/I link-edit
//TCPLIB DD DISP=SHR,DSN=hlq.SEZATCP
//SYSIN DD *
CHANGE EZASOKET(MYSOKET)
INCLUDE TCPLIB(EZASOKET)
ENTRY MYSOKET
NAME MYSOKET(R)
```

4.3.2 User Abend 4093

When you test Sockets Extended programs, you may encounter user abend code 4093.

This does not indicate a system problem, but it signals a syntax error in the parameters passed to the EZASOKET call, such as a wrong number of parameters or an invalid function code.

4.4 Sockets Extended Assembler Macro Interface

The socket functions available in the Sockets Extended macro-based interface are similar to those you find in the call-based interface.

You do have some extra facilities in the macro-based interface, which are mainly the following:

1. Support for multitasking environments
2. Support for asynchronous socket calls

3. Support for API type three (APITYPE-3) programs

From your macro-based assembler socket programs you can use the same EZACICxx routines as we mentioned earlier (see "Sockets Extended Call Interface" in topic 4.3), or you can implement similar functions in your assembler language environment.

![Diagram of sockets extended macro interface storage areas]

**Figure 27. Sockets Extended Macro Interface Storage Areas**

When you use the Sockets Extended macro interface you must provide the following couple of storage areas that are used by the socket macros:

1. A global storage area

   This storage area must exist once for an address space. The storage area can be allocated by the single task of your single-tasking application or the main task of your multitasking application.

   The storage area must be accessible from all program modules that issue socket macro calls inside your address space. The module that allocates this storage area must pass a pointer to the area to all modules inside the address space that uses socket calls.

2. A task storage area
If you develop a single-tasking socket application, you allocate one task storage area.

If you develop a multitasking socket application, you must allocate one task storage area per task.

The task storage area must be accessible from all program modules that issue socket calls in a task.

In the example illustrated in Figure 27, you will find the following four Sockets Extended storage areas:

1. The global storage area (EZAGLOB), which is allocated in the main task. A pointer to the global storage area is passed to each subtask in the PARAM keyword on the ATTACH macro call. Each of the subtasks establishes a DSECT for the global work area and sets up a base register for it.
2. The task storage area for the main task (EZAMTASK), which is only accessible to the main task.
3. The task storage area for subtask 1 (EZATASK1), which is only accessible to subtask 1.
4. The task storage area for subtask 2 (EZATASK2), which is only accessible to subtask 2.

If you use the asynchronous option on the socket macro calls by means of the ECB= keyword, you must remember to issue a socket sync macro call after the ECB associated with the asynchronous call has been posted. Returned information from the asynchronous socket call is not placed into your program variables until you issue the sync macro call. Please note that the ECB= keyword must point to an Event Control Block (ECB) followed by a 100 byte work area, which the socket macro interface will use to store temporary status information in.

When you assemble your Sockets Extended macro programs, you must include the tcpip.v3r1.SEZACMAC library in your assembler SYSLIB concatenation. See "Assemble JCL Procedure" in topic I.1 for a sample assemble procedure.

When you bind your program with the MVS Binder, you must include the tcpip.v3r1.SEZATCP library in your binder SYSLIB concatenation. See "Link/Edit JCL Procedure" in topic I.4 for a sample MVS binder procedure.

4.5 REXX Sockets

The REXX language is well suited for the development of prototypes. As REXX is an interpreted language, no time is lost by compilations. This is very useful if you want to test a number of alternatives; once you have saved a REXX procedure, you can run it. Also, REXX can be used for production applications as long as the performance requirements are within certain limits.

Coding REXX procedures for TCP/IP for MVS is pretty straightforward.

The only requirement for using the REXX socket interface is that you have access to the tcpip.v3r1.SEZALINK library. This library will normally be accessible through your system LINKLIST concatenation. If it is not, you will have to concatenate it to your TSO STEPLIB allocation.

Some hints may be helpful:

Before using any other call, you have to identify the TCP/IP system
you are using with an initialize call.

In this call you specify the following:

- The jobname of the TCP/IP system address space.

- The name of your so-called socket set. This name can be anything as long as it is used consistently within your job.

- Optionally, you can specify the number of sockets you would like to have preallocated, if the default of 40 is not appropriate.

Socketsets can be reused and should eventually be terminated. There is an option to inquire if there are presently any available socket set(s). If you re-initialize a socketset that was not closed, you will get an error.

If you run two REXX socket programs in each session of an ISPF split environment, the two REXX programs must use different socketset names.

All TCP/IP for MVS REXX socket calls are implemented as REXX functions that return a string that contains the return code as the first token. (The standard REXX rc return code variable is not used, so a signal on error statement does not cause the socket call errors to be caught.)

- If the call completed successfully, the return code is zero and the remainder of the returned string may contain other information, as defined by the called function.

- If the call did not complete successfully, an error message is passed after the return code.

You can handle TCP/IP for MVS REXX return codes as follows:

```
parse value socket(function,other parameters) with rc rest
if rc=0 then parse value rest call related return values ...
else say 'Error, reason:' rest
```

If you are use to writing REXX procedures on OS/2 and/or VM/CMS but not on MVS, you should be aware that TSO/E requires you to start your procedure explicitly with /* REXX ...*/. If you omit the word REXX, it will not work.

Please see Appendix E, "Sample REXX Socket Programs" in topic E.0 for a sample REXX server and client program.

### 4.6 Pascal API

The Pascal programming interface is based on Pascal procedures and functions that implement conceptually the same functions as the C socket interface. The routines have different names though, so in practical terms there is a considerable difference with the C socket calls.

You can use stream sockets, datagram sockets or raw sockets.

If you are a skilled Pascal programmer, you should be able to develop socket programs relatively easily using this Pascal programming interface.

To compile a Pascal program, you need the IBM Pascal compiler and library (5668-767).
The include files you will need are in the tcpip.v3r1.SEZACMAC library, which you will have to concatenate to the SYSLIB DD statement of your Pascal compile JCL. The library routines are in the tcpip.v3r1.SEZACMTX library, which you will have to concatenate to the SYSLIB DD statement of your linkage editor.

4.7 Inter-User Communication Vehicle (IUCV) Sockets

The IUCV socket programming interface is language independent. It is based on standard linkage calls for transferring data or control to IUCV and on asynchronous exits implemented via IRBs (Interrupt Request Blocks) for receiving data from IUCV. The programming interface is provided as assembler macros.

The IUCV programming interface is only provided with IBM TCP/IP Version 3 Release 1 for MVS for reason of compatibility with IBM TCP/IP Version 2 Release 2 for MVS, and we will not describe it in any further detail in this book.

--- Recommendation

We will recommend that, wherever it makes sense, you use the Sockets Extended programming interfaces instead of the IUCV interface.

---

5.0 Chapter 5. Your First Socket Program

In this chapter we will guide you through the development of a simple stream socket program. The program's purpose is to act as an iterative echo server program. An echo server just returns any data it receives to the client.

In this chapter we will explain all the basics of the individual socket calls, the data structures that are used with the socket calls, and the programming techniques associated with some of the more complicated socket calls, and we will discuss some general security aspects of socket programs.

See "Sample Stream Socket COBOL Server" in topic B.1 for the full sample server code, and "Sample Stream Socket COBOL Client" in topic B.2 for a sample client that can be used to test the server.

The description will be fairly elaborate as this is the first time you are facing actual socket programming. In the succeeding chapters, where we look more into the CICS and IMS socket environments, we will not go into the same kind of detail, but rather we will refer back to the examples you find in this chapter. So even if your purpose for reading this book is to develop IMS or CICS socket programs, we recommend you read this chapter before continuing with the IMS and CICS chapters.

The coding examples in this chapter will be in COBOL and based on the Sockets Extended call interface.

5.1 Type Conversion Between Programming Languages
5.2 Iterative Server Program Structure
5.3 Initialize the Socket API
5.4 Create a Socket
5.5 Bind a Socket to a Specific Port Number
5.6 Listen for Client Connection Requests
5.7 Accepting Connection Requests from Clients
5.8 Transferring Data Over a Stream Socket
5.9 Closing a Connection
5.10 Blocking, Non-blocking and Asynchronous Socket Calls
5.11 Socket Programs and MVS Security

5.1 Type Conversion Between Programming Languages

We will not show you the samples in all available programming languages. We have, for the major part of our samples, chosen COBOL because of both its widespread use and readability. Most readers, who are not familiar with COBOL, may be able to read the COBOL samples as pseudo-code. To enable you to convert the variable declarations, we have included a short type-conversion table (please see Table 6).

<table>
<thead>
<tr>
<th>Generic type</th>
<th>assembler</th>
<th>COBOL</th>
<th>PL/I</th>
<th>C/370</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 byte binary integer</td>
<td>name DC F'0'</td>
<td>name pic S9(8) binary</td>
<td>dcl name fixed(31)</td>
<td>long</td>
</tr>
<tr>
<td></td>
<td>value zero.</td>
<td>value zero.</td>
<td>binary init(0);</td>
<td></td>
</tr>
<tr>
<td>2 byte binary integer</td>
<td>name DC H'0'</td>
<td>name pic S9(4) binary</td>
<td>dcl name fixed(15)</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>value zero.</td>
<td>value zero.</td>
<td>binary init(0);</td>
<td></td>
</tr>
<tr>
<td>String of n bytes</td>
<td>name DC CLn'text'</td>
<td>name pic X(n) value</td>
<td>dcl name char(n)</td>
<td>char</td>
</tr>
<tr>
<td></td>
<td>'text'.</td>
<td>'text'.</td>
<td>init('text');</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
name denotes identifier name

Table 6. Language Type Definition Conversion

5.2 Iterative Server Program Structure

The reason we start with an iterative server is because it is quite simple to implement, and it allows us to introduce all the major basic socket calls with which you will have to work.

If each connection from a client is of a short duration, you may implement your server as an iterative server.

A typical scenario for an iterative server is that the client and the iterative server exchange a single request/reply sequence per connection.

If the lifetime of a connection is of a longer duration, involving a sequence of request/reply interactions possibly with user think-time involved, you should consider implementing your server as a concurrent server.
A Beginner’s Guide to MVS TCP/IP Socket Programming

<table>
<thead>
<tr>
<th>Initapi</th>
<th>Issue passive open 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obtain socket</td>
<td>5</td>
</tr>
<tr>
<td>Connect to server</td>
<td>Accept a connection request</td>
</tr>
<tr>
<td>Send data</td>
<td>Receive client data 6</td>
</tr>
<tr>
<td>Receive reply</td>
<td>Send reply to client 7</td>
</tr>
<tr>
<td>Close connection</td>
<td>Close connection</td>
</tr>
<tr>
<td>Termaapi</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 28. Iterative Server Main Logic

The sequence numbers in the following text are related to the corresponding numbers in Figure 28.

From a socket interface point of view, there is no difference between an iterative server that is implemented as a native MVS program, as a CICS transaction program or an IMS transaction program. The only difference is the way the iterative server program is started.

MVS  
You can start the server as a batch job, as a started task, as a TSO transaction or as an APPC/MVS transaction.

IMS  
You can start your iterative server as a Batch Message Program (BMP) or as a long-running Message Processing Program (MPP).

CICS  
You can start the server as a long-running CICS task.

5.3 Initialize the Socket API

For the Sockets Extended programming interface, the first call is an initapi call, where you identify your own process to the TCP/IP system address space.

```plaintext
* Variables used for the INITAPI call *
01 soket-initapi pic x(16) value 'INITAPI '.
01 maxsoc pic 9(4) Binary Value 50.
01 initapi-ident.
  05 tcpname pic x(8) Value 'T18ATCP'.
  05 myasname pic x(8) Value space.
01 subtask pic x(8) Value space.
01 maxsno pic 9(8) Binary Value zero.
01 erorno pic 9(8) Binary Value zero.
01 retcode pic s9(8) Binary Value zero.

* Initialize socket API *
Call 'EZASOKET' using soket-initapi
  maxsoc
  initapi-ident
  subtask
  maxsno
  erorno
  retcode.
```
If retcode < 0 then
  - process error -

You use the initapi call to both establish a communication path between your program and a TCP/IP address space, and identify your program to that TCP/IP address space.

If you have both a test and a production TCP/IP system executing in your MVS environment, you can control which system you are using via the TCPNAME parameter. You must initialize this parameter with the correct address space name of your TCP/IP system address space. The Sockets Extended programming interface does not pick up any default value from the tcpip.v3r1.TCPIP.DATA configuration data set.

Your MVS system

<table>
<thead>
<tr>
<th>TCP/PROD</th>
<th>TCP/TEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP</td>
<td>TCP/IP</td>
</tr>
<tr>
<td>_____</td>
<td><strong><strong>?</strong></strong></td>
</tr>
</tbody>
</table>

Figure 29. Identifying Your TCP/IP Address Space via TCPNAME

If you receive an error number of 10191 (IUCV returned an error code) during initapi processing, the most likely reason is that you did not specify a TCPNAME value that matches the name of any currently active TCP/IP system address space.

Ask your MVS TCP/IP systems programmer for the name of the TCP/IP system address space. You may want to implement logic that allows you to pass the name via, for example, either the PARM field on the EXEC JCL statement or a parameter SYSIN file that your program reads before it issues the initapi call. If you implement such logic, your operations department is able to change the TCP/IP setup without having to ask you to modify your program.

In order to identify your program, you use the fields MYASNAME and SUBTASK.
You can use any values you find relevant, but the combination of address space name and subtask ID must be different from the values used by any other program that connects to the same TCP/IP system address space.

For a single-threaded program, you can pass MYASNAME and SUBTASK initialized to space (X'40'). The Sockets Extended programming interface will pick up your correct address space name. However, your subtask ID will be 8 blanks. That will work for a single-threaded socket program, like our sample iterative echo server, but it will not work for multi-threaded programs, as each subtask must have a unique subtask ID. See "TPICLNID Obtain Values for TCP/IP Client ID" in topic G.1 for a sample subroutine that will return the current address space name and Task Control Block (TCB) address as two 8-byte character fields that you can use as MYASNAME and SUBTASK from both single-threaded and multi-threaded socket programs in a native MVS environment.

Note: In a CICS environment the subtask ID has a slightly different meaning. In CICS, all programs run under one MVS task control block. This is also the case for a concurrent server implementation. You can, instead of the TCB address, use the internal CICS task number as the source for your subtask ID. In a CICS program, you can find your current CICS task number by picking up the EIBTASKN field in the CICS command level interface block (EIB). Convert the task number to an EBCDIC representation and make that part of your subtask ID. If your current CICS task number is, for example, 129, you can use a subtask ID of 00000129; or, to identify it uniquely as a CICS task number and not a TCB address, you may prefix or suffix it by a non-hexadecimal character: 0000129T.

We recommend that you always pass a subtask ID on the initapi call in a CICS program. If you have two CICS socket programs that start at the same time and they both use a subtask ID of space, they will both specify the same client ID, as they both execute in the same address space.

MYASNAME and SUBTASK are used to complete a structure that is called the client ID structure. The client ID is specific to the IBM TCP/IP for MVS.
implementation. It is the identifier by which a process is known to the TCP/IP address space in MVS, hence the term client ID, which must be interpreted as the client of the TCP/IP system address space. Do not confuse this client ID with the client/server role of your application programs. From the TCP/IP system address space point of view, every application program in your MVS system that uses TCP/IP facilities is a client of the TCP/IP system address space. This client ID has actually nothing to do with the underlying protocols (TCP, UDP or IP). It is never exchanged over the IP network. The client ID is unique for the IBM TCP/IP for MVS socket interfaces. It is not part of the original BSD specifications.

<table>
<thead>
<tr>
<th>0</th>
<th>Domain</th>
<th>4 bytes binary AF_INET = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>+4</td>
<td>Address</td>
<td>8 bytes character jobname/address space name</td>
</tr>
<tr>
<td></td>
<td>Space name</td>
<td></td>
</tr>
<tr>
<td>+12</td>
<td>Subtask ID</td>
<td>8 bytes character subtask/task ID</td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
<td>20 bytes reserved (must be binary zeroes)</td>
</tr>
<tr>
<td>+20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+40</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 31. The Client ID Structure

Note: Please observe that the domain field in the client ID structure is a 4-byte field and not a 2-byte field.

The MAXSOC parameter on the initapi call is used to reserve storage for the maximum number of sockets this program intends to work with concurrently. The default value is 50 and the maximum in IBM TCP/IP Version 3 Release 1 for MVS is 2000.

The MAXSNO parameter is an output parameter from the initapi call. When the call returns to your program, it contains the highest socket descriptor number that can be assigned to your program.

5.3.1 Initializing a C-socket Program

5.3.2 Getclientid

5.3.1 Initializing a C-socket Program

In the C-socket API, you do not have an initapi function.

When the C-socket API processes the first valid socket call in a C-program, it performs a function that is equivalent to the initapi call. The C-socket API locates the TCP/IP address space to connect to via the tcpip.v3r1.TCPIP.DATA configuration data set. You can override the installation default by allocating the TCPIP.DATA data set of a test TCP/IP system to a DD name of SYSTCPD in the address space in which your C program executes. A socket call return code of EIBMIUCVERR accompanied by CONNECT error messages with a return code of 1011 usually means that you try to use a TCP/IP system address space that is not currently active on your MVS system.
If you use C-sockets, you have no influence on the content of your client ID. The C socket library routines sets it to the name of your address space and an EBCDIC representation of a storage address, which meaning is known to the C runtime environment.

A C-socket program may use the `maxdesc` socket call to increase the number of available socket descriptors.

### 5.3.2 Getclientid

You can, in all environments, retrieve your client ID by using the `getclientid` call. This call will return a client ID structure with the current client ID of the calling process.

```assembly
01 soket-getclientid pic x(16) value 'GETCLIENTID'.
01 clientid.
  05 clientid-domain pic 9(8) Binary.
  05 clientid-name pic x(8) value space.
  05 clientid-task pic x(8) value space.
  05 filler pic x(20) value low-value.
01 errno pic 9(8) Binary value zero.
01 retcode pic s9(8) Binary value zero.
```

* Variables used by the GETCLIENTID Call *

When you write iterative server programs, you are normally not concerned with the client ID, but if you write concurrent server programs, you use client IDs to give sockets to and take sockets from.

The `getclientid` call is not part of the original BSD socket implementation. If you use it in C-programs, you must consider portability issues if you want to be able to port your C-program to other operating system platforms.

### 5.4 Create a Socket

The server obtains a socket via the `socket` call. You must specify to what domain the socket belongs and what type of socket you want.

```assembly
01 soket-socket pic x(16) value 'SOCKET'.
01 afinet pic 9(8) Binary Value 2.
01 soctype-stream pic 9(8) Binary Value 1.
01 proto pic 9(8) Binary Value zero.
01 socket-descriptor pic 9(4) Binary Value zero.
```

* Variables used for the SOCKET call *

A Beginner's Guide to MVS TCP/IP Socket Programming
The internet domain has a value of two. A stream socket is requested by passing a value of one as type. The proto field is normally zero, which means that the socket API should choose the protocol to use for the domain and socket type requested. In this example the socket will use TCP protocols.

A socket descriptor representing an unnamed socket is returned from the socket call. An unnamed socket has no port and no IP address information associated; only the protocol information is available. The socket descriptor is a 2-byte binary field and must be passed on subsequent socket calls as such.

A socket is an unhandy concept for a program to work with because it consists of three different things: a protocol specification, a port number and an IP address. To represent the socket in a more handy way, we use the socket descriptor.

The socket descriptor is not in itself a socket, but it represents a socket and is used by the socket library routines as an index into a table of sockets owned by a given MVS TCP/IP client (represented by a client ID: address space name and task ID). On all socket calls that reference a specific socket, you must pass the socket descriptor that represents the socket with which you want to work.

<table>
<thead>
<tr>
<th>Socket Descriptor</th>
<th>Socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Our Listen socket</td>
</tr>
<tr>
<td>1</td>
<td>Our connected socket</td>
</tr>
</tbody>
</table>

Figure 32. MVS TCP/IP Socket Descriptor Table

The first socket descriptor your program is assigned is zero for a Sockets Extended program. If your program is a C-program, socket descriptors zero, one and two are reserved for std.in, std.out and std.err, and the first socket descriptor that is assigned for your AF_INET sockets is three or higher.

When a socket is closed, the socket descriptor becomes available and will be returned as a new socket descriptor representing a new socket at a succeeding request for a socket.
In the reference documentation, the socket descriptor is normally represented by a single letter: S or as two letters: SD.

When you have the socket descriptor, you can request the socket address structure from the socket programming interface via the getsockname call. Remember that a socket is not fully named (including both port and IP address) until after a successful bind, connect, or accept call.

If your socket program is capable of handling more sockets simultaneously, you must keep track of your socket descriptors. A good idea is to build a socket descriptor table inside your program where you store information related to the socket and the status of the socket. You will need this information if, for example, you use the select call. Besides that purpose, it is good to have in debugging situations.

### 5.5 Bind a Socket to a Specific Port Number

3 in Figure 28 in topic 5.2. The socket is bound to a specific port number via the bind call. By binding the socket to a specific port number, you avoid having an ephemeral port number assigned to the socket.

For servers it would be rather inconvenient to have an ephemeral port number assigned, because clients would have to connect to different port numbers for every instance of the server. By using a predefined port number, clients can be developed so they always connect to the same port number.

Client programs may also use the bind socket call, but normally client programs do not benefit from using the same port number every time they execute.

```plaintext
*---------------------------------------------------------------*
* Variables used for the BIND Call                             *
*---------------------------------------------------------------*
 01 socket-bind pic x(16) value 'BIND '.
 01 server-socket-address.
    05 server-afinet pic 9(4) Binary Value 2.
    05 server-port pic 9(4) Binary Value 9998.
    05 server-ipaddr pic 9(8) Binary Value zero.
    05 filler pic x(8) value low-value.
 01 socket-descriptor pic 9(4) Binary.
 01 errno pic 9(8) Binary Value zero.
 01 retcode pic s9(8) Binary Value zero.

*---------------------------------------------------------------*
* Bind socket to our server port number                        *
*---------------------------------------------------------------*
Call 'EZASOKET' using socket-bind
  socket-descriptor
  server-socket-address
  errno
  retcode.
If retcode < 0 then
  - process error -
```

Before you issue this call, you must build a socket address structure for your own socket with the following information:

1. Addressing family = two, which means: AF_INET.
2. Port number for your server application. For a Sockets Extended program, you will have to use a predefined port number, which is either a constant in your program or is passed to your program as an initialization parameter. If you develop your socket program in C, you can issue a `getservbyname` call to find the port number that is reserved for your server application in the `tcplib.v3r1.ETC.SERVICES` data set.

3. IP address on which your server application will accept incoming requests. If your application is executing on a multihomed host and you want to accept incoming requests over all available network interfaces, you must set this field to binary zeroes, which means: `INADDR_ANY`.

Until this point, you have not told TCP/IP anything about the purpose of the socket you obtained. You can use it as a client to issue connect requests to servers in the IP network, or you can use it to become a server yourself.

In socket terms, the socket at the moment is an active socket, which is the default status for a newly created socket.

### 5.6 Listen for Client Connection Requests

4. In Figure 28 in topic 5.2. When you call `listen`, you inform TCP/IP that you intend to be a server that will accept incoming requests from the IP network. By doing so, the socket status is changed from its default active status to a passive socket.

A passive socket does not take the initiative to initiate a connection; it just waits passively for clients to connect to it.

```
*---------------------------------------------------------------*
* Variables used by the LISTEN Call                              *
*---------------------------------------------------------------*
01 soket-listen  pic x(16) value 'LISTEN '.
01 backlog-queue pic 9(8) Binary Value 10.
01 socket-descriptor pic 9(4) Binary.
01 errno      pic 9(8) Binary Value zero.
01 retcode    pic s9(8) Binary Value zero.

*---------------------------------------------------------------*
* Issue passive open via Listen call                            *
*---------------------------------------------------------------*
Call 'EZASOKET' using soket-listen
  socket-descriptor
  backlog-queue
  errno
  retcode.
If retcode < 0 then
  - process error -

The backlog queue value is used by the TCP/IP address space when a connect request arrives and your server program is already connected to another client and is busy serving that client's request. TCP/IP will queue new connection requests up to the number you specify in the backlog queue parameter. If further connection requests arrive, they will be rejected by TCP/IP. You cannot in a C program specify a backlog value that exceeds the value assigned to SOMAXCONN in the socket header file supplied in `tcplib.v3r1.SEZACMAC`. The current implementation of IBM TCP/IP for MVS...
uses a value of 10. Most UNIX systems use a default value of 5 for the backlog queue.

The listen call does not establish any connections; it just turns the socket into a passive state, so it is prepared for connection requests from the IP network. If a connection request for this server arrives between the time of the listen call and the succeeding accept call, it will be queued according to the backlog value passed on the listen call.

5.7 Accepting Connection Requests from Clients

The listen call does not establish any connections; it just turns the socket into a passive state, so it is prepared for connection requests from the IP network. If a connection request for this server arrives between the time of the listen call and the succeeding accept call, it will be queued according to the backlog value passed on the listen call.

5.7 Accepting Connection Requests from Clients

5 in Figure 28 in topic 5.2. The accept call dequeues the first queued connection request or blocks the caller until a connection request arrives over the IP network.

*---------------------------------------------------------------*
* Variables used by the ACCEPT Call                            *
*---------------------------------------------------------------*
01 soket-accept       pic x(16) value 'ACCEPT            
01 client-socket-address.
   05 client-afinet  pic 9(4) Binary Value zero.
   05 client-port    pic 9(4) Binary Value zero.
   05 client-ipaddr  pic 9(8) Binary Value zero.
   05 filler         pic x(8) value low-value.
01 accepted-socket-descriptor pic 9(4) Binary Value zero.
01 socket-descriptor   pic 9(4) Binary.
01 errno             pic 9(8) Binary Value zero.
01 retcode           pic s9(8) Binary Value zero.
*---------------------------------------------------------------*
* Start iterative server loop with a blocking Accept Call     *
*---------------------------------------------------------------*
Call 'EZASOKET' using soket-accept
   socket-descriptor
   client-socket-address
   errno
   retcode.
If retcode < 0 then
   - process error -
else
   Move retcode to accepted-socket-descriptor.

This call works with two socket descriptors:

1. The first socket descriptor is representing the socket that was obtained, bound to the server port and optionally IP address, and turned into a passive state via the listen call.

2. The accept call will return a new socket descriptor, which will represent a complete association:

   Accepted_socket_descriptor represents:
   
   {TCP, server IP address, server port, client IP address, client port}

The original socket, which was passed to the accept call, is unchanged and is still representing only our server half association:

   Original_socket_descriptor represents:
   
   {TCP, server IP address, server port}
When control returns to your program, the socket address structure passed on the call has been filled with the socket address information of the connecting client.

The succeeding socket calls for the exchange of data between the client and the server will use the new socket descriptor. The original socket descriptor will remain unused until the iterative server has finished processing the client request, and it has closed the new socket. The iterative server will then reissue the accept call using the original socket descriptor and wait for a new connection.

5.8 Transferring Data Over a Stream Socket

6 and 7 in Figure 28 in topic 5.2. The stream concept implies two continuous streams of bytes flowing independent of each other in opposite directions.

In the case of stream sockets there is no one-to-one correspondence between send calls on one side and receive calls on the other side. Data, for example, that is sent by a single send call may have to be retrieved by a number of successive recv calls. The other way around may be equally likely.

The TCP protocol layer does not know anything about application-related boundaries on the stream; it is unaware of application records. As a consequence of this, part of your application design must be to develop a message design that your two application partners can use to determine when to stop issuing receive calls, when to start processing, and when to send something back. This design is important because, if two applications wait for each other to send data on the stream, they can wait forever.

The socket APIs do provide a technique to determine if any data on the stream is ready to be received. This is done via an ioctl socket call with a command of FIONREAD. The number of bytes that are currently available to be read from the stream is returned in the RETARG parameter as a binary full-word.

You can use the ioctl call to learn how many bytes are currently available to be read and then issue a recv call for that amount of bytes. But, if the message you expect to receive is longer than the available bytes, you have to wait a short amount of time and then repeat the process until you have the full message. This technique allows you to detect a faulty partner program that does not send a full message. You can implement timeout logic that determines the partner program is in error if you have not received a full message within a predefined amount of time.

---

**Note**

It is important to note that TCP leaves the design of application records and application protocols entirely to the application developer.

---

5.8.1 Streams and Messages
5.8.2 Reading and Writing Data From and To a Socket
5.8.3 Data Representation
5.8.1 Streams and Messages

How do you design an application protocol so that the partner program is able to chop the receive stream into individual messages?

Some socket applications are so simple that the receiver can just go on receiving data until the sender closes the socket. This might be the case for a simple file transfer application. Most applications are not that simple and usually require that the stream can be divided into a number of distinct messages.

A message exchanged between two socket programs must imbed information so that the receiver is able to decide how many bytes to expect from the sender and optionally what to do with the received message. The last aspect may not be important for some applications if the function of the application is so limited that all messages are treated in the same way; however, the first aspect is important to all applications.

There are a couple of commonly used techniques to imbed information about the length of a message into the stream as follows:

1. The message type identifier technique

If your messages are fixed length messages, you can implement a message ID per message type you work with. Each message type has a predefined length that is known by your client and server program. If you place the message ID in the start of each message, the receiving program is able to decide how long the message is (if it knows the content of the first couple of bytes in the message).

```
* Layout of a message between TPI client and TPI server *
*---------------------------------------------------------------*
01 tpi-message.
  05 tpi-message-id pic x.
  88 tpi-request-add value '1'.
  88 tpi-request-update value '2'.
  88 tpi-request-query value '3'.
  88 tpi-request-delete value '4'.
  88 tpi-query-reply value 'A'.
  88 tpi-response value 'B'.
  05 tpi-constant pic x(4).
  88 tpi-identifier value 'TPI '.
  .......
```

Each message ID is associated with a fixed length, which is known to your application.

2. The record descriptor word (RDW) technique

If your messages are variable length messages, you can implement a length field in the beginning of each message. Normally you would implement the length in a binary half-word with the value encoded in network byte order, but you may as well implement it as a text field.

```
* Transaction Request Message segment *
*---------------------------------------------------------------*
01 TRM-message.
  05 TRM-message-length pic 9(4) Binary Value 20.
  05 filler pic x(2) Value low-value.
```

A Beginner's Guide to MVS TCP/IP Socket Programming 73
3. The end-of-message marker technique

A third technique that is most often seen in C-programs is to send a null-terminated string. A null-terminated string is a string of bytes terminated by a byte with binary zero. The receiving program reads whatever data is on the stream and then loops through the received buffer separating each record where a null-byte is found. When the received records have been processed, the program issues a new read for the next chunk of data on the stream.

If your messages only contain character data, you may designate any non-display byte value as your end-of-message marker. Although this technique is most often seen with C-programs, it may be used with any programming language.

4. The TCP/IP buffer flushing technique

This technique is based on the observed behavior of the TCP protocol, where a send call followed by a recv call forces the sending TCP protocol layer to flush its buffers and forward whatever data may exist on the stream to the receiving TCP protocol layer. You can use this behavior to implement a half-duplex, flip-flop application protocol, where your two partner programs acknowledge the receipt of each message with, for example, a one-byte application acknowledgement message.
Figure 33. The TCP Buffer Flush Technique

In the above example, the client sends an 80 byte message. The server has issued a `recv` call for 1000 bytes but receives only the 80 bytes (RETCODE=80). The problem with this technique is that there is no guarantee that the server will receive the full 80-byte message on its receive call. It might only receive, for example, 30 bytes; but, with this technique, it has no way of detecting that it is missing another 50 bytes. The smaller the messages are the less likely it is that the server will only receive a part of the full message but you can never be fully sure.

If your partner program resides on any computer, ranging from mainframe computers to the smallest personal computer of any kind, you should not rely on this observed behavior; but use one of the safer techniques mentioned earlier.

___

Recommendation

We have included this technique in our description because we know it is widely used but we recommend that you only use it in controlled environments or in programs where you use non-blocking socket calls to implement your own time-out logic.

The first two techniques require that the receiving program is able to learn what is the content of the first bytes in the message, before it actually reads the entire message.

One way of solving this problem is to use the `peek` flag on a `recv` socket call.

A `recv` call with the peek flag on does not remove the data from the TCP buffers, but just copies the amount of bytes, you requested, into the application buffer you specified on the `recv` call.

If your message length field or message ID field is located, for example, within the first five bytes of each message, you can issue the following `recv` call:

```
01 socket-recv pic x(16) value 'RECV'.
01 recv-flag-peek pic 9(8) Binary value 2.
01 recv-peek-len pic 9(8) Binary value 5.
01 recv-peek-buffer.
  05 message-id pic x value space.
    88 tpi-query-reply value 'A'.
    88 tpi-response value 'B'.
  05 message-constant pic x(4).
    88 tpi-identifier value 'TPI'.
01 socket-descriptor pic 9(4) Binary Value zero.
01 errno pic 9(8) binary value zero.
01 retcode pic s9(8) binary value zero.
```
The `recv` call will block until some bytes have been received or the sender closes its socket. The above example is not complete because you cannot be sure that you actually received the requested 5 bytes. Your call may come back to you with only 1 byte received. In order to cope with that situation, you will have to repeat your `recv` call until all 5 bytes have been received. See "Reading and Writing Data From and To a Socket" in topic 5.8.2 for the technique to use.

If the other half connection closes the socket, the `recv` call will return zero in the retcode field.

The data is copied only into your application program buffer; it is still available for a real `recv` call, where you can specify the full length of the message you now know is available.

### 5.8.2 Reading and Writing Data From and To a Socket

Stream sockets during `read` and `write` calls may behave in a way that at a first glance you would expect to be an error. The `read` call may return fewer bytes, and the `write` call may write fewer bytes than requested. This is not an error, but a normal situation which your programs must deal with when they read or write data over a socket.

You may have to use a series of `read` calls to read a given number of bytes from a stream socket.

Each successful `read` call, returns in the retcode field, how many bytes were actually read. If you know you have to read, for example, 4000 bytes and the `read` call returns 2500, you have to reissue the `read` call with a new requested length of 4000 minus the 2500 already received (1500).

If you develop your program in COBOL, the following example will show you an implementation of such logic. In this example, the message to be read has a fixed size of 8192 bytes.
An actual execution of the program, with the above logic, used four read calls to retrieve the 8K of data. The first call returned 1960, the second call 3920, the third 1960 and the final call 352 bytes. It is not possible to predict in advance how many calls are needed to retrieve the message. It depends on the internal buffer utilization in TCP/IP. We observed other test cases where only two calls were needed to retrieve 8K.

In general, it would be good programming practice, whenever you know how many bytes to read, to issue read calls imbedded in logic, which is similar to the above.

If you work with short messages, you will in most situations receive the full message on the first read call; but there is absolutely no guarantee that it will work in all situations. The behavior of a write call is similar to that of a read call. You may have to issue more write calls in order to write out all the data you want to write.

* Buffer and length fields for write operation *

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>soket-write</td>
<td>pic x(16) value 'WRITE'</td>
</tr>
<tr>
<td>send-request-sent</td>
<td>pic 9(8) Binary value zero.</td>
</tr>
<tr>
<td>send-request-remaining</td>
<td>pic 9(8) Binary value zero.</td>
</tr>
<tr>
<td>send-buffer</td>
<td>pic x(8192) value space.</td>
</tr>
<tr>
<td>send-buffer-byte</td>
<td>pic x occurs 8192 times.</td>
</tr>
<tr>
<td>errno</td>
<td>pic 9(8) binary value zero.</td>
</tr>
<tr>
<td>retcode</td>
<td>pic s9(8) binary value zero.</td>
</tr>
</tbody>
</table>

* Send 8K data block *

A Beginner's Guide to MVS TCP/IP Socket Programming
move 8192 to send-request-remaining.
move 0 to send-request-sent.
Perform until send-request-remaining = 0
   Call 'EZASOKET' using socket-write
   socket-descriptor
   send-request-remaining
   send-buffer-byte(send-request-sent + 1)
   errno
   retcode
   If retcode < 0 then
      - process error and exit -
   end-if
   add retcode to send-request-sent
   subtract retcode from send-request-remaining
   If retcode = 0 then
      Move zero to send-request-remaining
   end-if
end-perform.

There are three groups of calls to use for reading and writing data over sockets:

**read** and **write**. These calls can only be used with connected sockets.
No processing flags can be passed on these calls.

**recv** and **send**. These calls also only work with connected sockets.
You can pass processing flags on these calls:

- **NOFLAG** - read or write data the same way as a read call or a write call would.
- **OOB** - read or write Out Of Band data (expedited data).
- **PEEK** - peek at data, but do not remove data from the buffers.

**recvfrom** and **sendto**. These calls work with both connected and non-connected sockets. You can pass addressing information directly as parameters on these calls.
The available flags are the same as described above.

A connected socket is either a stream socket for which a connection has been established, or it is a datagram socket for which you have issued a connect call to specify the remote datagram socket address.

### 5.8.3 Data Representation

If you use the socket API, your application must handle the issues related to different data representations on different hardware platforms. For character based data, some hosts use ASCII, while other hosts use EBCDIC. Translation between the two representations must be handled by your application. For integers, some hardware platforms use the big endian byte order approach (S/370/390, Motorola style), while others use little endian byte orders (Intel style). Figure 34 shows an example of the difference between big and little endian byte orders.

<table>
<thead>
<tr>
<th>big endian</th>
<th>high-order byte</th>
<th>low-order byte</th>
</tr>
</thead>
</table>
IBM S/370 and S/390 based computers all use the big endian byte order, while an IBM PS/2 uses the little endian byte order.

For data in protocol headers, these matters have been taken care of. TCP/IP has defined a network byte order standard to be used for all 16-bit and 32-bit integers that appear in protocol headers. This network byte order is based on the big endian byte order. This is the reason why, in the C-socket interface, you will find function calls like the following:

```
htons  This translates a short integer (2 bytes) from host byte order to network byte order.

ntohs This translates a short integer from network byte order to host byte order.

htonl  This translates a long integer (4 bytes) from host byte order to network byte order.

ntohl  This translates a long integer from network byte order to host byte order.
```

For the application data part of a message, it is all up to your socket-based application to deal with these matters. If you develop a server that serves clients on different hardware platforms, you must define your own standard and implement it as part of your application protocol.

In some instances it will be easiest for you to base your messages on text data. If you, as part of your message design, define a fixed text string in the beginning of each message, your application can test the contents of this string and decide if the data is in EBCDIC or is in ASCII. If the data is in ASCII, you can translate the full message from ASCII to EBCDIC on input and from EBCDIC to ASCII on output from MVS. An example of this design is the Transaction Request Message format used by the IMS Listener program. Bytes 4 to 11 have a fixed value of *TRNREQ*, which is used both to distinguish this message from other messages and to find out if the client is transmitting data in ASCII or EBCDIC.

If you mix text data and binary data in your messages, you must be sure to only apply translation between ASCII and EBCDIC to the text fields in your message.

If you use binary integer fields in your messages, it is recommended that you use the network byte order standard, which TCP/IP uses for all integers in protocol headers. If you design your messages according to the network byte order standard, your MVS programs do not need to translate or rearrange the bytes in binary integer fields. Your programs executing on little-endian hosts must use the integer conversion routines...
to convert integers between local format and the format used in the messages they exchange with your MVS programs.

Text data and binary two and four byte integers are fairly easy to handle in a heterogeneous computer environment. When it comes to more complex data types like floating point numbers or packed decimal, it becomes much more complicated because there is no generally accepted standard, and there is no easy support for transforming between the different formats. If you include these data types in your messages, be sure that the partner program knows how to interpret them. If the two computer systems use the same architecture, this is fully valid. If you exchange messages via socket programs between two MVS systems, you do not need to be worried about conversion.

5.9 Closing a Connection

For a connection-oriented reliable protocol, closing a socket imposes some problems because the TCP protocol layer must ensure that all data has been successfully transmitted and received before the socket resources can be safely freed at both ends.

The program that starts the close-down process, by issuing the first close call, is said to do an active close. The program that does the close call, as a consequence of the other program's close call, is said to do a passive close.

<table>
<thead>
<tr>
<th>Program A</th>
<th>TCP layer A</th>
<th>TCP layer B</th>
<th>Program B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call CLOSE</td>
<td>FIN segment sequence x</td>
<td>FIN segment sequence y</td>
<td>Call CLOSE</td>
</tr>
<tr>
<td></td>
<td>ACK sequence x+1</td>
<td>ACK sequence y+1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;______________</td>
<td>&lt;______________</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIMEWAIT</td>
<td>CLOSED</td>
<td></td>
</tr>
<tr>
<td>state</td>
<td>state</td>
<td>(2 * MSL)</td>
<td></td>
</tr>
<tr>
<td>CLOSED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>state</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 35. Closing Sockets

Program A does the active close, while program B does the passive close.

When a program calls the close socket function, the TCP protocol layer sends a segment that is known as a FIN (FINish) segment.

When program B receives the last acknowledgement segment, it knows that all data has been successfully transferred and that A has received and
processed the final FIN segment. The TCP protocol layer for program B can then safely remove the resources that were occupied by program B’s socket.

The TCP protocol layer for program A sends out an acknowledgement to the FIN segment it received from B; but program A’s TCP protocol layer does not know if that ACK segment arrived at program B’s TCP protocol layer or not. It must wait a reasonable amount of time to see if the final FIN segment from B is retransmitted indicating that B never received the final ACK segment from A. In that case, A must be able to retransmit the final ACK segment.

Program A’s socket cannot be freed until this time period has elapsed. The time period is defined to be twice the maximum segment life time, and it is normally between one and four minutes, depending on the TCP implementation.

If program A is the client in a TCP connection, this TIMEWAIT state does not impose any major problems. A client normally uses an ephemeral port number; and, if the client restarts before the TIMEWAIT period has elapsed, it is just assigned another ephemeral port number.

If program A, on the other hand, is the server in a TCP connection, this TIMEWAIT state does impose a problem. A server binds its socket to a predefined port number; and, if the server tries to restart and bind the same port number before the TIMEWAIT period has elapsed, it will receive an EADDRINUSE error code on the bind call. This situation may arise if a server crashes and you try to restart it immediately before the TIMEWAIT period has elapsed. In that case, you just have to be a little patient before you restart your server.

If the server is an important server and you cannot wait these one to four minutes, you may use the setsockopt call in the server to specify SO_REUSEADDR before it issues the bind call. In that case, the server will be able to bind its socket to the same port number as before even if the TIMEWAIT period has not elapsed; but the TCP protocol layer still prevents it from establishing a connection to the same partner socket address within the TIMEWAIT period. As clients normally initiate connections and clients use ephemeral port numbers, the probability of this situation arising is not very high.

5.9.1 Half Close
5.9.2 The Linger Option

5.9.1 Half Close

If you only want to close the stream in one direction, but still allow data to be transferred in the other direction, you may use the shutdown socket call instead of the close call. On the shutdown call, you are able to specify in which direction the stream should be closed down.

See Table 7 for effects on read and write calls when the stream is being shut down in one or both directions.

<table>
<thead>
<tr>
<th>Table 7. Effect of Shutdown Socket Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket calls in local program</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>write type calls</td>
</tr>
</tbody>
</table>
5.9.2 The Linger Option

By default a close socket call will return control to your program immediately, even if there is unsent data on the socket that still has to be transmitted. The data will be transmitted by the TCP protocol layer, but your program will not be notified of any errors. This is true for both blocking and non-blocking sockets.

You can request that you do not want control returned to your program before any unsent data has been transmitted and acknowledged by the receiver. You do so via the SO_LINGER option on a setsockopt call before you issue the actual close call. On the setsockopt call you pass the following two option value fields:

ONOFF   This is a full-word used to enable or disable the SO_LINGER option. Any non-zero value enables the option. A zero value disables the option.

LINGER  This is the linger time in seconds. This is the maximum time the close call will linger. If data is successfully transmitted before this time interval, control will be returned to your program. If this time interval expires before data has been successfully transmitted, control will also be returned to your program. You have no way you can distinguish between the two return causes.

Please note that, if you set a zero linger time, the connection will not be orderly closed but aborted, resulting in a RESET segment being sent to the connection partner instead of a normal close sequence.

Also note that, if the socket is in non-blocking mode, the close call is treated as if no linger option was set.

5.10 Blocking, Non-blocking and Asynchronous Socket Calls

The default mode of a socket call is blocking mode. All IBM TCP/IP for MVS socket APIs also support non-blocking socket calls. Some APIs, in addition to non-blocking calls, also support asynchronous socket calls.

Blocking

Let us first explain the default behavior of a socket call: the blocking mode. A blocking call will not return to your program until the event, you requested, has been completed. If, for example, you issue a blocking recv call, the call will not return to your program until data is available from the other socket application. A blocking accept call will not return to your program until a client connects to your socket program.
Non-blocking

You turn a socket into non-blocking mode via the ioctl call that specifies a command of FIONBIO and an argument that is a full-word (4 bytes) with a value of binary one (F'1'). Any succeeding socket calls against the involved socket descriptor will be non-blocking calls.

An alternative method is to use the fcntl call with a command code of binary four (F'4') and an argument with a value of four (F'4') to turn on non-blocking mode.

Non-blocking calls return to your program immediately with return information that tells you if the requested event happened or not. If the requested event did not happen, the error number is set to EWOULDBLOCK. This error number means that your call would have blocked if it had been a blocking call. If the call was, for example, a recv call, your program may implement its own wait logic and reissue the non-blocking recv call later. By using such a technique, your program may implement its own timeout rules and, for example, close the socket if it has not received any data from the partner program within an application determined period of time.

A new ioctl call can be used to turn the socket back into blocking mode with a command of FIONBIO and an argument that is a full-word with the value zero (F'0').

Please see "Datagram Socket COROL Client Program" in topic A.2 for an example of a datagram socket program that uses non-blocking recvfrom calls in order to implement its own timeout logic.

Note: The APPC Common Programming Interface for Communications (CPI-C) also provides so-called non-blocking calls. These calls, however, actually provide the more advanced facilities of TCP/IP asynchronous calls.

Asynchronous

Asynchronous calls are available with the Sockets Extended assembler macro API via the ECB keyword on the EZASMI macro call and in the IUCV API.

Like non-blocking calls, an asynchronous call also returns control to your program immediately. But in this case, there is no need to re-issue the call. When the requested event has taken place, the event control block that was specified on the EZASMI macro call is posted by the socket interface. Your program can either, at regular intervals, test if the wait bit is still on in the ECB, or it can issue an MVS WAIT macro call on this ECB or a combination of ECBS, where the socket call ECB is just one of a number of events for which the program is waiting.

Table 8 summarizes the actions taken by the socket programming interface (depending on the blocking or non-blocking state of a socket).

<table>
<thead>
<tr>
<th>Table 8. Effect of Blocking or Non-blocking Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call Type</td>
</tr>
<tr>
<td>--------------------</td>
</tr>
<tr>
<td>read type calls</td>
</tr>
</tbody>
</table>
Unless you are using the Sockets Extended assembler macro interface with an APITYPE of three on the `initapi` call, you are only allowed to have one outstanding socket call at any one time. The IUCV API also supports an APITYPE of three. The way you test pending activity on a number of sockets in a non-APITYPE three program is by using the `select` call. You pass a list of socket descriptors that you want to test for activity to the `select` call. You specify per socket descriptor what type of activity you want to test for:

- Pending data to read
- Ready for new write
- Any exception conditions

The `select` call can itself be blocking, non-blocking or, for the Sockets Extended assembler macro API or the IUCV API, asynchronous. If the call is blocking and none of the socket descriptors that are included in the list passed to the `select` call have had any activity, the call will not return to your program until one of them has activity, or until a timer value you pass on the `select` call expires.

In an Sockets Extended assembler macro program, you can use the asynchronous mode to control your own wait logic, where your program waits either for socket activity or some other events. A server program will typically have to wait for either socket activity or some operator command to shut it down. An Sockets Extended assembler macro program may wait on a list of two ECBS, where the first is the asynchronous `select` ECB, and the other one is an MVS modify command ECB (a CIB ECB).

A C-socket program may use the `selectex` call to include an external event in the list of events to wait for. This external event is typically an MVS modify command ECB.

Sockets Extended assembler macro APITYPE three programs will not be discussed further in this book. For more information on APITYPE three, please see Chapter 8 in *IBM TCP/IP for MVS: Application Programming Interface Reference*, SC31-7187. The information in the referenced chapter...
about APITYPE three also applies to Sockets Extended assembler macro programs.

5.11 Socket Programs and MVS Security

There are two aspects of security that we will include in the following discussion:

1. User or client authentication: Do we know the user and is the user who he/she claims to be?

2. Resource access authorization: Is the user allowed to use a specific resource or perform a specific function in MVS?

Unlike SNA LU 6.2, where user authentication can be made part of the conversation initiation, the TCP/IP transport protocol layers do not include any function that handles these aspects for us. If such functions are required, we must implement them in the socket applications and in the application protocol.

The following guidelines are related to socket programs running in native MVS address spaces. Both IMS sockets and CICS sockets include a security exit, you can use to authenticate client users that want to start IMS or CICS transactions via the IMS or CICS listener programs.

5.11.1 User or Client Authentication

5.11.2 Authorizing Access to MVS Resources

5.11.1 User or Client Authentication

In MVS we normally verify the authenticity of a user based on a user ID and a password that is passed to the MVS Security Access Facility (SAF) interface.

The client process will have to request the following information from the user:

- MVS user ID
- MVS password
- Optionally MVS group
- Optionally new MVS password

The application protocol must be designed so the client is able to pass the requested information to the server process, which must invoke the proper MVS functions to authenticate the user and return a positive or negative response to the client. Your application protocol should allow for a response from the server that tells the client user if the password has expired. The client process should allow the user to repeat the sign-on dialog enabling the user to type in a new password in addition to the previously entered user ID and current password.

In order to authenticate the user, the MVS program must use an authorized function in MVS: the RACROUTE REQUEST=VERIFY function. This function can only be used from an MVS process that runs in an authorized state. We will not recommend that you allow your server programs to run in an authorized MVS state, so our suggestion is to develop a user SVC routine, where you package the user authentication function into one SVC routine, which you invoke from your non-authorized server processes via the
Please see "TPIRACF Interface to RACROUTE REQUEST=VERIFY User SVC" in topic G.6 for a sample callable routine that can be used from any high-level language program, and "User SVC for RACROUTE REQUEST=VERIFY" in topic G.7 for a sample type 4 user SVC. In our implementation, we have added a check in the SVC code to see if the address space user of the calling program is authorized to use the SVC call. We have implemented this check to avoid giving all the programs in MVS access to the functions of the user SVC.

If you define your servers as resources in the RACF APPL resource class and permit your selected client users to use the individual APPL resources, you can add the server application name on the RACROUTE REQUEST=VERIFY call in order to decide if this particular user is or is not authorized to use this particular server.

As this authentication is done by the application code in the server, it is important to emphasize that an ill-behaving server can ignore the authentication return codes and continue processing the socket client request even if the user is unknown or not authorized to use the server program. You must ensure proper protection of the libraries where your server programs reside, in order to avoid having a healthy server program replaced by an ill-behaving replica.

Please see "Sample Stream Socket COBOL Server" in topic B.1 for a sample iterative server that authenticates each client user.

If you develop your server programs in C, you may optionally use the Kerberos services to authenticate your client users. But if you want to let the server issue further authorization requests in order to see if the client user may access specific MVS resources, like MVS data sets, you need an MVS user ID in addition to the Kerberos authentication.

### 5.11.2 Authorizing Access to MVS Resources

When you issue the RACROUTE REQUEST=VERIFY, an accessor environment element (ACEE) is constructed and a pointer to the ACEE is placed in the TCBSENV field in the current task control block (TCB).

If your server program opens an MVS data set, normal MVS authorization will be performed based on the ACEE pointed to by TCBSENV.

If your server program opens an MVS data set during initialization, before any clients have connected, the authorization will be done based on the address space security environment. The address space runs under the user ID of the batch job, or the started task user ID of a started task. The address space ACEE is pointed to by the ASXSBSENV field in the address space control block extension.

You can extend the access authorization to each individual user by a RACROUTE REQUEST=AUTH call. Under normal circumstances, your program does not have to run in an MVS authorized state in order to issue RACROUTE REQUEST=AUTH calls.

You are not limited to authorizing data set access. You can issue RACROUTE REQUEST=AUTH calls for any RACF defined resource.

Please see "TPIAUTH Issue RACROUTE REQUEST=AUTH for FACILITY Class" in topic G.8 for a sample callable routine that can be called from any high-level language program. The routine will issue an RACROUTE
REQUEST=AUTH call for the FACILITY class resource name passed to the routine by the calling program.

6.0 Chapter 6. Native MVS Concurrent Server Program

In this chapter we will guide you through the development of a concurrent server in the native MVS environment. Our sample concurrent server uses MVS subtasking and is implemented in assembler using the Sockets Extended assembler macro programming interface.

6.1 Concurrent Servers in the Native MVS Environment
6.2 MVS Subtasking Considerations
6.3 Program Structure
6.4 Initializing the Concurrent Server Program
6.5 Select Processing
6.6 Accepting Connection Requests from Clients

6.1 Concurrent Servers in the Native MVS Environment

The concurrent server is somewhat more complicated to implement. You have to split your logic into a main program and a child program. In addition to this split of your logic, you have to include logic to manage the different processes, which makes up your application.

In a UNIX based environment, you would implement such logic by means of the UNIX fork call. This call is not available in a traditional MVS environment, so you have to use some other facilities, which we will describe in the following sections of this book.

In an OpenEdition/MVS environment, the fork function is implemented using APPC/MVS to schedule and initiate a child process in another MVS address space, than the address space in which the original process is executing.

For the MVS address space examples presented in this book, we use the more traditional MVS subtasking facilities, where the main process and child processes operate as tasks within the same address space.

You can implement your concurrent server in both an IMS, a CICS and in a traditional MVS address space environment; but unlike the implementation of an iterative server, the implementation of a concurrent server differs between the environments. In this chapter, we will discuss the implementation of a concurrent server in an MVS address space, while we will return to the IMS and CICS concurrent server implementation in Chapter 9, "IMS Sockets" in topic 9.0 and Chapter 10, "CICS Sockets" in topic 10.0.

For the sake of simplicity, we will again limit the scope of our applications to the AF_INET addressing family and stream sockets.

If you are going to implement a high-performance server application that creates or accesses MVS resources of various kind, especially MVS data sets, you will most likely implement your server as a concurrent server in an MVS address space. The address space can be either TSO, batch or started task.

In order to implement concurrency in an MVS address space, you will have to use MVS multitasking facilities, which limits your available programming interfaces to Sockets Extended assembler macro programming interface or C sockets. You may also use the IUCV assembler programming
interface, but we see no real reason for doing so when you have the Sockets Extended assembler macro interface available.

For the Sockets Extended assembler macro interface, you can use standard MVS subtasking facilities by means of ATTACH and DETACH assembler macros.

If you use C sockets, you can use the subtasking facilities, which are part of the IBM implementation of C in an MVS environment.

We will use Sockets Extended assembler macro examples to illustrate the implementation of a concurrent server in an MVS address space environment.

6.2 MVS Subtasking Considerations

The fact that you use multiple tasks in an address space is the cause of some extra considerations, which you must take into account, when you design your application. These considerations apply equally to assembler programming and high-level languages that support subtasking.

When you use multiple tasks in an address space, your tasks may be concurrently dispatched on different processors if you execute your application on an n-way system. Two or more tasks may execute in parallel, one perhaps passing the other.

6.2.1 Access to Shared Storage Areas

6.2.2 Data Set Access

6.2.3 Task and Workload Management

6.2.4 Security Considerations

6.2.5 Reentrant Code

6.2.1 Access to Shared Storage Areas

If two tasks access the same storage area inside your address space, you must impose full control over the use of this storage area. If the storage area is a read-only area from which your tasks just fetch static information, there is no need for special attention. If the storage area is used to pass parameters between the tasks, you must ensure that only one task at a time is able to modify the contents of the storage area, and you must ensure that the task that is going to use the information in the storage area reads the information before it is modified by a third task. In other words, you have to serialize access to the shared resource (the storage area).

In an MVS environment you can do so via MVS latching services or the more traditional enqueue and dequeue system calls. In assembler you use the ENQ and DEQ macros.

| ENQ | DEQ |
| Task 1 --------> _______________> _________> |
| ^   ^           |
|     V           |
| Shared Storage Area | | | |
| ^   ^           |
|                V V |
| Task 2 ____________> .........._______> ___> |

A Beginner's Guide to MVS TCP/IP Socket Programming 88
Figure 36. Serialize Access to a Shared Storage Area

1. At time \( t_1 \), task 1 issues a serialize request by means of an enqueue call. On the enqueue call it passes two character fields that uniquely identifies the resource in question. What the value of these two fields are does not really matter; what matters is that other tasks use exactly the same values when they want to access this storage area. As no other task has issued an enqueue for the resource in question, task1 gets access to it and goes on making the required modifications in the storage area.

2. At time \( t_2 \), task 2 wants to access the same storage area and issues an enqueue call with the same resource names as task 1. Because task 1 already has enqueued, task 2 is placed in a wait and stays there until task 1 releases the resource.

3. At time \( t_3 \), task 1 releases the resource with a dequeue system call, and task 2 is immediately taken out of its wait and can now begin to make its modifications to the shared storage area.

4. At time \( t_4 \), task 2 has finished updating the shared storage area and releases the resource with a dequeue system call.

In the above example we assumed that we only needed to serialize access when the tasks wanted to update information in the shared storage area. There are situations where this is not sufficient. If you use a storage area to pass parameters to some kind of service task inside your address space, you must ensure that the service task has read the information and acted accordingly before another task in your address space tries to pass information to the service task in the same storage area.

Let us, for example, imagine that we have a service task, to which other tasks pass information that has to be written to some kind of logging or trace sysout file. In our design we have a common storage area, which is accessible from all tasks within our address space. We use fields in this common storage to pass parameters to the service task in order for it to print information on sysout.
Figure 37. Synchronize Use of a Common Service Task

1. At time $t_1$, task 1 gets access both to the common storage area and to implicitly use the service task in question.

2. At time $t_2$, task 2 also has a need to use the services of our service task, but it is placed into a wait, because task 1 already has the resource.

3. At time $t_3$, task 1 has finished placing values into the common storage area and signals the service task to start processing it. This is done via a POST system call. Immediately following this call, task 1 enters a wait, where it waits until the service task has completed its processing. The service task starts, processes the data in the common storage and prints, for example, a line or two to a sysout file.

4. At time $t_4$, the service task has finished its work and signals back to task 1 that task 1 can continue, while it enters a new wait for a new work request.

5. At time $t_5$, task 1 releases the lock it obtained at time $t_1$, and task 2 is immediately taken out of its wait and now starts filling in its values into the common storage area before posting the same service task to process a new request.

The above technique is relatively simple. It can be made much more complicated and also more efficient by using internal request queues where the requesting task does not need to wait for the service task to complete the request. You can experiment with such implementations, but they are outside the scope of this book.

When you use the enqueue system call, you have an option to test if a resource is available or not. In some situations, this might be handy if you do not want to enter a wait at some particular point in your processing but want to take some other action if the resource is not available.

6.2.2 Data Set Access

When you access MVS data sets in a multitasking environment, you must observe some general rules as follows:

1. A given DD-name can only be used by one open Data Control Block (DCB) at a time. If you need to have more DCBs open for the same data set, you have to use different DD-names. It can only be recommended for read access.

2. Only the task that opens a DCB can issue read and write requests using
that DCB. You cannot let your main task open a DCB and then have your
subtasks issue read or write requests to that DCB. One way to deal
with this is the technique we described above with a special services
task that opens a DCB to a particular data set. Other tasks then
issue requests to this service task for access to the data set. Such
a service task is in general called a Data Services Task (DST). One
very common implementation of a DST is the example we used above,
which was to print out log and trace information to a sysout file.

3. A last reminder for data set access is that authorization checking for
access to a data set is done when the data set is opened and not for
each read or write request. If you develop a multitasking server,
where you establish task level security environments for each
transaction entering your server, you have to consider how you will
authorize access to the information in a data set owned by a DST. You
can, of course, open and close the data set for each transaction, but
that may prove to be unacceptable from a performance point of view.
It depends on the nature of your application.

### 6.2.3 Task and Workload Management

When your program is started by MVS, it is executing as the main task of
the address space in which it was started.

In the examples we use in this book, we use the main task as the main
process of our concurrent server implementation. The child processes will
then be started as subtasks of the main task.

You have generally two ways to manage your child processes as:

1. Every time a connection request arrives, you start a new subtask,
   which processes one connection and then terminates.

2. During initialization, the main task starts a number of subtasks.
   Each subtask initializes and enters a wait-for-work status. When a
   connection request arrives, the main process selects the first subtask
   that is waiting for work and schedules the connection to that subtask.
   The subtask processes the connection and, when done, enters a new
   wait-for-work status.

The last approach is the most efficient because we only indulge the
overhead of creating new tasks once during server startup. It is also a
bit more complicated to implement than the first approach:

You must decide on the number of server subtasks you start during
initialization. If more connection requests arrive, than you have
server subtasks available, you must include code to deal with that
situation: either reject a connection or dynamically increase the
number of subtasks in your concurrent server address space. If you
include logic to increase subtasks dynamically during peak hours, you
might also include logic to decrease number of subtasks dynamically
during low-activity hours. This is what we term workload management.

The subtasks must be reusable and include logic to enter wait-for-work
status and be able to process connection requests serially.

The main process must be able to deal with situations where a server
subtask abends or terminates because of some other reason. Should the
subtask be reinstated, and how do you avoid reinstate-abend loops?

To implement what we call graceful shutdown, you also have to
implement a technique for signalling to the subtasks that they should terminate in an orderly manner. A simple technique is to post the subtask from the main process with a return code of zero for work and some other return code value for termination.

In the concurrent MVS server example you find in this book, we used the technique with a pool of subtasks that waited for work. We did not implement a dynamic increase of subtasks, but chose to send a negative reply back to the requester if no server subtasks were available.

6.2.4 Security Considerations

When you start your server address space in MVS, a security environment is established for the address space based on the user ID of your batch job or TSO user or based on the started task user ID associated with your started task procedure name in RACF started task table (ICHRIN03).

If you do nothing else, all tasks in your address space will execute under the security environment of the address space. Access to MVS resources during processing of client requests will be authorized based on the MVS address space security environment.

For some applications this may be sufficient. For others it is not.

You are able to work with task level security environments where each task in an address space may have a different security environment. You build and delete task level security environments with the RACROUTE REQUEST=VERIFY function in MVS. To use this, your task must run in an authorized state, so it is a function you must implement carefully in order not to jeopardize security in your environment. See "Socket Programs and MVS Security" in topic 5.11 for a general discussion on socket programs and security considerations.

6.2.5 Reentrant Code

It is not a requirement to develop reentrant code, but it is a more efficient use of main storage resources to do it. If you start 20 subtasks all using the same program, the program will be loaded into virtual storage in 20 copies if it is not reentrant but only in one copy if it is reentrant.

For most high-level languages, it is often just a matter of an option on the compile step.

If you develop your program in assembler, it might be somewhat more complicated; however, good use of macros for program initiation and termination may solve parts of the burden.

6.3 Program Structure

Figure 38 shows the basic logic in a multitasking concurrent server.
6.4 Initializing the Concurrent Server Program

1. The server must always start out with an **initapi** call before it uses any other socket calls.

```plaintext
EZASMI TYPE=INITAPI, *Initialize socket API
EZASMI MAXSOC=TPIMMAXS, *So many concurrent sockets
EZASMI SUBTASK=TPIMTCBE, *My TCB address in EBCDIC
EZASMI IDENT=IDENTSTR, *TCP/IP AS name and my AS name
EZASMI MAXSNO=TPIMMAXD, *Max. no of socket descriptors
EZASMI ERRNO=EZAERROR
EZASMI ICM R15,15,RETCODE, *Initapi OK
EZASMI BM EZAERROR, *- No.
```

---

The sequence numbers in the following sections all refer to the corresponding numbers in **Figure 38**.
Jobname and task ID is initialized to address space name and EBCDIC representation of TCB address before the call is issued.

During initialization the server main process attaches a number of subtasks. How you do this, what program you start and what parameters you pass is, of course, application dependent. In our sample server the subtask program is called TPISERV. For each subtask, the main process maintains a control block that we called TPISCB (Subtask Control Block). A pointer to this control block is passed to the subtask program.

*---------------------------------------------------------------------*
* Attach a subtask                                                    *
*---------------------------------------------------------------------*
LA R3,TPISCB *-> Subtask Control Block
LA R8,TPISTECB *-> Term. ECB
ATTACH EP=TPISERV, *Server subtask main module
       PARAM=((R3)), *Pass TPISCB as only parameter
       ECB=(R8) *Termination ECB
ST R1,TPISTCB *-> TCB of subtask

When the subtask terminates, either because of an abend or because of normal termination, the Event Control Block (ECB) at label TPISTECB is posted by MVS.

During subtask initialization, the subtasks issue initapi calls, where they identify themselves with the same address space name as the main process and with an EBCDIC representation of their TCB addresses.

*---------------------------------------------------------------------*
* Initialize socket API in subtask with passed values                *
*---------------------------------------------------------------------*
MVC IAPITCP,TPIMTCPI *TCP/IP address space name
MVC IAPIAS,TPIMCNAM *Our address space name
EZASMI TYPE=INITAPI, *Initialize socket API
       MAXSOC=IAPISOC, *This many sockets
       SUBTASK=TPISTCBE, *My TCB address in EBCDIC
       IDENT=IAPIIDEN, *TCP/IP AS name and my AS name
       MAXSNO=IAPISNO, *This many socket descriptors
       ERNNO=ERRNO, *
       RETCODE=RETCODE
ICM R15,15,RETCODE *Did we do well ?
BM EZAERROR *- No, deal with it.
* IAPIIDEN DS 0C
IAPITCP DC CL8' ' *TCP/IP Address space name
IAPIAS DC CL8' ' *Child process address space name
* TPISTCBE DC CL8' ' *Child process TCB address in EBCDIC
* IAPISNO DC AL4(10) *Max socket descriptors
IAPISOC DC AL2(10) *Max sockets
The subtasks then enter a wait for work status, waiting for the main process to pass work.

The server main process now issues the same series of socket calls as the iterative server to obtain a socket, bind it to the server port number and open it in passive mode via a listen call.

### 6.5 Select Processing

3 When all initialization has been done, and the server main process is ready to enter normal work, it builds a bit mask for a `select` call. The `select` call is used to test pending activity on a list of socket descriptors owned by this process. Before you issue the `select` call, you must construct three bit strings representing the sockets you want to test for as follows:

- Pending read activity
- Pending write activity
- Pending exceptional activity

The format of the bit strings is a bit awkward for an assembler programmer who is used to bit strings starting off from the left. These do not.

The first rule is that the length of a bit string is always expressed as a number of full-words. If the highest socket descriptor you want to test is socket descriptor number three, you have to pass a 4 byte bit string as this is the minimum length. If the highest number is 32, you must pass 8 bytes (2 full-words).

The number of full-words in each select mask can be calculated as

\[ \text{INT} \left( \frac{\text{number of socket descriptors}}{32} \right) + 1 \]

Let us look at the first full-word you pass in a bit string:

<table>
<thead>
<tr>
<th>Bit nbr:</th>
<th>!</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 0:</td>
<td>!</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SD nbr:</td>
<td>!</td>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
</tr>
<tr>
<td>Byte 1:</td>
<td>!</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SD nbr:</td>
<td>!</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>Byte 2:</td>
<td>!</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SD nbr:</td>
<td>!</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Byte 3:</td>
<td>!</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SD nbr:</td>
<td>!</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

We use standard assembler numbering notation; the left most bit or byte is relative zero.

If you want to test socket descriptor number 5 for pending read activity, you raise bit2 in byte 3 of the first full-word (X'00000020'). If you want to test both socket descriptor 4 and 5, you raise both bit2 and bit3 in byte3 of the first full-word (X'00000030').
If you want to test socket descriptor number 32, you must pass two full-words, where the numbering scheme for the second full-word resembles that of the first. Socket descriptor number 32 is bit7 in byte3 of the second full-word. If you wanted to test socket descriptors 5 and 32, you would pass two full-words with the following content:
X'0000002000000001'.

The bits in the second full-word represents the following socket descriptor numbers:

<table>
<thead>
<tr>
<th>Bit nbr:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 4:</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SD nbr:</td>
<td>63</td>
<td>62</td>
<td>61</td>
<td>60</td>
<td>59</td>
<td>58</td>
<td>57</td>
<td>56</td>
</tr>
<tr>
<td>Byte 5:</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SD nbr:</td>
<td>55</td>
<td>54</td>
<td>53</td>
<td>52</td>
<td>51</td>
<td>50</td>
<td>49</td>
<td>48</td>
</tr>
<tr>
<td>Byte 6:</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SD nbr:</td>
<td>47</td>
<td>46</td>
<td>45</td>
<td>44</td>
<td>43</td>
<td>42</td>
<td>41</td>
<td>40</td>
</tr>
<tr>
<td>Byte 7:</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SD nbr:</td>
<td>39</td>
<td>38</td>
<td>37</td>
<td>36</td>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
</tr>
</tbody>
</table>

To set and test these bits in an easy way, we developed the following assembler macro:

```assembly
MACRO
TPIMASK &TYPE,&MASK=,&SD=
.* *******************************************************************
.* TYPE is either:
.* TEST for testing a bit
.* SET for setting a bit
.* Mask Bit mask area
.* SD A halfword containing socket descriptor
.* *******************************************************************
SR R14,R14 *Nullify
AIF ('&SD'(1,1) EQ '(').SDREG
LH R15,&SD *Socket descriptor
AGO .SDOK
.SDREG ANOP
LR R15,&SD *Socket descriptor
.SDOK ANOP
D R14,=A(32) *Divide by 32
SLL R15,2 *Multiply offset with word length
AIF ('&MASK'(1,1) EQ '(').MASKREG
LA R1,&MASK *Here mask starts
AGO .MASKOK
.MASKREG ANOP
LR R1,&MASK *Here mask starts
.MASKOK ANOP
AR R15,R1 *Here our word starts
LA R1,1 *Rightmost bit on
SLL R1,0(R14) *Shift left rest from division
O R1,0(R15) *Or bits from mask
AIF ('&TYPE' EQ 'SET').DOSET
C R1,0(R15) *If equal, bit was on
MEXIT
.* *******************************************************************
```
If you develop your program in another programming language, you may be able to benefit from the EZACIC06 routine, which is provided as part of IBM TCP/IP for MVS. This routine translates between a character string mask (one byte per flag) and a bit string mask (one bit per flag). If you use the `select` call in COBOL, you will find EZACIC06 very useful.

You build the three bit strings for the socket descriptors you want to test, and the `select` call passes back three corresponding bit strings with bits raised for those of the tested socket descriptors that have pending activity.

```cobol
*---------------------------------------------------------------------*
* Test for socket descriptor activity via the SELECT call            *
*---------------------------------------------------------------------*
EZASMI TYPE=SELECT, *Select call C
MAXSOC=TPIMMAXD, *Max. this many descr. to test C
TIMEOUT=SELTIMO, *One hour timeout value C
RSNDMSK=RSNDMASK, *Read mask C
RRETMSK=RRETMASK, *Returned read mask C
WSNDMSK=WSNDMASK, *Write mask C
WRETMSK=WRETMASK, *Returned write mask C
ESNDMSK=ESNDMASK, *Exception mask C
ERETMSK=ERETMASK, *Returned exception mask C
ECB=ECBSELE, *Post this ECB when activity occurs C
ERRNO=ERRNO, *- ECB points to an ECB plus 100 C
RETCODE=RETCODE, *- bytes of workarea for socket C
ERROR=EZAERROR *- interface to use.
ICM R2,15,RETCODE *If Retcode < zero it is
BM EZAERROR *- an error
*
SELMASKS DS 0F
RSNDMASK DC XL8'00000000' *Read mask
RRETMASK DC XL8'00000000' *Returned read mask
WSNDMASK DC XL8'00000000' *Write mask
WRETMASK DC XL8'00000000' *Returned write mask
ESNDMASK DC XL8'00000000' *Exception mask
ERETMASK DC XL8'00000000' *Returned exception mask
*
NOSELCD DC A(0) *Keep track of selected sd's
SELTIMO DC A(3600,0) *One hour timeout
ECBSELE DC A(0) *Select ECB
DC 100X'00' *Required by EZASMI
*
TPIMMAXD DC AL4(50) *Maximum descriptor number
*
ERRNO DC A(0) *Errorno from EZASMI
RETCODE DC A(0) *Returncode from EZASMI
*
```

In the above `select` call we use the asynchronous facilities of the Sockets Extended assembler macro interface. By placing an ECB parameter on the `EZASMI` macro call, the `select` call will not block our process; we will receive control immediately even if none of the specified socket descriptors had activity. You can use this technique, if you want to enter a wait, waiting for a series of events of which the completion of a `select` call is just one. In our sample application, we placed the main process into a wait from where it would return if any of the following
A Beginner's Guide to MVS TCP/IP Socket Programming

events occurred:

1. Socket descriptor activity occurred and the select call was posted.
2. One of our subtasks terminated unexpectedly.
3. The MVS operator issued a Modify command to stop the server.

If the reason for exiting the wait is socket activity, you must synchronize your task with the socket interface by issuing an EZASMI Synchronize call.

```plaintext
*---------------------------------------------------------------------*
* Synchronize after asynchronous SELECT call *
*---------------------------------------------------------------------*
EZASMI TYPE=SYNC, *Synchronize function C
    ECB=ECBSELE, *Select ECB plus 100 bytes workarea C
    ERNNO=ERNNO, * C
    RETCODE=RETCODE, * C
    ERROR=EZAERROR
    ICM R15,15,RETCODE *Was everything OK
    SM EZAERROR *- No, some error
    ST R15,NOSELCD *Number of sd’s selected

The areas pointed to by the return mask keywords on the previous select call is not filled in with the returned bit masks until you issue the synchronize call.

The number of socket descriptors with pending activity is returned in the RETCODE field. You must process all selected socket descriptors before you issue a new select call. A selected socket descriptor will only be selected once.

When a connection request is pending on the socket for which the main process issued the listen call, it will be reported as a pending read.

When the main process has given a socket, and the subtask has taken the socket, the main process socket descriptor is selected with an exception condition. The main process is expected to close the socket descriptor when this happens.

6.6 Accepting Connection Requests from Clients

If the listener socket was selected with a pending read, a new connection request has arrived, and the following socket call must be an accept:

```plaintext
*---------------------------------------------------------------------*
* ACCEPT the connection from a client *
*---------------------------------------------------------------------*
EZASMI TYPE=ACCEPT, *Accept new connection C
    S=TPIMSN, *On listener socket descriptor C
    NAME=SOCSTRUC, *Returned client socket structure C
    ERNNO=ERNNO, * C
    RETCODE=RETCODE, * C
    ERROR=EZAERROR
    ICM R15,15,RETCODE *OK?
    SM EZAERROR *- No, error indicated
    ST R15,NEWSOC *Returned new socket descriptor

* SOCSTRUC DS 0F *ACCEPT Socket address structure
  SSTRFAM DC AL2(2) *TCP/IP Addressing family
  SSTRPORT DC AL2(0) *Port number
The `accept` call returns a new socket descriptor representing the connection with the client. The original listen socket descriptor is available for a new `select` call.

### 6.6.1 Give Socket to Subtask

**6.6.2 Take Socket from Main Process**

#### 6.6.1 Give Socket to Subtask

The socket represented by the new socket descriptor has to be passed on to an available subtask. What technique the main process uses to find an available subtask is not so important. Let us assume that the main process has found an available subtask to which it gives the socket via a `givesocket` call:

```c
*---------------------------------------------------------------------*
* Give socket to subtask                                              *
*---------------------------------------------------------------------*
MVC CLNNAME,TPIMCNAM *Our Client ID Address Space Name
MVC CLNTASK,TPISTCBE *Give to this subtask
EZASMI TYPE=GIVESOCKET, *Givesocket C
S=NEWSOC, *Give this socket descriptor C
CLIENT=CLNSTRUC, *- to a specific child process C
ERRNO=ERRNO, C
RETCODE=RETCODE, C
ERROR=EZAERROR
ICM R15,15,RETCODE *OK ?
SM EZAERROR "- No, tell about it.

*  
CLNSTRUC DS 0F *GIVESOCKET: Client structure
CLNFAM DC A(2) *TCP/IP Adressing family
CLNNAME DC CL8' ' *Address space name of target
CLNTASK DC CL8' ' *Task ID of child process subtask
CLNRESV DC XL20'00' *Reserved
*  
NEWSOC DC AL2(0) *Socket descriptor from Accept
*  
ERRNO DC A(0) *Errorno from EZASMI
RETCODE DC A(0) *Returncode from EZASMI
```

If you are programming in C, for example, you may not be able to decide the full client ID of the subtask. In that case, you can pass the task ID field as eight blanks on the `givesocket` call, which means that any task within your own address space can take the socket. But only the task to which you pass the socket descriptor number will actually take it, so it is not a big exposure.

After you have issued the `givesocket` call, you must remember to include the given socket descriptor in the exception select mask on the next `select` call.
Your main process is now ready to wake up the selected subtask via a POST system call.

If there were no more sockets selected on the previous `select` call, your main process can now build a new set of select masks and issue a new `select` call.

### 6.6.2 Take Socket from Main Process

6. The subtask is brought back to life as a result of the POST system call issued from the main process, and it immediately issues a `takesocket` call to receive the socket, which was passed from the main process.

```plaintext
*---------------------------------------------------------------------*
* Take socket from main process                                      *
*---------------------------------------------------------------------*

EZASMI TYPE=TAKESOCKET, *Takesocket C
CLIENT=TPIMCLNI, *Main task client id structure C
SOCRECV=TPISSOD, *Main task socket descriptor C
ERRNO=ERRNO, C
RETCODE=RETCODE, C
ERROR=EZAERROR

ICM R15,15,RETCODE *Did we do well ?
BM EZAERROR *- No, deal with it.
STH R15,TPISNSOD *Server subtask socket descr.no

* 

TPIMCLNI DS 0C *Main task client id
TPIMCDOM DC A(0) *Domain: AF-INET
TPIMNAM DC CL8' ' *Our address space name
TPIMCTSK DC CL8' ' *Main task TCB address in EBCDIC
DC 20X'00' *Reserved (part of clientid)

* 

TPISSOD DC AL2(0) *Parent socket descr. no.
TPISNSOD DC AL2(0) *Subtask socket descr. no.
```

In order to take a socket, the subtask must have knowledge of the client ID of the task that gave the socket and the socket descriptor used by that task. These values must be passed to the subtask from the main process before a `takesocket` call can be issued.

On the `takesocket` call, you specify the full client ID of the process that gave the socket, and you specify the socket descriptor number used by the process that gave the socket.

A new socket descriptor number to be used by the subtask is returned in the `RETCODE` field if the `takesocket` call is successful.

As soon as your subtask has taken the socket, the main process will be posted in its pending `select` with a pending exception activity, which means that the main process must close its socket descriptor.

From here on, processing is quite trivial.

7. The client sends its request to the subtask, which processes it and sends back a reply 8.

9. Finally the client process and the server subtask close their sockets, and the server subtask enters a new wait for work status.
Chapter 7. Socket Client Programs

A socket based client program will use a subset of the socket calls we have discussed so far, plus a few extra, which are mainly used by client programs.

As many of the calls apply to both server and client programs, and we have shown both COBOL and assembler examples of server program socket calls until now; we will now illustrate the client socket calls with REXX samples.

From a socket point of view, there is no difference between a client program that executes in a normal MVS address space and one that executes in an IMS or CICS environment. There is a difference in programming language support. REXX sockets for example, is not supported in either IMS or CICS environments.

For a sample COBOL based client, please see "Sample Stream Socket COBOL Client" in topic B.2.

For a sample REXX based client, please see "REXX Client" in topic E.1 or "TPI REXX Client" in topic H.2.1.

7.1 General REXX Subroutine for Socket Calls

7.2 Initializing the Socket API

7.3 Connecting a Client to a Server

7.4 Closing the Socket

7.5 Terminating the REXX Socket API

7.1 General REXX Subroutine for Socket Calls

In the sample REXXX programs we developed, we packaged the actual REXXX socket call into a REXXX subroutine in order to enable easy tracing facilities. If we had problems with the socket calls, we could just add the appropriate REXXX statements to this one subroutine, and we would enable trace output for all socket calls.

See "TPI REXXX Client" in topic H.2.1 for a sample REXXX that uses this subroutine with trace points imbedded.

We called the subroutine DoSocket.

```rexx
/*-------------------------------------------*/
/* */
/* DoSocket procedure. */
/* */
/* Do the actual socket call, and parse the return code. */
/* Return rest of string returned from socket call. */
/* */
/* -------------------------------------------*/
DoSocket:
    numargs = arg() /*Number of passed args */
    argstring = '' /*Init arg string */
    do subix=1 to numargs /*Build argument string */
        argstring = argstring||'arg('subix')'
    end /*for the socket call */
    if subix=numargs then do /*If not last argument */
        argstring = argstring||',' /*add a comma */
    end /* */
end /* */
```

A Beginner’s Guide to MVS TCP/IP Socket Programming
7.2 Initializing the Socket API

If you use the Sockets Extended programming interfaces, you will have to issue the `initapi` call in order to establish your client programs clientID with the TCP/IP system address space.

In a REXX program you use the `initialize` call to do this:

```rexx
/* Initialize REXX socket interface */
sockval = DoSocket('Initialize', 'tpirexxc')
if sockrc <> 0 then do
   say 'Socket initialize failed, rc='sockrc
   say sockval
   exit(sockrc)
end
```

A REXX socket program will get a client ID, where the address space name is set to the correct address space name and the task ID is set to the value of the first parameter passed on the `initialize` call, which in the above scenario is a text string with the value `tpirexxc`.

7.2.1 Getclientid

7.2.1 Getclientid

You use the `getclientid` call to obtain the client ID by which your program has been identified to the TCP/IP address space.

A `getclientid` call will in REXX look like:

```rexx
/* Get our client ID */
sockval = DoSocket('Getclientid')
if sockrc <> 0 then do
   say 'Getclientid failed, rc='sockrc
   say sockval
   exit(sockrc)
end
```

The client ID is returned as a string. In the above example, the value of the returned string is:

```
AF_INET TSOUSER1 tpirexxc
```

Domain is AF_INET, address space name is TSO user ID and task ID is the value that was passed on the `initialize` call.

7.3 Connecting a Client to a Server
If you know the IP address of the server, you can go on issuing a **socket** call followed by a **connect** call.

If you only know the host name, you will have to resolve the host name into one or more IP addresses using the **gethostbyname** call.

```rexx
 servipaddr = DoSocket('Gethostbyname', tpiserver)
 if sockrc <> 0 then do
   say 'Gethostbyname failed, rc='sockrc
   say sockval
   x=Doclean
   exit(sockrc)
 end
```

The REXX **gethostbyname** call returns a list of IP addresses if the host is a multihomed host. You can parse the REXX string and place the IP addresses into a REXX stem variable using the following piece of REXX code:

```rexx
 numips = words(servipaddr)
 do i = 1 to numips
   sipaddr.i = word(servipaddr, i)
 end
 sipaddr.0 = numips
```

When you issue a **connect** call to an IP address that is currently not available, your connect call will eventually time out, giving an error number of 60 (ETIMEDOUT). The socket you used on such a failed **connect** call cannot be reused for another **connect** call. If you try to do it, you will receive error number 22 (EINVAL). You have to close the socket, and get a new socket before you reissue the **connect** call with the next IP address in the list of IP addresses that were returned by the **gethostbyname** call.

The **connect** call can be placed in a loop that is terminated when either a connect is successful or the list of IP addresses is exhausted.

```rexx
 i = 1
 connected = 0
 do until (i > sipaddr.0 | connected)
   sockdescr = DoSocket('Socket')
   if sockrc <> 0 then do
     say 'Socket failed, rc='sockrc
     exit(sockrc)
   end
```

A Beginner's Guide to MVS TCP/IP Socket Programming 103
7.3.1 Accessing a Host Entry Structure with EZACIC08

If you develop your socket program in other programming languages other than REXX, the `gethostbyname` call will not return a string with IP addresses but a pointer to a storage area that is known as a host entry structure or, for short, a HOSTENT structure. A `gethostbyaddress` call will also return a host entry structure to your program.

The host entry structure may be complicated depending on the number of aliases and IP addresses a given host has.

If you develop your programs in assembler, this structure is quite straightforward and does not impose any major problems to you. But if you develop your program in, for example, COBOL, you could have problems extracting relevant information from this structure because COBOL is not famous for its pointer manipulation features.

IBM TCP/IP for MVS supplies you with a routine that makes it more simple to extract information from the host entry structure. The name of this routine is EZACIC08.

---

**Diagram of Host Entry Structure**

```
Hostentry pointer -> Host name X'00'
| name pointer | Alias list
| Alias list pointer | Alias 1 pointer -> Alias 1 X'00'
| Family F'2' | Alias n pointer -> Alias n X'00'
| Address len F'4' | F'0'
| Address list pointer | INET address list
```

---

A Beginner's Guide to MVS TCP/IP Socket Programming
Figure 39. Host Entry Structure

The notation X’00’ is used to show that the value is a variable length string that is terminated by a null-byte.

In a COBOL program, the gethostbyname call and the use of EZACIC08 followed by a socket call and a connect call could be implemented as follows:

*---------------------------------------------------------------*
* Variables used for socket calls in general                    *
*---------------------------------------------------------------*
01 errno pic 9(8) binary value zero.
01 retcode pic s9(8) binary value zero.
*---------------------------------------------------------------*
* Variables used for the GETHOSTBYNAME Call                    *
*---------------------------------------------------------------*
01 socket-gethostbyname pic x(16) value 'GETHOSTBYNAME '.
01 host-namelen pic 9(8) Binary Value 5.
01 host-name pic x(5) Value 'mvs18'.
01 host-entry-addr pic x(4) Value low-value.
*---------------------------------------------------------------*
* Variables used for the call to EZACIC08                      *
*---------------------------------------------------------------*
01 host-alias-seq pic 9(4) Binary Value zero.
01 host-addr-seq pic 9(4) Binary Value zero.
01 host-name-length pic 9(4) Binary Value zero.
01 host-name-value pic x(255) Value space.
01 host-alias-count pic 9(4) Binary Value zero.
01 host-alias-length pic 9(4) Binary Value zero.
01 host-alias-value pic x(255) Value space.
01 host-addr-type pic 9(4) Binary Value zero.
01 host-addr-length pic 9(4) Binary Value zero.
01 host-addr-count pic 9(4) Binary Value zero.
01 host-addr-value pic x(4) Value low-value.
01 host-return-code pic s9(8) Binary Value zero.
*---------------------------------------------------------------*
* Variables used for the CONNECT Call                          *
*---------------------------------------------------------------*
01 socket-connect pic x(16) value 'CONNECT '.
01 server-socket-address.
   05 server-afinet pic 9(4) Binary Value 2.
   05 server-port pic 9(4) Binary Value 3001.
   05 server-ipaddr pic x(4) Value low-value.
   05 filler pic x(8) value low-value.
01 connect-status pic 9(4) Binary value zero.
88 connect-done value 1.

* Get IP addresses out of the HOST Entry structure.

*
Move zero to connect-status.
Perform until ((host-addr-count = host-addr-seq and host-addr-seq > 0) or connect-done)
If host-alias-seq > host-alias-count then subtract 1 from host-alias-seq
end-if
Call 'EZACIC08' using host-entry-addr
host-name-length
host-name-value
host-alias-count
host-alias-seq
host-alias-length
host-alias-value
host-addr-type
host-addr-length
host-addr-count
host-addr-seq
host-addr-value
host-return-code
If host-return-code < 0 then - process error -
end-if
Move host-addr-value to server-ipaddr

* Get an AF_INET socket to use for connect *
*------------------------------------------------------------*
Call 'EZASOKET' using soket-socket
afinet
soctype-stream
proto
errno
retcode
If retcode < 0 then - process error -
end-if
Move retcode to socket-descriptor

* Try to connect to iterative server on returned IP address *
*------------------------------------------------------------*
If host-return-code = 0 then
Call 'EZASOKET' using soket-connect
socket-descriptor
server-socket-address
errno
retcode
If retcode < 0 then
Call 'EZASOKET' using soket-close
socket-descriptor
errno
retcode
If retcode < 0 then
- process error -
end-if
else
move 1 to connect-status
end-if
end-if
end-perform.

if not connect-done then
- process error -
else
- connected to server -
end-if

1 Each call to EZACIC08 will advance the alias sequence and address sequence numbers. In this context we are only interested in the IP addresses; so, in order to avoid a retcode -2 from EZACIC08, we reset the alias sequence to the last alias sequence number if the alias sequence number exceeds the available alias names.

Please note that EZACIC08 may return the following return code values:

-1 The host entry structure is invalid.
-2 The alias sequence field is invalid (greater than the alias count field).
-3 The address sequence field is invalid (greater than the address count field).

7.4 Closing the Socket

When your client program has connected to the server, they can exchange whatever amount of data they find relevant until one of them closes down the socket.

    /*---------------------------------------------*/
    /* Close the socket                               */
    /*---------------------------------------------*/
    sockval = DoSocket('Close', sockdescr)
    if sockrc <> 0 then do
        say 'Socket Close failed, rc='sockrc
        say sockval
        exit(sockrc)
    end

7.5 Terminating the REXX Socket API

In a REXX socket environment you will finish off using the socket interface with a terminate socket call.

    /*---------------------------------------------*/
    /* Terminate socket interface                   */
    /*---------------------------------------------*/
    sockval = DoSocket('Terminate')
    if sockrc <> 0 then do
        say 'Terminate failed, rc='sockrc
        say sockval
Chapter 8. Datagram Socket Programs

This chapter explains the special characteristics of a datagram socket program that uses UDP protocols.

Please see Appendix A, "Sample Datagram Socket Programs" in topic A.0 for sample datagram socket programs.

8.1 Datagram Socket Characteristics
8.2 Datagram Socket Program Structure
8.3 Use of Connect on a Datagram Socket
8.4 Transferring Data Over a Datagram Socket

8.1 Datagram Socket Characteristics

The most significant characteristics of datagram sockets are as follows:

1. Datagram sockets are connection-less.

   There is no connection setup done by the UDP protocol layer. No data is exchanged between sending and receiving UDP protocol layers until your application issues its first `send` call.

   If your UDP server program has not been started or it resides on a host that is currently unreachable from your client host, your client UDP application may wait forever for a reply to the datagram it sent to a UDP server. You will normally have to implement time-out logic in your client UDP program to detect this situation.

2. The UDP protocol layer does not implement any reliability functions.

   The implication of this is that a datagram sent from one UDP program to another may never arrive. Neither the sending program nor the anticipated receiving program will ever learn from the UDP protocol layer that such a condition exists.

   If your UDP application requires reliability, you must implement reliability code in your UDP client and server programs. This includes the ability to detect missing datagrams, datagrams arriving out of sequence, duplicated datagrams or corrupted datagrams.

   It is not a trivial matter to implement such functions, and we recommend you use the TCP protocols and not the UDP protocols if your application has strict reliability requirements.

3. Unlike a TCP socket, where there is no one-to-one relationship between `send` calls and `recv` calls, a UDP socket `send` corresponds to exactly one UDP socket `recv` call.

8.2 Datagram Socket Program Structure

The terms client and server are somewhat misleading for datagram socket programs. Two socket programs that have each bound a socket to a local address may send any number of datagrams to each other in any sequence.
The program that sends the first datagram will, in our terminology, act as a client. Any datagram sent to a destination address for which no program has bound a socket is lost. Care must be taken so that the program you intend to be the client does not begin sending datagrams until after the server program has bound its socket to the expected destination address.

The typical structure for a datagram socket server is a structure that resembles the iterative server we defined in "Iterative Server" in topic 3.7.1.

<table>
<thead>
<tr>
<th>Datagram Client Program</th>
<th>Datagram Server Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize socket API</td>
<td>Initialize socket API</td>
</tr>
<tr>
<td>Obtain a datagram socket</td>
<td>Obtain a datagram socket</td>
</tr>
<tr>
<td>Bind socket to local address</td>
<td>Bind socket to local address</td>
</tr>
<tr>
<td>Send a datagram &gt;Receive a datagram</td>
<td>Do forever</td>
</tr>
<tr>
<td>Receive a datagram&lt;Send a datagram</td>
<td>Process data</td>
</tr>
<tr>
<td>Close socket</td>
<td>Close socket</td>
</tr>
<tr>
<td>Terminate socket API</td>
<td>Terminate socket API</td>
</tr>
</tbody>
</table>

Figure 40. Datagram Server Program Structure

The server program must bind its socket to a predefined server port number, so the clients know to which port they should send their datagrams. In the socket address structure that the server passes on the bind call, it can specify if it will accept datagrams from all the available network interfaces, or if it will only receive incoming datagrams from a specific network interface. This is done by setting the IP address field of the socket address structure to either INADDR_ANY or a specific IP address.

The client program will also need to bind its socket to a local address, if it wants the server program to be able to return a datagram to it. In contrast to the server, the client does not need to specify any specific port number on the bind call; an ephemeral port number chosen by the UDP protocol layer will be sufficient. This is called a dynamic bind.

The server enters a blocking recvfrom call. In this example, we use the recvfrom call, because in addition to a datagram, it also returns the remote socket address. Our UDP server needs that address in order to return a reply to the client, which it does by issuing a sendto call, where it passes both a datagram and the remote socket address of the client as parameters on the call.

After the server has processed one request, it loops back into a new blocking recvfrom call, waiting for another datagram from possibly another client.

If more UDP datagrams arrive in the UDP protocol layer destined for the server UDP socket, the UDP protocol layer queues those datagrams and passes them to the server one after the other on succeeding recvfrom calls.
The UDP receive queue size in IBM TCP/IP Version 3 Release 1 for MVS is by
default 20, but you can customize IBM TCP/IP for MVS to use an unlimited
queue size by specifying the NOUDPQUEUELIMIT keyword in the ASSORTEDPARMS
section of the tcpip.v3r1.PROFILE.TCPIP configuration data set.

If the UDP receive queue exceeds its maximum size, any excess datagrams
are discarded without further notification.

8.3 Use of Connect on a Datagram Socket

You are able to use the connect call on a datagram socket, but it does not
perform the same function for a datagram socket as it does for a stream
socket.

On a connect call, you specify the remote socket address you want to
exchange datagrams with. This serves two purposes:

1. On succeeding calls to send datagrams, you can use the send call
   without specifying a destination socket address. The datagram will be
   sent to the socket address you specified on the connect call.

2. On succeeding calls to receive datagrams, only datagrams that
   originate from the socket address you specified on the connect call
   will be passed to your program from the UDP protocol layer.

Please remember, that a connect call for a datagram socket does not
establish any connection. No data is exchanged over the IP network as a
result of a connect call for a datagram socket. The functions performed
are local, and control is returned to your application immediately.

8.4 Transferring Data Over a Datagram Socket

If you are use to the MVS notion of records, it is extremely simple to
transfer data over a datagram socket. You send and you receive records of
data. One send call results in exactly one recv call.

If your sending program sends a datagram of, for example, 8192 bytes and
your receiving program issues a recv call, where it specifies a buffer
size of, for example, 4096 bytes, it will receive the 4096 bytes it
requested and the remaining 4096 bytes in the datagram will be discarded
by the UDP protocol layer without further notification to either sender or
receiver.

9.0 Chapter 9. IMS Sockets

This chapter includes information on how you develop IMS applications that
use the IMS sockets feature of IBM TCP/IP for MVS.

We will explain how IMS sockets is implemented in IBM TCP/IP Version 3
Release 1 for MVS and discuss the impact this implementation has on your
application design.

For basic socket programming information, please refer to Chapter 5, "Your
First Socket Program" in topic 5.0.

If you need IMS socket call reference information or information on how
9.1 IMS and TCP/IP Networks

You may roughly divide IMS applications into the following two major groups:

1. IMS applications that communicate with a user based on the traditional IBM 3270 protocol

   The IMS applications typically use the Message Formatting Services (MFS) component of IMS to translate between the 3270 data stream and the record oriented data format, the IMS applications use.

   If the end user is connected via a TCP/IP network, the end user can use TN3270 emulation software to emulate an IBM 3270 terminal. From an IMS point of view, such a terminal is a fully normal IBM 3270 terminal, and no extra software is needed in IMS to support such connections. All existing IBM 3270 based IMS applications can be used from TCP/IP workstations in this way.

2. IMS applications that communicate with another application in a typical client/server fashion

   The programming interfaces used by IMS client/server applications vary and depend on the ability of both the client and the server platform:

   Advanced Program to Program Communication (APPC) is supported by IMS.

   IMS applications may use the Common Programming Interface Communications (CPIC) API for such applications. If the client platform supports the LU6.2 protocols, those protocols offer you many good features that you will not find in, for example, TCP or UDP protocols. Such features are session and conversation security, synchronization levels and conversation state control.

   Message Queuing Interface (MQI) applications are supported by IMS.

   If your applications are fully asynchronous in nature, and both your IMS platform and your partner platform supports MQI applications, this is an easy way to implement client/server applications.

   Distributed Computing Environment / Remote Procedure Call (DCE/RPC) server applications are supported in IMS if you use the MVS/ESA OpenEdition Distributed Computing Environment Application Support Server for IMS (IMS/AS) feature.

   If your partner program resides on a platform that supports the Distributed Computing Environment, DCE/RPC is both a good and strategic choice. Remote Procedure Call APIs are generally easier
to use than native socket APIs, because RPC APIs hide many of the differences in data formats you typically find in a distributed environment. DCE/RPC also includes directory services that will assist your clients in locating a server and integrating security features based on the DCE security implementation.

Native socket APIs are supported in the IMS environment if you install the IBM TCP/IP Version 3 Release 1 for MVS IMS sockets feature.

IMS sockets have been developed to provide access to IMS resources from hosts that only support TCP/IP and the native socket APIs. Currently you will find many platforms that do not support either SNA LU6.2 or DCE/RPC but do support native TCP/IP.

9.2 Overview of IMS Sockets

The IMS socket feature consists of software that enables you to use socket programs in an IMS dependent region. As we defined in "General Socket Program Structure" in topic 3.7, socket programs can fall into the following three categories:

1. Client programs that request services from a server program

Any IMS application can turn itself into a socket client by issuing the proper socket calls for a socket client.

The IMS application can be a Message Processing Program (MPP) or a Batch Message Program (BMP) that has been scheduled by traditional IMS methods.

2. Iterative server programs that process client requests one at a time in a serial fashion

Such a server program will typically be a BMP that is started once and sits around forever to serve client requests serially. From an IMS socket point of view, an iterative server may also be implemented as an MPP; but that is not expected to be a typical implementation because of the region occupancy characteristics of an iterative server program.

3. Concurrent server programs that consist of both a concurrent server main process (the scheduling process) and a number of parallel child processes that process the client requests in parallel

The IMS sockets implementation of a concurrent server in IMS is based on a BMP that runs the concurrent server main process and a number of Message Processing Regions (MPR) that execute the concurrent server child processes. The main process inserts IMS transaction requests to the IMS message queues, and IMS schedules these transactions in the MPRs.

The IMS socket feature includes the following two components:

The IMS listener

The IMS listener is a generic concurrent server main process.

The IMS listener is supplied as a general purpose concurrent server main process. If you have requirements that go beyond the features of
this general purpose implementation, you are able to write your own concurrent server main process and use that instead of the IMS listener.

The IMS assist module

The IMS assist module enables you to develop concurrent server programs that use the normal IMS call API to receive and send data over a socket. This mode of programming is called implicit-mode as opposed to programs that include explicit socket calls, called explicit-mode programs. In an implicit-mode COBOL IMS sockets program, you call CBLADLI instead of the normal CBLTDLI module, but the call syntax is identical. For PLI programs, you use PLIADLI; for assembler programs, you use ASMAIDL, and for C programs you use CADLI. We will use the collective term xxxADLI for all four assist module entry points. Please note that the AIBTDLI programming interface that was introduced with IMS/ESA Version 3.3 is not supported by the IMS sockets assist module.

Client programs and iterative server programs can only be developed as explicit-mode programs. Server programs that are started by the IMS listener may be developed as either explicit-mode or implicit-mode programs.

See Figure 41 for an overview of the structure of IMS sockets.
IMS sockets do not introduce any new programming interfaces. IMS sockets utilize existing APIs:

The C-socket API for IMS sockets explicit-mode applications written in C.

The Sockets Extended call API for IMS sockets explicit-mode applications written in, for example, COBOL or PL/I.

The IMS call API for IMS sockets implicit-mode applications written in any IMS supported programming language.

IMS sockets do not limit your programs to stream sockets (TCP protocols). You may also use datagram sockets (UDP protocols) or even raw sockets (IP protocols). Due to the unreliable nature of both datagram sockets and raw sockets, most IMS socket applications are assumed to use stream sockets. The IMS listener will only accept transaction requests via stream sockets.

Client and iterative server IMS socket applications are not different in design or implementation from any normal MVS client and iterative server socket applications, so we will not describe these in further detail in this chapter. Instead, refer to Chapter 5, "Your First Socket Program" in topic 5.0 and Chapter 7, "Socket Client Programs" in topic 7.0 for information on such programs.

9.3 Concurrent Server in an IMS Environment

It is expected that the major part of your IMS socket applications will be based on the concurrent server implementation using the IMS listener, and this implementation is what the rest of this chapter will focus on.

The following section will define some terms that are used by the IMS sockets implementation:

segment
A segment is a segment of data that is formatted according to IMS message standards, where the first 2 bytes contain the length in binary network byte order of the data including the length bytes:

```
0 2 4
|LL|zz|data |
|__|__|_______ ___|
|<_______LL____ __>|
```

The value of the zz field is not defined; we recommend you initialize it to binary zeroes.

message
A message is a sequence of segments where the last segment is an End Of Message (EOM) segment:

```
0 2
```
A Beginner's Guide to MVS TCP/IP Socket Programming

LL = 4

A message may consist of many segments, terminated by an EOM segment.

| LL | zz | segment 1 data | LL | zz | segment 2 data | 04 | 00 |
|____|____|_______________|____|____|_______________|____|____|
|<_____segment 1_______>|<_____segment 2________>|<EOM>|
See "IMS Listener Security Exit" in topic C.4 for the sample IMS listener security exit.

9.3.2 Remote Client Design Considerations

Use of the IMS listener imposes some design restrictions on your client program:

When the client has established a connection with the IMS listener, it must send a Transaction Request Message (TRM) segment with a layout as expected by the IMS listener.

```
0 2 4 12 20
|LL|zz|*TRREQ*|trancode|optional user-data|
________|________|________|_____________ ___|
<________________LL__________________ __>|
```

The client must be prepared to receive a Request Status Message (RSM) segment from the IMS listener, in case the IMS listener cannot initiate the requested transaction successfully.

```
0 2 4 12 16
|LL|zz|*REQSTS*| rc |reas|
________|____|____|
<________|____|____>
```

LL = 20

The returncode and reasoncode are returned as binary 2 byte fields with the appropriate codes in network byte order.

The possible codes are defined in IBM TCP/IP for MVS: IMS TCP/IP Application Development Guide and Reference, SC31-7186, and by your listener security exit.

If the server program is an explicit-mode server program, the rest of the interactions between the client and the server are left to be designed by you. You are free to implement as many interactions (send / receive) sequences as required by your application. There are no restrictions on format, length or number of interactions.

If the server program is an implicit-mode server program, there are more client design considerations that must be taken into account. Please see "Implicit-mode Server Program" in topic 9.3.4, for information on these additional considerations.

Your client programs must include logic that examines the first received data from IMS to decide if it is an RSM segment that rejects the transaction or if it is valid output from the IMS server program. One technique to simplify this is always to let the server program send a positive RSM segment as the first output. A positive RSM segment can be defined as an RSM segment with rc=0 and reason code=0. In this way the client program will always receive an RSM segment. It is either a
confirmation that the transaction request was successfully started or that it was rejected, and the client program can act accordingly.

When the IMS listener receives a valid TRM segment, it inserts a Transaction Initiation Message (TIM) segment on an IO Program Control Block (IO PCB) in order to let IMS schedule the proper MPP.

All IMS transaction codes that the IMS listener must be able to initiate are defined in a configuration data set that is read by the IMS listener program when it is started. If a client sends a TRM segment with an IMS transaction code that is not defined in this configuration data set, the request will be rejected with a reason code of 1 in the RSM segment.

For every IMS transaction code in the IMS listener configuration data set, there is an attribute associated that tells the IMS listener if the transaction is implemented as an explicit-mode or as an implicit-mode server program. The reason for this attribute is that the IMS listener process is different for explicit-mode and implicit-mode server programs.

### 9.3.3 Explicit-mode Server Program

For an explicit-mode server program, the IMS listener only inserts the TIM on the IMS message queue. When the server MPP is scheduled by IMS, it receives the TIM on its initial Get Unique (GU) on the IO PCB. All the data exchanged between the client and the server program are handled via explicit socket calls in the server.

```plaintext
Client                                           IMS Listener
| Send   |                                  |                |
|TRM ___Transaction Request Message (TRM)___>Receive TRM | 1
|      |                                  |  If any error |
|      |<__Request Status Message (RSM)_________   Reject | 2
|      |(Only sent if TRM is rejected) | else |
|      |                                  |  Givesocket    | 3
|      |                                  |  ISRT TIM ____  | 4
|      |                                  |  _____________|
|      |                                  |  V
|      |____ Transaction                 |
|      |____ Initiation                  |
|      |____ Message via IMS message     |
|      |____ IMS message queues          |
|      |____ IMS explicit mode server__  |
|      |____>GU IOPCB 5                |
|      |  Initapi 6                     |
|      |  Takesocket 7                  |
|Send/ |                                  |  
|Recv <________Free format messages___________>Explicit socket calls 8
|      |                                  |  
|Close |                                  |  Close socket 9 |
```
The sequence numbers in the following explanation are all related to Figure 42.

The sample data structures and coding examples are provided in COBOL.

The IMS listener is started as an IMS Batch Message Program (BMP) that opens a socket and listens on a port that you define for connections from the TCP/IP network and retrieval of Transaction Request Messages (TRM) from TCP/IP clients.

The layout of a TRM without optional user security data is:

```
*---------------------------------------------------------------*
* Transaction Request Message segment                          *
*---------------------------------------------------------------*
01 TRM-message.
   05 TRM-length-ll    pic 9(4) Binary Value 20.
   05 TRM-length-zz    pic x(2) Value low-value.
   05 TRM-id           pic x(8) Value '*TRNREQ*'.
   05 TRM-trancode     pic x(8) Value 'TRANCODE'.
   05 TRM-security-data.
```

If your installation has implemented an IMS listener security exit routine, you pass the required security related data as part of the TRM segment. You do so by extending the segment with your data following the TRM-trancode field. Remember to update the TRM-length-ll field with the correct segment length according to your extension.

2 If a condition that is detectable by the IMS listener prevents it from scheduling the requested IMS transaction, it will send back a Request Status Message (RSM) segment to the client, informing the client of the cause of rejection.

The format of an RSM segment is:

```
*---------------------------------------------------------------*
* Transaction Request Status Message segment                   *
*---------------------------------------------------------------*
01 RSM-message.
   05 RSM-length-ll    pic 9(4) Binary Value 20.
   05 RSM-length-zz    pic x(2) Value low-value.
   05 RSM-reqsts       pic x(8).
     88 RSM-msg        Value '*REQSTS*'.
   05 RSM-return-code  pic 9(8) Binary Value zero.
     88 RSM-OK         Value zero.
     88 RSM-error      Value 8.
   05 RSM-reason-code  pic 9(8) Binary Value zero.
     88 RSM-not-defined Value 1.
     88 RSM-IMS-error   Value 2.
     88 RSM-buffer-full Value 4.
     88 RSM-AIB-error   Value 5.
     88 RSM-tran-unavailable Value 6.
     88 RSM-format-error Value 7.
```

3 If the TRM is accepted, the IMS listener issues a givesocket call, where it gives the socket to the child process.
The IMS listener constructs an address space name and a task ID to be used by the child process. The address space name is constructed according to the `ADDRSPCPFX` keyword value in the listener configuration data set. If this value is for example IL, the address space names generated by the listener will be IL000000 and upwards. The task ID is set to a fixed value of IMSERVER.

This information is passed in the TIM to the child process. If the child process is a Sockets Extended program, it can use these values on its `initapi` call.

4 Based on information in the TRM, the IMS listener initiates IMS transactions by inserting IMS messages, called Transaction Initiation Messages (TIM), over an IMS alternate IO PCB.

5 The concurrent server child processes are scheduled as normal IMS message processing programs that retrieve the TIM on their first Get Unique (GU) on the IO PCB.

```
*---------------------------------------------------------------*
* Transaction Initiation Message segment                        *
*---------------------------------------------------------------*
01 TIM-message.
  05 TIM-length-ll pic 9(4) Binary Value zero.
  05 TIM-length-zz pic x(2) value low-value.
  05 TIM-id pic x(8) value space.
  05 TIM-lstn-name pic x(8) value space.
  05 TIM-lstn-task pic x(8) value space.
  05 TIM-srv-name pic x(8) value space.
  05 TIM-srv-task pic x(8) value space.
  05 TIM-lstn-socketid pic 9(4) Binary value zero.
  05 TIM-tcpip-name pic x(8) value space.
  05 TIM-data-type pic 9(4) value zero.
  88 TIM-ascii value 0.
  88 TIM-ebcdic value 1.
```

```
* Receive TIM from listener                                    *
*---------------------------------------------------------------*
Call 'CBLTDLI' using dli-gu
  iopcb
  TIM-message.
If iopcb-status = 'QC' then
  Move zero to return-code
  Goback.
```

Please note that the TIM message segment includes a half-word where you can test the contents to decide if the client process is an ASCII client or an EBCDIC client. The IMS listener is able to make that decision based on the fixed text (*TRNREQ*) in the TRM.

```
* If client is ascii, translate RSM text to ascii before send   *
*---------------------------------------------------------------*
If TIM-ascii then
  Move 8 to ezacic-len
  Call 'EZACIC04' using RSM-oky
    ezacic-len.
```

6 If the child process is an Sockets Extended application, it starts
with an initapi call, where it identifies itself as a client of the TCP/IP address space named in the TIM (TIM-tcpip-name). It is recommended that the child process identifies itself with the client ID constructed by the IMS listener and passed to the child process in the TIM (TIM-srv-name and TIM-srv-task); but, in the IBM TCP/IP Version 3 Release 1 for MVS implementation, it is actually not a requirement. The IMS listener does not give the socket to a specific client ID but rather to a client ID with a blank address space name and task name. This means that any task that refers to a socket descriptor that has been given by the IMS listener, but not yet taken, will receive it. This is, under normal circumstances, not a big exposure because the time period between the givesocket and the takesocket is normally less than a few hundred milliseconds.

*---------------------------------------------------------------*
* Variables used for the INITAPI call                           *
*---------------------------------------------------------------*
01 soket-initapi    pic x(16) value 'INITAPI'.
01 maxsoc          pic 9(4) Binary Value 50.
01 initapi-ident.
   05 tcpname    pic x(8) Value space.
   05 myjobname  pic x(8) Value space.
01 subtask         pic x(8) value space.
01 maxsno          pic 9(8) Binary Value zero.
01 errno           pic 9(8) binary value zero.
01 retcode         pic s9(8) binary value zero.

*---------------------------------------------------------------*
* Initialize socket API with the values, we got from           *
* the IMS listener.                                          *
*---------------------------------------------------------------*
Move TIM-srv-name to myjobname.
Move TIM-srv-task to subtask.
Move TIM-tcpip-name to tcpname.
Call 'EZASOKET' using soket-initapi
   maxsoc
   initapi-ident
   subtask
   maxsno
   errno
   retcode.
   If retcode < 0 then
      - process error -

When the child process has identified itself, it can issue a takesocket call to take over the socket from the listener.

*---------------------------------------------------------------*
* Variables used by the TAKESOCKET Call                        *
*---------------------------------------------------------------*
01 soket-takesocket pic x(16) value 'TAKESOCKET'.
01 take-from-clientid.
   05 take-from-domain pic 9(8) Binary Value 2.
   05 take-from-name  pic x(8) value space.
   05 take-from-task  pic x(8) value space.
   05 filler         pic x(20) value low-value.
01 errno           pic 9(8) binary value zero.
01 retcode         pic s9(8) binary value zero.
01 socket-descriptor pic 9(4) Binary value zero.

*---------------------------------------------------------------*
* Issue a take-socket with the values we got from              *

7 When the child process has identified itself, it can issue a takesocket call to take over the socket from the listener.
* the IMS listener. *

---------------------------------------------------------------

move TIM-lstn-name to take-from-name.
move TIM-lstn-task to take-from-task.
Call 'EZASOKET' using soket-takesocket
   TIM-lstn-socketid
take-from-clientid
erro
retcode.
If retcode < 0 then
   - process error -
else
   move retcode to socket-descriptor.

On the takesocket call, the child process passes the client ID of the IMS listener as the client ID of the process that gave the socket.

8 The socket is now fully transferred to the child process, and the socket applications may exchange data with each other. When you use explicit-mode, your server program is responsible for applying any required translation to the data it receives or transmits.

9 Finally the client and the server child processes close their sockets and break the connection.

9.3.4 Implicit-mode Server Program

For an implicit mode server program, the IMS listener performs a number of actions in addition to those performed for explicit-mode server programs:

1. It receives all input segments from the client. The client signals the end of input by sending an EOM segment.

2. If the client is an ASCII client (determined by examining the contents of the TRNREQ constant in the TRM), the full input message is translated from ASCII to EBCDIC. This action implies that you must be careful with the design of the implicit mode application messages so that you do not include any non-character data fields except the length field in each segment.

3. The IMS listener then inserts the TIM as the first message segment on the IMS message queue and follows it by all the input segments retrieved from the client program (leaving out the EOM segment).

The reason for doing it this way is that the implicit mode server program retrieves its input via normal IMS GU and Get Next (GN) calls on its IO PCB.

__________________________________________________________________________

Client                                    IMS Listener_____________
|Send  |                                  |                         |
|TRM ___Transaction Request Message (TRM)___>Receive TRM  |
|      |                                  |  If any error           |
|      |<__Request Status Message (RSM)_________  Reject     |
|      |__(Only sent if TRM is rejected)    |  else                  |
|      |                                  |    ISRT TIM ______________|
|      |                                  |    _____>RECV input segment |
|      |                                  |    | Do while not EOM    |
|Send___Input segments in IMS format ___|    | ISRT input segment ___|
For an IMS programmer, it is an easy task to write an implicit mode server program in IMS. No socket calls are used at all. The server program uses only normal DLI calls. The only difference from a traditional IMS program is that you do not call the conventional IMS language interface routines, but instead you call routines which are supplied as part of the IMS socket support (the IMS assist routines). They are identified by module names that are almost identical to those you already know: CBLADLI, PLIADLI, ASMDLI and CADLI. The call syntax is identical to the syntax used with the conventional routines.

The logic in an implicit mode echo server program could look like the following (this example has, for the sake of simplicity, been stripped for error checking logic):

```plaintext
*---------------------------------------------------------------*
* Receive input segments from client and echo them back         *
*---------------------------------------------------------------*
Get-unique.
Call 'CBLADLI' using dli-gu
  iopcblk
  buffer.
If iopcblk-status = 'QC' then
  go to exit-now.
Perform until iopcblk-status not equal space
  Call 'CBLADLI' using dli-isrt
  iopcblk
```

---

Figure 43. Implicit-mode Server Program Initiation

A Beginner’s Guide to MVS TCP/IP Socket Programming
The server program is easy to write for your IMS programmer, but your client programmer must adhere to a set of rules and restrictions that are imposed by the assist routines. Please see Figure 44 for an overview of the processing performed by the assist module.

<table>
<thead>
<tr>
<th>Implicit mode</th>
<th>Assisted Module</th>
<th>IMS Control</th>
<th>IMS Listener</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server Program</td>
<td>Assist Module</td>
<td>IMS CTL</td>
<td>IMS Listener</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>ACCEPT connection</td>
<td>(_) CONNECT</td>
<td>RECV TRM segment</td>
<td>RECV input segment</td>
</tr>
<tr>
<td>RECV input segment</td>
<td>...</td>
<td>RECV EOM segment</td>
<td>GIVESOCKET</td>
</tr>
<tr>
<td>Msg</td>
<td>ISRT TIM segment</td>
<td>ISRT input segment</td>
<td></td>
</tr>
<tr>
<td>Schedule</td>
<td>MPP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Figure 44. IMS Assist Module Process Flow
The assist module acts as an interface between an implicit mode server program and the actual socket programming interface.

The design restrictions that apply to a client program that communicates with an implicit-mode server program are:

There can only be one interaction per IMS transaction. Your client must send all input data before it turns around and issues the first read for output data from the IMS transaction. No continued dialog between the client and server process is possible. IMS conversational transaction mode is not supported.

Your client program must send data in the required format, which is 2 bytes segment length followed by two bytes binary zero followed by your input data. In the following example, 20 bytes of user data is sent to the server:

```
01 Message-segment.
  05 Segment-length-ll pic 9(4) Binary value 24.
  05 Segment-length-zz pic xx value low-value.
  05 Segment-data pic x(20)
    Value 'Data segment 1'.
```

The client program may send more succeeding segments of variable length input data. The last segment must always be an End Of Message (EOM) segment:

```
01 EOM-message-segment.
  05 filler pic 9(4) Binary value 4.
  05 filler pic xx value low-value.
```

The total length of the input message must not exceed 32K.

If your client program is running on an ASCII host, all data in all input segments are translated from ASCII to EBCDIC, and all data in all output segments from the server program is translated from EBCDIC to ASCII. You had better ensure that all data exchanged between the client and the server is text data, otherwise unexpected results might occur (or will certainly occur).

When the IMS server program inserts output segments to the client on its IO PCB, the assist module accumulates the output into a buffer, which has a maximum size of 32K, but the assist module does not send anything over the socket connection until the server program signals a commit point (by means of a new GU on the IO PCB). At this time, the assist module sends each accumulated output segment in the same format as the input segments. When the last segment has been sent, the assist module generates an EOM segment and sends it to the client. All output segments are sent directly over the socket connection, so the IMS message queues are not used for output data.

The assist module then passes the GU call to the real DLI language interface routines, which perform the real IMS commit and optionally returns a new input message. If the GU is successful (either passing a new transaction or returning a QC status code), the assist module finishes the previous transaction by sending a Completed Status Message (CSM) segment to the client, and the socket is closed.

Please see "IMS Recovery Considerations" in topic 9.5 for information on unsuccessful DL/I calls and recovery considerations.
**9.4 Dual-purpose IMS Programs**

An IMS MPP that uses MFS to communicate with IBM 3270 terminals interfaces with MFS using record formats as defined in an MFS message input descriptor (MID) and an MFS message output descriptor (MOD). If you develop your socket client so it bases its interface to the IMS MPP on the exact same formats as defined in the MPP's MID and MOD, the same MPP may be used concurrently with IBM 3270 terminals and socket clients.

---

![Diagram](image-url)

**Figure 45. Dual-purpose IMS Program Input/Output Flow**

It is easiest to develop the socket client if the MPP uses MFS formatting option 1, but both formatting option 2 and 3 may be used. Option 2 and 3 do add some complexity to the socket client program, as it must emulate the MFS processing done for each formatting option.

The IMS socket interface does not pass a MOD name back to a socket client.
As long as the MID - MOD sequence is predetermined via the NEXT keywords in the MFS format set, this is not a problem. If your MPP changes the next MOD name dynamically, the socket client will probably need that information in order to apply the correct interpretation to the segments it receives from the IMS MPP. If the decision cannot be made based on segment length or content, you may have to add code to the MPP that imbeds the MOD name in a segment sent to the socket client. The MPP is able to detect if input is from a socket client or from an IBM 3270 terminal by looking at the input segment ZZ field. If input is from an IBM 3270 terminal, it contains the MFS formatting option used for the segment. You can design your socket clients so that they send a ZZ value of zero and use that value to detect that input is from a socket client. When your MPP needs to change the MOD name, you can add code that adds a special output segment to the socket output stream with the MOD name imbedded. Such a segment could, for example, be:

```
0 2 4 12
|LL|zz|*TRNMOD*|modname |
|__|__|________|________|
```

Based on your MPP logic, this segment will only be sent to a socket client, and the socket client uses it to determine that the following segment is formatted according to the MOD name passed in the preceding *TRNMOD* segment.

To use an existing MPP as a dual-purpose program, you must change all DL/I calls from xxxTDLI to xxxADLI. xxxADLI is the IMS sockets assist module. It will act as the normal xxxTDLI module if input is not from a socket client. If your existing MPP uses the AIBTDLI interface, the call syntax must be changed to the older xxxTDLI calls.

Please see "Dual Purpose Implicit Mode IMS Server Program" in topic C.1, for a sample COBOL based dual-purpose IMS program.

9.5 IMS Recovery Considerations

There are some issues with the IMS implementation of sockets that you must be aware of when you plan to use IMS sockets.

1. In the IBM TCP/IP Version 3 Release 1 for MVS implementation of IMS sockets, a socket is not a recoverable IMS resource, and the messages exchanged over a socket are not recoverable because they are not passing through the IMS message queues. The only IMS recoverable message in this context is the TIM segment, which is being inserted by the IMS listener as a normal IMS message segment. When an MPP is scheduled, the TIM is read and a socket is taken from the IMS listener. For an implicit mode server the input segments from the client is also passed to the server via the IMS message queue; so, for an implicit mode server, both the TIM and the input message are IMS recoverable resources. IMS may pseudo abend an MPP after it has taken a socket and rescheduled it. When the MPP is rescheduled, possibly in another MPR, the TIM is reread on the first GU (because the TIM is recovered by IMS), but the socket is no longer available for a takesocket call, and the takesocket call will fail.
An explicit mode program can test the socket return code and error number fields and take appropriate action, but an implicit mode program is supposed to use the standard IMS call interface, so it does not see any socket return codes or error numbers. If an underlying socket call results in an error situation, the status must be returned to the implicit mode program via the standard IMS IO PCB status code field. The IMS IO PCB may be defined as follows:

```
*---------------------------------------------------------------*
* Input-Output PCB layout                                       *
*---------------------------------------------------------------*
01 iopcb.
  05 iopcb-lterm       pic x(8).
  05 iopcb-assist-status-bin pic s9(4) comp.
  05 iopcb-assist-status-char redefines
      iopcb-assist-status-bin pic x(2).
      88 iopcb-assist-aib-error value 'EA'.
      88 iopcb-assist-buffer-full value 'EB'.
      88 iopcb-assist-tim-only value 'EC'.
  05 iopcb-status      pic x(2).
      88 iopcb-dli-stop value 'QC'.
      88 iopcb-dli-ok   value ' '.
      88 iopcb-assist-error value 'ZZ'.
  05 iopcb-cdate       pic s9(7) comp-3.
  05 iopcb-ctime        pic s9(7) comp-3.
  05 iopcb-input-msgno pic 9(8) binary.
  05 iopcb-output-mod  pic x(8).
  05 iopcb-userid      pic x(8).
```

If a socket interface error occurs, an IO PCB status code of ZZ will be returned to the implicit mode application. More information about the socket error is located in the two reserved bytes of the IMS IO PCB that follows the LTERM name.

2. You must also be aware that IMS sockets do not assist you in synchronizing updates done by your IMS socket server and socket client. The assist module does send back the CSM segment at a point in time where the IMS resources have been committed by IMS, but this is not a two-phase commit protocol that can be used to ensure synchronization of updates. IMS will never know if the client was able to commit or not commit its resources following the receipt of the CSM segment.

3. If the transaction code in the TRM segment is unavailable (either not defined to IMS or temporarily stopped), the IMS listener sends back an RSM segment with proper return codes. We recommend that you always include code in your client programs that are able to deal with an RSM segment, and interpret the return code in order to inform the user of the reason for rejection. When the IMS listener rejects a transaction, it does write out a short message on SYSPRINT in the listener BMP address space. You may be able to look at that with SDSF.

### 10.0 Chapter 10. CICS Sockets

This chapter explains how CICS sockets are implemented and how you can use CICS sockets to implement a concurrent socket server as CICS transaction programs.

For a more detailed explanation of CICS sockets, you may read CICS/ESA and TCP/IP for MVS Socket Interface, GG24-4026. It was written for IBM TCP/IP
Version 2 Release 2 for MVS but is still a good introduction to CICS sockets.

For basic socket programming information, please refer to Chapter 5, "Your First Socket Program" in topic 5.0.

If you need CICS socket call reference information or information on how to customize the CICS socket feature, please see IBM TCP/IP for MVS: CICS TCP/IP Socket Interface Guide and Reference, SC31-7131, and MVS TCP/IP V3R1 Implementation Guide, GG24-3687.

10.1 CICS and TCP/IP Networks
10.2 Overview of CICS Sockets
10.3 Concurrent Server in a CICS Environment
10.4 Link Editing CICS Socket Programs

10.1 CICS and TCP/IP Networks

CICS applications may be divided into the same major groups as we used for IMS applications:

1. The first major group is CICS applications that communicate with a user based on the traditional IBM 3270 protocol.

   Such CICS applications typically uses the Basic Mapping Support (BMS) facilities of CICS to translate between the 3270 data stream and the record oriented format that is expected by a CICS application.

   Users in a TCP/IP network may use this type of CICS applications if their workstation has TN3270 emulation software.

2. The second major group is CICS applications that communicate with another application in a typical client/server fashion.

   Like IMS, CICS supports a range of programming interfaces to be used by client/server applications:

   Advanced Program to Program Communication (APPC) is supported by CICS, including the Common Programming Interface for Communications (CPI-C).

   Message Queuing Interface (MQI) applications are supported by CICS.


   CICS intercommunication facilities enables you to use standard CICS functions across CICS systems implemented on different platforms, for example, CICS/ESA, CICS OS/2 or CICS/6000. The CICS intercommunication facilities include:

   - Function shipping
   - Distributed program link (DPL)
Asynchronous processing
- Transaction routing
- Distributed transaction processing (DTP)

We refer you to the relevant CICS documentation for details on these very powerful facilities.

If you develop client/server applications for platforms that support CICS, the CICS intercommunication facilities offer you a consistent way to implement your application across a number of platforms.

Native socket APIs are supported in CICS, if you install the IBM TCP/IP Version 3 Release 1 for MVS CICS sockets feature.

If your partner programs are located on hosts that only support TCP/IP and the native socket APIs, the CICS sockets feature is your choice for creating CICS client/server applications.

10.2 Overview of CICS Sockets

The CICS sockets feature consists of software that enables you to use TCP/IP socket programs in a CICS task:

1. Client programs that request service from remote servers

   Any CICS task may turn itself into a socket client program by issuing the proper client socket calls.

2. Iterative server programs that process client requests one at a time in a serial fashion

   In a CICS environment, an iterative server will be considered a long-running CICS task. Such a server task is typically started when CICS is started and keeps running until CICS is shut down.

3. Concurrent server programs

   In the CICS environment, the concurrent server main process will be a long-running CICS task that accepts connection requests from the network and initiates child processes by issuing EXEC CICS START commands. The child processes execute as normal short CICS tasks.

See Figure 46 for an overview of how the different application types are implemented in a CICS environment.
Figure 46. CICS Socket Application Overview

The CICS sockets feature includes the following components:

The CICS listener

The CICS listener is a general purpose concurrent server main process. The name of the program is EZACICO2, and it is started by the CSKL CICS transaction code. When you enter the CSKE transaction code to enable the task related user exit, the transaction code CSKL is started by the enable program.

A CICS adapter that provides an interface between CICS tasks and the TCP/IP system address space.

The CICS adapter consists of four components:

1. The first is a stub module that must be link edited with any CICS program that uses socket functions. The stub module is called EZACICAL.

2. The second is a CICS task related user exit (TRUE) that acts as the interface between a CICS task, which uses socket functions, and the TCP/IP communicating subtasks in the CICS address space. The name of the task related user exit is EZACICO1.

3. Every time a CICS task issues its first socket call, a companion MVS subtask is started in the CICS address space. This subtask handles the actual socket communication between the CICS address space and the TCP/IP system address space. When the CICS task terminates, the companion MVS subtask is terminated too. The name of the module that executes in these subtasks is EZACICO3.

4. The last is a set of administrative routines that are used to enable and disable the CICS sockets task related user exit function. CICS transaction code CSKE is used to enable the task related user exit, and transaction code CSKD is used to disable it.

Figure 47 shows an overview of the CICS adapter components.
You may develop your CICS sockets programs using one of the following IBM TCP/IP for MVS socket APIs:

The C-socket API for C based CICS programs (not all C socket calls are supported in the CICS sockets environment).

The Sockets Extended call API for programs written in, for example, COBOL or PL/I.

The IBM TCP/IP Version 2 Release 2 for MVS CICS sockets call API. This call API is supported by IBM TCP/IP Version 3 Release 1 for MVS for compatibility purposes. We recommend that you use the Sockets Extended API for development of new CICS sockets applications.

The CICS sockets feature supports stream sockets (TCP protocols) and datagram sockets (UDP protocols), but not raw sockets.

Client and iterative server CICS sockets applications do not differ from
similar programs in a native MVS environment, so we will not go into
detail with those in this chapter. The concurrent server implementation
is specific to the CICS environment, and we will discuss that in more
details in the sections that follow.

10.3 Concurrent Server in a CICS Environment

IBM TCP/IP for MVS supplies a generic concurrent server main process
called the **CICS listener**. The CICS listener is generic in the sense that
it acts as a CICS transaction scheduler that receives Transaction Request
Messages (TRM) from the TCP/IP network.

The CICS listener is implemented as a long-running CICS transaction (CSKL)
that is started when you enable the socket programming interface in CICS
via the **CSKE** CICS transaction.

The CICS listener uses **EXEC CICS START** commands to schedule new
transactions in CICS. The transaction to start is derived from fields in
a predefined layout of the Transaction Request Message, which a TCP/IP
client sends to the CICS listener over a socket connection.

The format of the TRM for the CICS listener is shown in **Figure 48**.
The CICS listener TRM is variable in length, but the listener always issues a `recv` call for 50 bytes, which is the maximum allowed length of a TRM. If your client sends a TRM of, for example, four bytes and then goes on pushing data to the child server onto the stream, some of this data may be returned to the CICS listener and may be lost. We recommend that you always either send a 50 byte TRM message holding your actual TRM data padded with spaces up to the 50 byte limit, or that your remote clients always issue a `receive` call immediately after having sent the TRM in order to flush the TCP send buffer (See "Streams and Messages" in topic 5.8.1 for details on this technique).

Each field in the TRM is separated by a comma. The absence of a field for example, the client data field is signalled by two successive commas: `TRN,,IC,113015`.

<table>
<thead>
<tr>
<th>tran</th>
<th>This is the CICS transaction code you want to start.</th>
</tr>
</thead>
<tbody>
<tr>
<td>client-data</td>
<td>You can optionally include up to 35 bytes of data, but it might be a more clean design to reserve these 35 bytes for other purposes. You could for example define a standard for inclusion of security related information in the transaction request message. You can write a security exit to be included in the CICS listener. This exit will receive the 35 bytes of user data, and can make user verification decisions based on this. Please see CICS/ESA and TCP/IP for MVS Sockets Interface, GG24-4026, for recommendations on how you can implement a listener security exit in CICS.</td>
</tr>
<tr>
<td>IC or TD</td>
<td>IC or TD means Transient Data or Interval Control. If you specify IC, then you can specify the interval time following (hhmmss) as the last part of the TRM.</td>
</tr>
</tbody>
</table>

To have the CICS listener start, for example, CICS transaction TPIT immediately and without any client-data, the client would connect via the socket interface to the CICS listener and send the following TRM to the listener:

```
TPIT (padded with blanks up to 50 bytes)
```

If the user security exit accepts the transaction request, the CICS listener will issue a `givesocket` and an `EXEC CICS START` of the TPIT CICS transaction.

Under normal circumstances, the CICS listener will not send data back to the client over the socket connection. However, if the CICS listener is unable to start a CICS transaction, either because of validation errors or because the user security exit rejected the transaction, it will send back a 72 byte long error message to the socket client in the following format:

```
TCPCICSERR: specific error text (padded with blanks to 72)
```
The CICS listener will translate the error message to ASCII if the client sent a TRM in ASCII. Good programming practices recommend that you include logic in your client program to test for the fixed text **TCPCICSERR** in the first 10 bytes of the first message it receives from the server side and act accordingly.

The CICS listener gives the socket to a process in the same address space as itself, but it does not give it to a specific task id. The implication of this is that the CICS transaction must start in the same CICS address space as the one where the listener is executing.

On the **EXEC CICS START** command in the CICS listener, a Transaction Initiation Message (TIM) is passed to the started CICS transaction. The TIM includes the following information:

1. It includes the socket descriptor, which was returned to the listener on an `accept` call and which was given via a `givesocket` call.

2. It includes 8 bytes with the CICS address space name, where the listener is executing.

3. It includes another 8 bytes with the subtask id of the listener.

4. It includes the 35 bytes of client data that can be included in the transaction request. The security user exit may optionally exclude this information from the area passed to the CICS transaction.

5. Finally it includes a socket address structure for the client that initiated this request.

    *---------------------------------------------------------------*
    * Transaction Initiation Message from CICS listener            *
    *---------------------------------------------------------------*

    01 TIM.  
      05 give-take-sd pic 9(8) Binary.  
      05 lstn-asname pic x(8).  
      05 lstn-subtask pic x(8).  
      05 client-in-data pic x(35).  
      05 filler pic x(1).  
      05 sockaddr-in.  
        10 sin-family pic 9(4) Binary.  
        10 sin-port pic 9(4) Binary.  
        10 sin-addr pic 9(8) Binary.  
        10 sin-zero pic x(8).  

When the transaction is scheduled by CICS, it must first retrieve the TIM from CICS.

    *---------------------------------------------------------------*  
    * Retrieve Transaction Initiation Message from Listener          *  
    *---------------------------------------------------------------*  

    move 72 to cleng.  
    exec cics retrieve  
      into(TIM)  
      length(cleng)  
    end-exec.  

The server program may then issue an `initapi` call to identify itself.
* initapi parameters

*---------------------------------------------------------------*
01 errno pic 9(8) binary value zero.
01 retcode pic s9(8) binary value zero.
01 init-maxsoc pic 9(4) Binary value 10.
01 init-ident.
   05 init-tcpname pic x(8) value 'T18ATCP'.
   05 init-asname pic x(8) value space.
01 init-subtask.
   05 init-cics-task pic 9(7).
   05 filler pic x value 'I'.
01 init-maxsno pic 9(8) Binary value zero.

*---------------------------------------------------------------*

* Initialize socket API

*---------------------------------------------------------------*
move space to init-asname.
move eibtaskn to init-cics-task.
call 'EZASOKET' using soket-initapi,
   init-maxsoc
   init-ident
   init-subtask
   init-maxsno
   errno
   retcode.
if retcode < 0 then
   move 'Initapi failed' to cics-msg-area
   perform write-cics thru write-cics-exit
   go to pgm-exit.

The concurrent server child program may, instead of an initapi call, start out directly with a takesocket call. If the server program starts with the takesocket call, the CICS sockets interface will assign default values to the client ID. The address space name will be set to the CICS address space name, and the subtask ID will be set to the CICS task number suffixed with the letter T. The maximum number of sockets that will be available for a program, that does not issue the initapi call, is 50.

The takesocket call parameters are based on the TIM values. The server program must take the socket from the client ID passed in the TIM fields: LSTN-ASNAME and LSTN-SUBTASK. The socket descriptor to take is passed in the TIM field: GIVE-TAKE-SD.

*---------------------------------------------------------------*

* Takesocket parameters

*---------------------------------------------------------------*
01 soket-takesocket pic x(16) value 'TAKESOCKET '.
01 sockid pic 9(4) binary.
01 errno pic 9(4) binary value zero.
01 retcode pic s9(8) binary value zero.
01 clientid-lstn.
   05 cid-domain-lstn pic 9(8) binary.
   05 cid-name-lstn pic x(8) value space.
   05 cid-subtask-lstn pic x(8) value space.
   05 cid-res-lstn pic x(20) value low-value.

*---------------------------------------------------------------*

* Take socket from CICS Listener

*---------------------------------------------------------------*
move sin-family to cid-domain-lstn.
move lstn-asname to cid-name-lstn.
move lstn-subtask to cid-subtask-lstn.
move low-value to cid-res-lstn.
move give-take-sd to sockid.
call 'ezasoket' using soket-takesocket
   sockid
   clientid-lstn
   errno
   retcode.

if retcode < 0 then
   move 'Take socket error' to cics-msg-area
   perform write-cics thru write-cics-exit
   go to pgm-exit
else
   move retcode to sockid
end-if.

After your CICS program has taken the socket from the CICS listener, it will, from a socket program point of view, act as any other socket program in MVS. It may enter a number of receive/send sequences and will finally close the socket and call the termapi function.

If your client process may run on non-EBCDIC platforms, you must remember to include in your message design a way for the server to detect that the client sends and expects data in ASCII.

As for any CICS program, you should try to avoid long conversations with the end-user. The CICS task ties up resources for other CICS tasks while it is active. From a CICS resource point of view, a design, where your socket client starts a number of short consecutive CICS transactions, will be better than a design where your socket client starts one CICS transaction that stays active for a longer period. The issues are well-known to most CICS programmers. Try to avoid conversational transactions and base your design on some kind of pseudo-conversational implementation instead.

You can use the supplied CICS listener function as it is, or you can of course also write your own listener application, which basically serves the same purpose as the supplied CICS listener.

10.4 Link Editing CICS Socket Programs

When you develop your CICS socket programs, you may use either the EZACICAL call interface or the EZASOKET call interface that is supplied with IBM TCP/IP Version 3 Release 1 for MVS. You may even mix calls to the two interfaces in the same program.

When you link edit your CICS sockets program, you must always explicitly include the EZACICAL module even if you do not call EZACICAL. The EZACICAL module will resolve all external references to EZASOKET for CICS socket programs, in addition to bringing in the proper CICS socket interface code.

Please see "COBOL Compile JCL Procedure" in topic I.2 and "Link/Edit JCL Procedure" in topic I.4 for sample compile and MVS binder JCL for a COBOL language CICS socket program.

Note: Be aware that, if your CICS sockets program only uses the Sockets Extended API (calls to EZASOKET), a link edit step without specific inclusion of EZACICAL will give a returncode of zero, but the EZASOKET code included will not be the CICS sockets version. When you execute the
CICS program, it may seem to work, but each socket call will put the CICS main task TCB into an MVS wait, which is not to be recommended.

11.0 Chapter 11. Debugging and Tracing Socket Programs

This chapter will include information on the techniques you have available for debugging socket applications.

We will recommend some programming practices that will provide you with accurate information in exception situations, and we will introduce relevant tracing facilities in IBM TCP/IP Version 3 Release 1 for MVS.

11.1 Exception Handling
11.2 Application Trace Facilities
11.3 TCP/IP Packet Trace
11.4 IUCV Socket API Trace Function

11.1 Exception Handling

All socket calls return some kind of status information. Most calls return a socket interface return code (RETCODE) and an error identification number (ERRNO).

The return code can have one of the following values on return from a socket call:

-1 The socket call was unsuccessful. The ERRNO field should be examined to determine the cause of the error.

0 The socket call was successful.

>0 The socket call was successful. The value returned in RETCODE is call specific. For read and write type socket calls, the value informs you of how many bytes were actually read or written on this call. Other calls that may return a positive return code are:

- **accept** The value returned is the new socket descriptor number.
- **fcntl** For a query call, a return code of four means that the socket is in non-blocking mode.
- **select** A positive return code represents the number of ready sockets in the select masks.
- **socket** A return code of zero or above, represents the new socket descriptor.
- **takesocket** A return code of zero or above, represents the new socket descriptor.

For some calls a RETCODE of -1 may be acceptable, and the situation must be handled by the program. An example of such a call is the **connect** call that returns a RETCODE of -1 (and an ERRNO value of EADDRNOTAVAIL or ETIMEDOUT), when a connect request to an IP address fails. If the host that the program tries to connect to has more network interfaces, the program can retry the connect with the next IP address in the host entry structure.
Another example is socket calls for sockets that are in non-blocking mode. If the call had been in blocking mode and the call because of this blocking mode would have blocked, the non-blocking call will instead return a RETCODE of -1 and an ERRNO value of EWOULDBLOCK.

We strongly recommend that you include logic after each socket call to filter out acceptable RETCODE and ERRNO combinations, and process these as appropriate. All unacceptable combinations should, as a minimum, result in logging of an error message and proper logic to clean up any sockets that were left in an uncertain state. This most often means: closing the socket.

In a C program you can print out a socket error message by using the tcperror() routine.

```c
if (send(sd, buf, sizeof(buf), 0) < 0)
    tcperror("Write returned error");
    s=close(sd);
    exit(8);
}
```

A corresponding routine does not exist for the other socket programming interfaces. If you use one of these, you will have to create a similar routine as part of your application.

```
01 ezaerror-msg.
  05 filler pic x(9) Value 'Function='.
  05 ezaerror-function pic x(16) Value space.
  05 filler pic x(9) Value ' Retcode='.
  05 ezaerror-retcode pic ---99.
  05 filler pic x(9) Value ' Errorno='.
  05 ezaerror-errno pic zzz99.
  05 filler pic x(1) Value ' '.
  05 ezaerror-text pic x(50) Value ' '.

01 ezaerror-retcode.
  05 ezaerror-function pic x(16) Value space.
  05 ezaerror-retcode pic ---99.
  05 filler pic x(9) Value ' Errorno='.
  05 ezaerror-errno pic zzz99.
  05 filler pic x(1) Value ' '.
  05 ezaerror-text pic x(50) Value ' '.
```

```
Call 'EZASOCKET' using soket-write
    socket-descriptor
    send-request-remaining
    send-buffer-byte(send-request-sent + 1)
    errno
    retcode
If retcode < 0 then
    move soket-write to ezaerror-function
    move 'Write call failed' to ezaerror-text
    move errno to ezaerror-errno
    move retcode to ezaerror-retcode
    display ezaerror-msg
Call 'EZASOCKET' using soket-close
    socket-descriptor
    errno
    retcode
    move 8 to return-code
    goback
```
You may have to use different techniques for printing the error message depending on your runtime environment. For a CICS program, you may want to direct the error message to a CICS transient data queue:

```
*---------------------------------------------------------------*
* Write out an error message to CSMT                           *
*---------------------------------------------------------------*
exec cics writeq td
  queue('CSMT')
  from(ezaerror-msg)
  length(ezaerror-msg-len) nohandle
end-exec.
```

Refer to IBM TCP/IP for MVS: Application Programming Interface Reference, SC31-7187, Appendix B for a complete list of error codes that may be returned by each of the socket programming interfaces.

### 11.2 Application Trace Facilities

If you include proper logic to both deal with exception situations and log error information, you will have a good chance of identifying the cause of most problems your programs may encounter.

During the development phase of new socket applications, it has proven to be useful to include logic in your programs that will actually log tracing information for both successful and unsuccessful socket calls. If you do include such logic, base your tracing logic on some global switch so that you can turn tracing on or off either by passing a runtime parameter to the program when it starts or by setting a constant in the programs static working storage and recompile it.

If you encounter problems, which cannot be identified by your application trace facilities, you have a couple of tracing options you can use within the TCP/IP product.

Tracing within the TCP/IP product can take place at a number of different levels. We recommend that you limit yourself to the following two of these levels:

- **A trace of the IP packets that are received or transmitted over a network interface**
  
  If you suspect that you have a problem with the contents of data you receive or send out or that there is a problem with the sequence of data you receive, the packet trace is likely to reveal those problems to you.
  
  We recommend that you start with this level of tracing. The tracing operation is easy to perform, and the amount of trace data can be controlled via the parameters that are passed to the packet trace function of TCP/IP.

- **A trace of the IUCV based socket API (these are C-sockets, Sockets Extended and REXX sockets)**
  
  If the socket calls you execute in your program do not result in any IP packets being exchanged over the network, this level of tracing can be necessary to identify the source of your problem.
  
  This is very detailed and complicated to perform. We recommend that you only use this trace in extreme cases, where all other methods of
Both of the traces can only be performed by system personnel. They are initiated via OBEYFILE TCP/IP commands. The OBEYFILE command can only be executed from specially authorized users on your MVS system.

### 11.3 TCP/IP Packet Trace

Packet tracing captures IP packets as they enter or leave the device drivers, which are part of TCP/IP for MVS. A packet trace shows you the actual IP packets that are exchanged over the IP network. You can analyze IP and TCP or UDP headers as well as your own application data.

The tracing function is implemented in the TCP/IP address space for those device drivers that are part of the TCP/IP address space in the SNALINK LU0 and SNALINK LU6.2 address spaces for the SNALINK devices and finally in the X.25 address space for the X.25 device driver.

You select what you want to trace via the PKTTRACE command, which is passed either to the TCP/IP address space via an OBEYFILE command or to the other device driver address spaces via an MVS console modify command.

The trace data is collected by MVS Generalized Trace Facility (GTF). You must start a GTF collection address space before you start the actual packet trace function in TCP/IP:

```
//GTFTCPIP PROC MEMBER=GTFPARM

//IEFPROC EXEC PGM=AHLGTF,PARM='MODE=EXT,DEBUG=NO,TIME=YES',
// REGION=2280K,DPRTY=(15,15)
//IEFRDER DD DSN=TCPIP.V3R1.GTF.TRACE,DISP=SHR
//SYSLIB DD DSN=TCPIP.PROCLIB(&MEMBER.),DISP=SHR
```

The PKTTRACE command is used to start the packet trace function in TCP/IP:

```
PKTTRACE CLEAR
PKTTRACE PROT=TCP IP=9.67.56.18 DSTPORT=9997 SRCPORT=9997
TRACE PACKET
```

You can limit the packet trace to certain source or destination ports using a specific protocol on a certain IP address. Please refer to IBM TCP/IP for MVS: Customization and Administration Guide, SC31-7134, for
details about the PKTTRACE command.

In the above example all packets that come from port number 9997 or go to port number 9997 on this host are traced if they come from or go to the remote host with IP address 9.67.56.18. This example was used to trace the packets that were exchanged between a server application bound to port 9997 on our host and a client running on the 9.67.56.18 host.

After you have started the tracing functions, you execute your application programs. If your own program produces its own tracing output, be sure to save this output so that you will be able to correlate it with the packet trace output.

You stop the packet trace again via another OBEYFILE command:

NOTRACE PACKET

After you have stopped the packet trace function in TCP/IP, you can stop the GTF collection address space, and format the GTF trace data set with a TCP/IP utility program called TRCFMT, or you can format the GTF trace data set using your normal IPCS or AMDPRDMP program, which will invoke the TCP/IP formatting routines for the packet trace records.

```bash
//jobname JOB 1,pgmname,CLASS=A,MSGCLASS=X,NOTIFY=tsouser
//TRACE EXEC PGM=IKJEFT01
//FMTIN DD DSN=TCP/IP.V3R1.GTF.TRACE,DISP=SHR
//FMTOUT DD SYSOUT=* 
//SYSTSPPRT DD SYSOUT=* 
//SYSTSIN DD *
TRCFMT PRINT=EBCDIC 
/*
```

The FMTIN DD statement identifies the trace data set that you specified on the GTF collection address space JCL.

The formatting routines format the protocol headers and dumps the user data area in hexadecimal in either EBCDIC or in ASCII translation depending on your TRCFMT options.

You can request that TRCFMT formats the packet trace not for print but for download to a Sniffer Network Analyzer or a DataqLANce* Network Analyzer, if you prefer to use that instead of a printed report.

Figure 50 shows an example of an IP packet that has been formatted by the TRCFMT formatting program.

---

**PKT** 0000004 **DATE=95/02/28** **TIME=12:12:02.699893**
**FROM** LINK=IUCLM18A **DEV=IUCV**
**IP** **SRC=9.67.56.18** **DST=9.67.56.81**
**VER=4** **HDLEN=5** **TOS=X'00'** **TOTLEN=576** **ID=22670** **FLAGS=B'000'**
**FRAGOFF=0** **TTL=60** **PROTOCOL=TCP** **CHECKSUM=X'A141'**
**TCP** **SRC=1031** **DST=9997** **SEQ=903654777** **ACK=898100877**
**WINDOW=28672** **CHECKSUM=X'8D9F' **URGPRTR=0** **ACK**
**DATA LEN=536**
0000 F0F0F0 F1404040 F0F0F0F0 F2404040 F0F0F0F0 F3404040 F0F0F0F0 F4404040 F0F0F0F0 F5404040 F6404040 F0F0F0F0 F7404040 F0F0F0F0 F8404040 F0F0F0F0 F9404040 F0F0F0F1 F0404040 F0F0F0F1 F1404040 F0F0F0F1 F2404040 F0F0F0F1 F3404040 F0F0F0F1 F4404040 F0F0F0F1 F5404040
---

A Beginner's Guide to MVS TCP/IP Socket Programming 141
Figure 50. Sample Packet Trace Output

Each IP packet is printed. The packet number, since the start of the trace and the absolute timestamp, is included on each packet so you can calculate elapsed time between packets.

The IP header section contains formatted information from the IP header. In this section you find the source and destination IP address of the packet, and you find information on the underlying protocol (in this example it is a TCP segment that is contained within this IP packet).

In the TCP header section you can see the source and destination port numbers.

This IP packet is a 536 bytes long TCP segment that is sent from port 1031 on IP host 9.67.56.18 to port 9997 on IP host 9.67.56.81.

The samples in Figure 51 to Figure 53 show you a successful TCP connection setup (the so-called three-way handshake sequence).

The client application is located on host 9.67.56.18 and the server application on host 9.67.56.81 port 9997.

Figure 51. Packet Trace of TCP Connection: SYN Segment

The client TCP protocol layer starts the TCP connection setup when the client application issues a `connect` socket call. The first TCP segment sent is a SYN segment, where the client host advertises its receive TCP window size and the maximum TCP segment size it is prepared to receive. In this example, the server host may send segments of maximum 536 bytes to the client host, and the client host opens a TCP window of 28672 bytes.

The client host chooses its initial sequence number (ISN) for this connection. This initial value depends on TCP implementation and the amount of time elapsed since the start of TCP/IP on this host. In this example, the initial value chosen by the client TCP/IP host is 903654776.

Figure 52. Packet Trace of TCP Connection: ACK Segment

The server responds with an acknowledgement segment (ACK) that contains the sequence number of the last received segment. It also sends its own initial sequence number.

The client application sends the second TCP segment, which is the ACK segment. This segment contains the sequence number of the last received segment and the acknowledgement number of the next expected segment.

Figure 53. Packet Trace of TCP Connection: Data Segment

The third segment is a data segment, which contains the payload of the application. In this example, it contains sequence number 903654777.

The second ACK segment is sent, which acknowledges the receipt of the data segment. It also contains the sequence number of the next expected segment.
The server host responds with a TCP segment where both the ACK and the SYN flags are set. This segment acknowledges the FIN segment sent from the client by returning an ACK sequence number of 903654777, which is the ISN sent by the client plus one, as the SYN segment itself is defined to consume one sequence number. The server host chooses its initial sequence number as 898100876 and advertises its current receive TCP window size and a maximum TCP segment size of 536 bytes.

The last segment exchanged in this connection setup sequence is an ACK segment from the client acknowledging the SYN + ACK segment from the server. The TCP connection is now established and the two socket applications may begin to exchange data over the socket connection.

For further analysis of packet trace output, we refer you to two excellent books by W. Richard Stevens: TCP/IP Illustrated Volume 1 by W. Richard Stevens, SR28-5586, and TCP/IP Illustrated Volume 2 by Gary R. Wright and W. Richard Stevens, SR28-5630.

11.4 IUCV Socket API Trace Function

The packet trace component sends its trace records to GTF. This is not the case with the IUCV socket trace. The socket trace data is written in formatted form directly to an output data set allocated to the TCP/IP address space. By default the TCP/IP address space will write the trace data to a SYSDEBUG DD statement, but you can modify this dynamically via an OBEYFILE command.

The main issue, when you want to use the socket API trace, is that you cannot limit the trace in any way. When you start the socket API trace, all socket activity on your TCP/IP system is traced. You have the following two ways in which you can handle this:

1. You quiesce your TCP/IP system so that all activity except your test application is brought to a halt. Depending on your environment, this
2. You implement a secondary TCP/IP stack on your MVS system with an IUCV link connecting it to your primary TCP/IP system. This secondary TCP/IP system can run completely stripped of anything else, other than your test application. This was the technique we used in the ITSO-Raleigh environment to exercise the socket API trace functions. We ran our test server connected to the secondary TCP/IP system, and the client connected to the primary TCP/IP system on the same MVS system. Please refer to MVS TCP/IP V3R1 Implementation Guide, GG24-3687, for instructions on how you can run two TCP/IP stacks on the same MVS system.

You start the socket API trace via an OBEYFILE command with the following content:

```
FILE 'TCPIP.V3R1.RAIANJE.B.TRACE'
TRACE SOCKET
MORETRACE SOCKET
```

The FILE statement instructs the TCP/IP address space to direct trace output to the specified sequential data set. If the data set already exists, it will be overwritten. If you allocate the data set in advance, you must allocate it with RECFM=VB and LRECL=137.

The TRACE statement instructs the TCP/IP address space to start the socket API trace. If you want the trace output to include the data you send and receive, you must also specify the MORETRACE statement.

You then start the application you want to trace. Do not run unnecessary long tests. The amount of trace data can be quite voluminous.

You stop the trace with another OBEYFILE command containing the following:

```
NOTRACE SOCKET
SCREEN
```

The NOTRACE statement stops the socket trace.

The SCREEN statement closes and deallocates the trace data set you specified on the FILE statement when you started your socket trace, and it returns trace output to the default SYSDEBUG DD allocation.

The trace data set contains formatted trace information ready for print or browse.

The samples in Figure 54 to Figure 61 cover the initial socket calls of an iterative server. A few of the lines have been split into two lines in order to make the samples more readable. The extra lines can be identified by the lack of timestamp and message number.

```
17:00:55 EZB7254E IUCV API greeter called for ACB 67531840:
17:00:55 EZB6710I IUCV interrupt -> IUCV-API-greeter (from External interrupt handler)
17:00:55 EZB6696I Interrupt type: Pending connection
17:00:55 EZB6697I Path id: 2
17:00:55 EZB6698I Address space: AsID00FB, User1:
17:00:55 EZB7274I IucvAccMsglim: Path ExtInt 2 (No CCB), msgid '6956577', user1 'TCP/IP ', user2 'TCP/IP ', msglim 2, retcc 0, iprc
17:00:55 EZB7254E IUCV API greeter called for ACB 67531840:
```
Figure 54. Socket API Trace: INITAPI Call

Figure 54 represents one call. The call type can be determined by examining the TrgCls field. See Table 9 for a list of possible values.

The first call is an initapi call, and it results in an initial IUCV message, that includes:

- The maximum number of sockets your program will work with (0002), which is the value passed on the MAXSOC parameter. The default value is 50, but you may specify a maximum value of 2000.
- The APITYPE (0002).
- Your subtask ID (LARGE), which is the value you pass on the SUBTASK parameter.

The reply message includes the maximum socket descriptor that your application can use (X'00000031'), which is equal to 49. So even if you specify a MAXSOC value of 2, your default maximum socket descriptor number is 49 (lowest is 0 and highest is 49).

Figure 55. Socket API Trace: GETCLIENTID Call

The TrgCls field tells us that this is a getclientid call. The returned data for this call is a client ID structure. The first full-word shows the protocol domain (2); the next 8 bytes shows the address space name.
The **TrgCls** field tells us that this is a **socket** call.

The **socket** call is for a socket in the internet domain (domain 2) using a stream socket (socket type 1) and the default protocol for such a socket (protocol 0), which for the internet domain is TCP.

The socket descriptor is returned in the RETCODE field as 0.

---

**Figure 56. Socket API Trace: SOCKET Call**

---

According to the **TrgCls** field, this is a **bind** call on socket descriptor 0.

The socket is bound to an AF-INET address (2); the port number is 9997 (X'270D'), and the IP address is any IP address on this host (0).
This call is a `listen` call on socket descriptor 0.

The backlog queue size is 10, as specified on the BACKLOG parameter on the `listen` call.
This call is an **accept** call. The caller is put into a blocked state until a connection request arrives. The call returns a new socket descriptor in the RETCODE parameter (socket descriptor 1), and the socket address structure of the connecting socket, which is an AF_INET socket (2) with port number 1047 (X'0417') and IP address 9.67.56.18 (X'09433812').

The **TrgCls** field tells us that this is a receive call on socket descriptor 1. The **PrmMsgLo** field indicates that this is a receive call with a flag value of 2, which means it is a MSG_PEEK call.

The peek call returns 16 bytes (X'10'). The socket address structure of the sender is returned and is equal to the socket that connected in the preceding accept call (AF_INET, port 1047 and IP address 9.67.56.18).
A Beginner's Guide to MVS TCP/IP Socket Programming

17:01:15 EZB5062I 055678: 04055558C 00000008 0405553C 00000010 040FC1A9 00000218
17:01:15 EZB7273I Data, length = 8:
17:01:15 EZB5062I 05558C: 000020417 09433812 00000000 00000000
17:01:15 EZB5062I 05555C: 00020417 00000000 00000000 00000000
17:01:15 EZB5062I 055678: 040555558C 00000008 0405553C 00000010 040FC1A9 00000218
17:01:15 EZB5062I 00000218 00000000 00000000 00000000
17:01:15 EZB5062I 00000000 00000000 00000000 00000000

Figure 61. Socket API Trace: RECEIVE Call

The last call we show in this example is a receive call. 536 bytes (X'218') is returned to the application.

If you need to analyze the socket trace in more detail, you can find more information on the IUCV interface in chapter 8 in IBM TCP/IP for MVS: Application Programming Interface Reference, SC31-7187.

<table>
<thead>
<tr>
<th>Call-type</th>
<th>TrgCls</th>
<th>PrmMsg</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITAPI</td>
<td>High Order Bytes</td>
<td>Low Order Bytes</td>
</tr>
<tr>
<td>ACCEPt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLOSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONNECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCNTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GETHOSTID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GETHOSTNAME</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Beginner's Guide to MVS TCP/IP Socket Programming

| GETPEERNAME | 9 | 0009 | ssss | 0 | 0 | Socket address of remote socket |
| GETSOCKNAME | 10 | 000A | ssss | 0 | 0 | Socket address of socket |
| GETSOCKOPT  | 11 | 000B | ssss | level | option name | Value of option |
| IOCTL       | 12 | 000C | ssss | Command and argument |
| SELECT / SELECTX | 13 | 000D | descriptor set size | Select masks |
| READ / READV | 14 | 000E | ssss | 0 | 0 | Received data |
| RECV / RECVFROM / RECVMSG | 16 | 0010 | ssss | 0 | flags | Received data |
| LISTEN      | 19 | 0013 | ssss | 0 | backlog | Data to be sent |
| SEND / SENDMSG | 20 | 0014 | ssss | Data to be sent |
| SENDTO      | 22 | 0016 | ssss | Flags and data to be sent |
| SETSOCKOPT  | 23 | 0017 | ssss | Option name and value |
| SHUTDOWN    | 24 | 0018 | ssss | 0 | direction | Option name and value |
| SOCKET      | 25 | 0019 | 0000 | Domain, type and protocol |
| WRITE / WRITEV | 26 | 001A | ssss | Data to be written |
| GETCLIENTID | 30 | 001E | 0000 | The client ID of this socket |
| GIVESOCKET  | 31 | 001F | ssss | Client ID to give |
| TAKESOCKET  | 32 | 0020 | 0000 | Client ID to take |

Note: ssss denotes a socket descriptor number.

Table 9. Important IUCV Socket Trace Entry Fields

A.0 Appendix A. Sample Datagram Socket Programs

This appendix contains the following two sets of datagram socket programs:

The first sample datagram application is written in COBOL using the Sockets Extended call API. This sample consists of a server ("Datagram Socket COBOL Server Program" in topic A.1) and a client ("Datagram Socket COBOL Client Program" in topic A.2).

The second sample datagram application is written in C. This sample consists of a server ("Datagram Socket C Server Program" in topic A.3) and a client ("Datagram Socket C Client Program" in topic A.4). This C datagram application is written so the source code can be ported between OS/2 and MVS.
A.1 Datagram Socket COBOL Server Program

Identification Division.
*==========================================================================================================*
* Name: TPIDGSRV - MVS iterative echo server using UDP protocols. Client is TPIDGCLN *
* Function: This server works with datagram sockets. It waits for incoming datagrams on port 9999. Received datagrams are echoed back to the client that sent them. If a received datagram is a close-down message, the server terminates itself.
* Interface: TCP/IP address space name via EXEC PARM field.
* Logic: 1. Establish server setup
* 2. Bind datagram socket to local port 9999
* 3. Receive datagram
* 4. Received datagram is echoed back to UDP client that sent it.
* Returncode: - none -
* Written: April 8, 1995 at ITSO Raleigh
*==========================================================================================================*

Program-id. tpidgsrv.

Environment Division.

Data Division.

Working-storage Section.
* Socket interface function codes
*-----------------------------------------------------------------------------------------------------------------
01 soket-functions.
  02 soket-accept pic x(16) value 'ACCEPT '.
  02 soket-bind pic x(16) value 'BIND '.
  02 soket-close pic x(16) value 'CLOSE '.
  02 soket-connect pic x(16) value 'CONNECT '.
  02 soket-fcntl pic x(16) value 'FCNTL '.
  02 soket-getclientid pic x(16) value 'GETCLIENTID '.
  02 soket-gethostbyname pic x(16) value 'GETHOSTBYADDR '.
  02 soket-gethostbyaddr pic x(16) value 'GETHOSTBYNAME '.

A Beginner's Guide to MVS TCP/IP Socket Programming
A Beginner's Guide to MVS TCP/IP Socket Programming

02 soket-gethostid pic x(16) value 'GETHOSTID'.
02 soket-gethostname pic x(16) value 'GETHOSTNAME'.
02 soket-getpeername pic x(16) value 'GETPEERNAME'.
02 soket-getsockname pic x(16) value 'GETSOCKNAME'.
02 soket-getsockopt pic x(16) value 'GETSOCKOPT'.
02 soket-givesocket pic x(16) value 'GIVESOCKET'.
02 soket-initapi pic x(16) value 'INITAPI'.
02 soket-ioct1 pic x(16) value 'IOCTL'.
02 soket-listen pic x(16) value 'LISTEN'.
02 soket-read pic x(16) value 'READ'.
02 soket-recev pic x(16) value 'RECV'.
02 soket-recevfrom pic x(16) value 'RECVFROM'.
02 soket-select pic x(16) value 'SELECT'.
02 soket-send pic x(16) value 'SEND'.
02 soket-sendto pic x(16) value 'SENDTO'.
02 soket-setsockopt pic x(16) value 'SETSOCKOPT'.
02 soket-shutdown pic x(16) value 'SHUTDOWN'.
02 soket-socket pic x(16) value 'SOCKET'.
02 soket-takesocket pic x(16) value 'TAKESOCKET'.
02 soket-termapi pic x(16) value 'TERMAPI'.
02 soket-write pic x(16) value 'WRITE'.

*---------------------------------------------------------------*
* Work variables                                                *
*---------------------------------------------------------------*
01 errno pic 9(8) binary value zero.
01 retcode pic s9(8) binary value zero.
01 client-ipaddr-dotted pic x(15) value space.
88 close-down-message-received value '*CLSDWN*'.
01 saved-message-id-len pic 9(8) Binary value 8.

*---------------------------------------------------------------*
* Variables used for the INITAPI call                           *
*---------------------------------------------------------------*
01 maxsoc pic 9(4) Binary Value 2.
01 initapi-ident.
   05 tcpname pic x(8) Value space.
   05 asname pic x(8) Value space.
01 subtask pic x(8) value space.
01 maxsno pic 9(8) Binary Value 1.

*---------------------------------------------------------------*
* Variables returned by the GETCLIENTID Call                    *
*---------------------------------------------------------------*
01 clientid.
   05 clientid-domain pic 9(8) Binary.
   05 clientid-name pic x(8) value space.
   05 clientid-task pic x(8) value space.
   05 filler pic x(20) value low-value.

*---------------------------------------------------------------*
* Variables used for the SOCKET call                            *
*---------------------------------------------------------------*
01 afinet pic 9(8) Binary Value 2.
01 soctype-datagram pic 9(8) Binary Value 2.
01 proto pic 9(8) Binary Value zero.
01 socket-descriptor pic 9(4) Binary Value zero.

*---------------------------------------------------------------*
* Variables used for the BIND call                              *
*---------------------------------------------------------------*
01 server-socket-address.
   05 server-afinet pic 9(4) Binary Value 2.
   05 server-port pic 9(4) Binary Value 9999.
   05 server-ipaddr pic 9(8) Binary Value zero.
   05 filler pic x(8) value low-value.
Variables used by the RECVFROM Call

01 client-socket-address.
  05 client-afinet       pic 9(4) Binary Value zero.
  05 client-port        pic 9(4) Binary Value zero.
  05 client-ipaddr      pic 9(8) Binary Value zero.
  05 filler             pic x(8) value low-value.

Buffer and length field for recvfrom and sendto operation

01 send-request-len  pic 9(8) Binary Value zero.
01 read-request-len  pic 9(8) Binary Value zero.
01 read-buffer       pic x(8192) value space.
01 filler redefines read-buffer.
  05 message-id        pic x(8).
  05 filler            pic x(8184).

recvfrom and sendto flags

01 sendto-flag       pic 9(8) Binary value zero.
01 recvfrom-flag     pic 9(8) Binary value zero.

Error message for socket interface errors

01 ezaerror-msg.
  05 filler pic x(9) Value 'Function='.
  05 ezaerror-function pic x(16) Value space.
  05 filler pic x value ' '.
  05 ezaerror-retcode  pic ----99.
  05 filler pic x value ' '.
  05 ezaerror-errno    pic zzz99.
  05 filler pic x value ' '.
  05 ezaerror-text     pic x(50) value ' '.

Linkage Section.

EXEC-parameter-field.

01 EXEC-parameter-field.
  05 parm-ll             pic 9(4) Binary.
  05 parm-tcpname        pic x(8).

Procedure Division using EXEC-parameter-field.

* Initialize socket API

If parm-ll < 8 then
  Display 'Invalid or missing TCP address space name'
  Display ' in EXEC PARM field: PARM='xxxxxxxx''
  Go to exit-now
end-if.
  Move soket-initapi to ezaerror-function.
  Move parm-tcpname to tcpname.
  Call 'TPICLNID' using asname subtask.
  Call 'EZASOKET' using soket-initapi
  maxsoc
  initapi-ident
subtask
maxsno
errno
retcode.
If retcode < 0 then
    move 'Initapi failed' to ezaerror-text
    perform write-ezaerror-msg thru write-ezaerror-msg-exit
    go to exit-now.

*---------------------------------------------------------------*
* Let us see the client-id                                      *
*---------------------------------------------------------------*

move soket-getclientid to ezaerror-function.
Call 'EZASOKET' using soket-getclientid
   clientid
erro
retcode.
If retcode < 0 then
    move 'Getclientid failed' to ezaerror-text
    perform write-ezaerror-msg thru write-ezaerror-msg-exit
    go to exit-term-api.
Display 'Client ID = ' clientid-name ' ' clientid-task.

*---------------------------------------------------------------*
* Get us a datagram socket descriptor                          *
*---------------------------------------------------------------*

move soket-socket to ezaerror-function.
Call 'EZASOKET' using soket-socket
   afinet
   soctype-datagram
   proto
erro
retcode.
If retcode < 0 then
    move 'Socket call failed' to ezaerror-text
    perform write-ezaerror-msg thru write-ezaerror-msg-exit
    go to exit-term-api.
Move retcode to socket-descriptor.

*---------------------------------------------------------------*
* Bind socket to our server port number                        *
*---------------------------------------------------------------*

Move soket-bind to ezaerror-function.
Call 'EZASOKET' using soket-bind
   socket-descriptor
   server-socket-address
erro
retcode.
If retcode < 0 then
    move 'Bind call failed' to ezaerror-text
    perform write-ezaerror-msg thru write-ezaerror-msg-exit
    go to exit-close-socket.

*---------------------------------------------------------------*
* Loop reading and sending client datagrams until               *
* server receives a datagram that starts with the               *
* text *CLSDWN* - then we shut down.                            *
*---------------------------------------------------------------*
Perform until close-down-message-received
Display 'Entering new blocking recvfrom call'
move soket-recvfrom to ezaerror-function
move 8192 to read-request-len
Call 'EZASOCKET' using soket-recvfrom
  socket-descriptor
  recvfrom-flag
  read-request-len
  read-buffer
  client-socket-address
  errno
  retcode
If retcode < 0 then
  move 'Recv-from call failed' to ezaerror-text
  perform write-ezaerror-msg thru
  write-ezaerror-msg-exit
  go to exit-close-socket
end-if
Call 'TPIINTOA' using client-ipaddr
  client-ipaddr-dotted
Display 'Data from ip address ' client-ipaddr-dotted
Display ' and port number ' client-port
Move message-id to saved-message-id
if not close-down-message-received then
  Call 'EZACIC05' using saved-message-id
  saved-message-id-len
end-if
If close-down-message-received then
  Display 'We received a shut-down message'
else
  move soket-sendto to ezaerror-function
  move 8192 to send-request-len
  Call 'EZASOCKET' using soket-sendto
    socket-descriptor
    sendto-flag
    send-request-len
    read-buffer
    client-socket-address
    errno
    retcode
If retcode < 0 then
  move 'Sendto call failed' to ezaerror-text
  perform write-ezaerror-msg thru
  write-ezaerror-msg-exit
  go to exit-close-socket
end-if
end-if
end-perform.

*---------------------------------------------------------------*
* Close socket                                                 *
*---------------------------------------------------------------*
exit-close-socket.
move soket-close to ezaerror-function
Call 'EZASOCKET' using soket-close
  socket-descriptor
  errno
  retcode.
If retcode < 0 then
move 'Close call failed' to ezaerror-text
perform write-ezaerror-msg thru write-ezaerror-msg-exit.

*---------------------------------------------------------------*
* Terminate socket API                                        *
*---------------------------------------------------------------*
exit-term-api.
   Call 'EZASOKET' using soket-termapi.

*---------------------------------------------------------------*
* Terminate program                                            *
*---------------------------------------------------------------*
exit-now.
   move zero to return-code.
   Goback.

*---------------------------------------------------------------*
* Subroutine                                                   *
*---------------------------------------------------------------*
* *                                                            *
* Write out an error message                                   *
*---------------------------------------------------------------*
write-ezaerror-msg.
   move errno to ezaerror-errno.
   move retcode to ezaerror-retcode.
   display ezaerror-msg.
write-ezaerror-msg-exit.
exit.

A.2 Datagram Socket COBOL Client Program

Identification Division.
*==================================*
*==================================*
* Name: TPIDGCLN - Client to test MVS Datagram server TPIDGSRV (UDP protocols). *
* Function: Sends 8K message to server and receives reply. *
* If client start option specifies CLOSE, the *
* client sends a shutdown datagram to the server. *
* This program uses non-blocking recvfrom calls *
* in order to implement its own timeout logic *
* in case the server does not respond to its *
* request. *
* Interface: CLOSE option in EXEC PARM field *
* Logic: 1. Sends datagram to server *
*        2. Reads echoed datagram from server *
* Returncode: - none - *
* Written: April 8, 1995 at ITSO Raleigh *
*---------------------------------------------------------------*
Program-id. tpidgcln.
Environment Division.

Data Division.

Working-storage Section.

Socket interface function codes

```
01 soket-functions.
  02 soket-accept pic x(16) value 'ACCEPT'.
  02 soket-bind pic x(16) value 'BIND'.
  02 soket-close pic x(16) value 'CLOSE'.
  02 soket-connect pic x(16) value 'CONNECT'.
  02 soket-fcntl pic x(16) value 'FCNTL'.
  02 soket-getclientid pic x(16) value 'GETCLIENTID'.
  02 soket-gethostbyaddr pic x(16) value 'GETHOSTBYADDR'.
  02 soket-gethostbyname pic x(16) value 'GETHOSTBYNAME'.
  02 soket-gethostid pic x(16) value 'GETHOSTID'.
  02 soket-gethostname pic x(16) value 'GETHOSTNAME'.
  02 soket-getpeername pic x(16) value 'GETPEERNAME'.
  02 soket-getsockname pic x(16) value 'GETSOCKNAME'.
  02 soket-getsockopt pic x(16) value 'GETSOCKOPT'.
  02 soket-givesocket pic x(16) value 'GIVESOCKET'.
  02 soket-initapi pic x(16) value 'INITAPI'.
  02 soket-ioctl pic x(16) value 'IOCTL'.
  02 soket-listen pic x(16) value 'LISTEN'.
  02 soket-read pic x(16) value 'READ'.
  02 soket-recv pic x(16) value 'RECV'.
  02 soket-recvfrom pic x(16) value 'RECVFROM'.
  02 soket-select pic x(16) value 'SELECT'.
  02 soket-send pic x(16) value 'SEND'.
  02 soket-sendto pic x(16) value 'SEN DTO'.
  02 soket-shutdown pic x(16) value 'SHUTDOWN'.
  02 soket-socket pic x(16) value 'SOCKET'.
  02 soket-takesocket pic x(16) value 'TAKESOCKET'.
  02 soket-termapi pic x(16) value 'TERMAPI'.
  02 soket-write pic x(16) value 'WRITE'.
```

Work variables

```
01 errno pic 9(8) binary value zero.
01 retcode pic s9(8) binary value zero.
01 index-counter pic 9(8) binary value zero.
01 buffer-element.
  05 buffer-element-nbr pic 9(5).
  05 filler pic x(3) value space.
01 server-ipaddr-dotted pic x(15) value space.
01 close-server pic 9(8) binary value zero.
  88 close-server-down value 1.
01 timer-accum pic 9(8) binary value zero.
01 timer-interval pic 9(8) binary value 1000.
```

Variables used for the INITAPI call

```
01 maxsoc pic 9(4) binary Value 1.
01 initapi-ident.
```
* Variables returned by the GETCLIENTID Call
  *---------------------------------------------------------------*
  01 clientid.
  05 clientid-domain pic 9(8) Binary.
  05 clientid-name pic x(8) value space.
  05 clientid-task pic x(8) value space.
  05 filler pic x(20) value low-value.

* Variables used for the SOCKET call
  *---------------------------------------------------------------*
  01 afinet pic 9(8) Binary Value 2.
  01 soctype-datagram pic 9(8) Binary Value 2.
  01 proto pic 9(8) Binary Value zero.
  01 socket-descriptor pic 9(4) Binary Value zero.

* Variables used for the GETHOSTBYNAME Call
  *---------------------------------------------------------------*
  01 host-namelen pic 9(8) Binary Value 5.
  01 host-name pic x(5) Value 'mvs18'.
  01 host-entry-addr pic 9(8) Binary Value zero.

* Variables used for the call to EZACIC08
  *---------------------------------------------------------------*
  01 host-alias-seq pic 9(4) Binary Value zero.
  01 host-addr-seq pic 9(4) Binary Value zero.
  01 host-name-length pic 9(4) Binary Value zero.
  01 host-name-value pic x(255) Value space.
  01 host-alias-count pic 9(4) Binary Value zero.
  01 host-alias-length pic 9(4) Binary Value zero.
  01 host-alias-value pic x(255) Value space.
  01 host-addr-type pic 9(4) Binary Value zero.
  01 host-addr-length pic 9(4) Binary Value zero.
  01 host-addr-count pic 9(4) Binary Value zero.
  01 host-addr-value pic 9(8) Binary Value zero.
  01 host-return-code pic s9(8) Binary Value zero.

* Server socket address structure
  *---------------------------------------------------------------*
  01 server-socket-address.
  05 server-afinet pic 9(4) Binary Value 2.
  05 server-port pic 9(4) Binary Value 9999.
  05 server-ipaddr pic 9(8) Binary Value zero.
  05 filler pic x(8) value low-value.

* Variables used for the IOCTL call
  *---------------------------------------------------------------*
  01 ioctl-command-fionbio pic x(4).
  01 ioctl-command-string pic x(16) value 'FIONBIO'.
  01 ioctl-reqarg-non-blocking pic 9(8) Binary value 1.
  01 ioctl-retarg pic 9(8) binary value zero.

* Buffer and length fields for sendto operation
  *---------------------------------------------------------------*
  01 send-request-length pic 9(8) Binary value zero.
  01 send-buffer.
  05 send-buffer-total pic x(8192) value space.
  05 closedown-message redefines send-buffer-total.
A Beginner's Guide to MVS TCP/IP Socket Programming

10 closedown-id pic x(8).
10 filler pic x(8184).
05 send-buffer-seq redefines send-buffer-total pic x(8) occurs 1024 times.

*---------------------------------------------------------------*
* Buffer and length fields for recvfrom operation *
*---------------------------------------------------------------*
01 read-request-length pic 9(8) Binary value zero.
01 read-buffer pic x(8192) value space.

*---------------------------------------------------------------*
* Other fields for sendto and recvfrom operation *
*---------------------------------------------------------------*
01 sendto-flag pic 9(8) Binary value zero.
01 recvfrom-flag pic 9(8) Binary value zero.

*---------------------------------------------------------------*
* Error message for socket interface errors *
*---------------------------------------------------------------*
01 ezaerror-msg.
  05 filler pic x(9) Value 'Function='.
  05 ezaerror-function pic x(16) Value space.
  05 filler pic x value ' '.
  05 filler pic x(8) Value 'Retcode='.
  05 ezaerror-retcode pic ---99.
  05 filler pic x value ' '.
  05 filler pic x(9) Value 'Errortno='.
  05 ezaerror_errno pic zzz99.
  05 filler pic x value ' '.
  05 ezaerror-text pic x(50) value ' '.

Linkage Section.

*--------------
01 EXEC-parameter-field.
  05 parm-ll pic 9(4) Binary.
  05 parm-close-option pic x(5).

*---------------------------------------------------------------*
* Initialize send buffer *
*---------------------------------------------------------------*
perform varying index-counter from 0 by 1 until index-counter > 1023
  move index-counter to buffer-element-nbr
  move buffer-element to send-buffer-seq(index-counter)
end-perform.

*---------------------------------------------------------------*
* Initialize socket API *
*---------------------------------------------------------------*
Move soket-initapi to ezaerror-function.
Call 'TPICLNID' using asname subtask.
Call 'EZASOKET' using soket-initapi
maxsoc
initapi-ident
subtask
maxsno
errno
retcode.
If retcode < 0 then
  move 'Initapi failed' to ezaerror-text
  perform write-ezaerror-msg thru write-ezaerror-msg-exit
  go to exit-now.

*---------------------------------------------------------------*
* Let us see the client-id                                      *
*---------------------------------------------------------------*
move soket-getclientid to ezaerror-function.
Call 'EZASOKET' using soket-getclientid
  clientid
erno
  retcode.
If retcode < 0 then
  move 'Getclientid failed' to ezaerror-text
  perform write-ezaerror-msg thru write-ezaerror-msg-exit
  go to exit-term-api.
Display 'Our client ID = ' clientid-name ' ' clientid-task.

*---------------------------------------------------------------*
* Get us a datagram socket descriptor                          *
*---------------------------------------------------------------*
move soket-socket to ezaerror-function.
Call 'EZASOKET' using soket-socket
  afinet
  soctype-datagram
  proto
  errno
  retcode.
If retcode < 0 then
  move 'Socket call failed' to ezaerror-text
  perform write-ezaerror-msg thru write-ezaerror-msg-exit
  go to exit-term-api.
Move retcode to socket-descriptor.

*---------------------------------------------------------------*
* Get host entry structure pointer based on host name          *
*---------------------------------------------------------------*
move soket-gethostbyname to ezaerror-function.
Call 'EZASOKET' using soket-gethostbyname
  host-namelen
  host-name
  host-entry-addr
  retcode.
If retcode < 0 then
  move 'Gethostbyname failed' to ezaerror-text
  perform write-ezaerror-msg thru write-ezaerror-msg-exit
  go to exit-close-socket.

*---------------------------------------------------------------*
* Get info out of the HOSTENT structure                        *
* As we do not know if server IP address is there, we can       *
* only use the first returned address for our datagram.        *
*---------------------------------------------------------------*
move 'EZACIC08' to ezaerror-function.
Call 'EZACIC08' using host-entry-addr
    host-name-length
    host-name-value
    host-alias-count
    host-alias-seq
    host-alias-length
    host-alias-value
    host-addr-type
    host-addr-length
    host-addr-count
    host-addr-seq
    host-addr-value
    host-return-code.
If host-return-code = -1 then
    move host-return-code to retcode
    move 'Host translation failed' to ezaerror-text
    perform write-ezaerror-msg thru
    write-ezaerror-msg-exit
    go to exit-close-socket
end-if.

Move host-addr-value to server-ipaddr.
Call 'TPIINTOA' using server-ipaddr server-ipaddr-dotted.
Display 'Sending datagram to ' server-ipaddr-dotted.

*---------------------------------------------------------------*
* Send datagram to server                                       *
*---------------------------------------------------------------*

move soket-sendto to ezaerror-function.
move 8192 to send-request-length.
If close-server-down then
    Display 'Sending server shutdown message'
    move '*CLSDWN*' to closedown-id.
Call 'EZASOKET' using soket-sendto
    socket-descriptor
    sendto-flag
    send-request-length
    send-buffer-total
    server-socket-address
    errno
    retcode.
If retcode < 0 then
    move 'Write call failed' to ezaerror-text
    perform write-ezaerror-msg thru
    write-ezaerror-msg-exit
    go to exit-close-socket
end-if.
If close-server-down then
    go to exit-close-socket.

* We do not know, if the server is there, so we will not enter *
* a blocking receive for the echoed datagram. Instead we turn *
* the socket into non-blocking mode, and enters a loop where we *
* issue a non-blocking recvfrom call. If no data, we go into *
* a one second wait and then reissue the recvfrom call. If we *
* have not received a reply within 30 seconds, we timeout and *
* terminate the client. *
*------------------------------------------------------------------*

A Beginner's Guide to MVS TCP/IP Socket Programming 161
Move soket-ioctl to ezaerror-function.
Call 'TPI IOCTL' using ioctl-command-string
ioctl-command-fionbio.
If return-code > zero then
  move 'Call to TPI IOCTL failed' to ezaerror-text
  perform write-ezaerror-msg thru
  write-ezaerror-msg-exit
  go to exit-close-socket
end-if.
Call 'EZASOKET' using soket-ioctl
socket-descriptor
ioctl-command-fionbio
ioctl-reqarg-non-blocking
ioctl-retarg
errno
retcode.
If retcode < 0 then
  move 'IOCTL call failed' to ezaerror-text
  perform write-ezaerror-msg thru
  write-ezaerror-msg-exit
  go to exit-close-socket
end-if.
move 0 to timer-accum.
perform until timer-accum >= 30000
  move soket-recvfrom to ezaerror-function
  move 8192 to read-request-length
  Call 'EZASOKET' using soket-recvfrom
  socket-descriptor
  recvfrom-flag
  read-request-length
  read-buffer
  server-socket-address
  errno
  retcode
  If retcode < 0 and errno not = 35 then
    move 'Recv-from call failed' to ezaerror-text
    perform write-ezaerror-msg thru
    write-ezaerror-msg-exit
    go to exit-close-socket
  end-if
  If errno = 35 then
    If timer-accum < 30000 then
      Display 'Waiting one second'
      add timer-interval to timer-accum
      Call 'TPI WAIT' using timer-interval
    else
      Display 'Timed out before server returned datagram'
    end-if
  else
    Display 'We have recieved our echoed datagram'
    Move 31000 to timer-accum
  end-if
end-perform.
*---------------------------------------------------------------*
* Close socket *
*---------------------------------------------------------------*
exit-close-socket.
move soket-close to ezaerror-function
Call 'EZASOKET' using soket-close
   socket-descriptor
   errno
   retcode.
If retcode < 0 then
   move 'Close call failed' to ezaerror-text
   perform write-ezaerror-msg thru write-ezaerror-msg-exit.

*---------------------------------------------------------------*
* Terminate socket API*
*---------------------------------------------------------------*
exit-term-api.
   Call 'EZASOKET' using soket-termapi.

*---------------------------------------------------------------*
* Terminate program *
*---------------------------------------------------------------*
exit-now.
   move zero to return-code.
   Goback.

*---------------------------------------------------------------*
* Subroutine. *
* *
* *
* *
* Write out an error message *
*---------------------------------------------------------------*
write-ezaerror-msg.
   move errno to ezaerror-errno.
   move retcode to ezaerror-retcode.
   display ezaerror-msg.
write-ezaerror-msg-exit.
exit.

A.3 Datagram Socket C Server Program

/* Portable UDP socket server - February 1995 */
define WAIT
#include <stdlib.h>
#include <time.h> /* time stamp */

#ifdef MVS
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>
#include <inet.h>
#include <socket.h>
#include <errno.h>
#include <tcperrno.h>
#include <dis.h>
#else
/* On OS/2, use SO32DLL.LIB TCP32DLL.LIB */
#include <types.h>
#include <sys\socket.h>
#include <inet\in.h>
#include <errno.h> /* sock_errno() */
define close soclose
#endif

#define tcperror psock_errno
#include "d:\rbb\dis.h"
#endif
#include <netdb.h> /* should not precede #include <manifest.h> on MVS */
#endif
#endif
#endif
#define wait(text) printf("Hit <enter> key to continue with " #text ".\n") ; getchar();
int yesno(char * what)
{
    char answ ;
    for (; ; ) {
        printf("Enter \'Y\' to continue, \'N\' to stop with %s: ",what);
        answ = ' '|getchar(); /* EBCDIC uppercase translation */
        getchar(); /* absorb enter key */
        switch (answ) {
            case 'Y': return 1; break;
            case 'N': return 0; break;
            default: ;
        } /* endswitch */
    } /* endfor */
}
#define wait(text) printf("Hit <any> key to continue with " #text ".\n") ; getch();
int yesno(char * what)
{
    for (; ; ) {
        printf("Enter \'Y\' to continue, \'N\' to stop with %s\n",what);
        switch (' '|getch()) { /* ASCII lowercase translation */
            case 'y': return 1; break;
            case 'n': return 0; break;
            default: ;
        } /* endswitch */
    } /* endfor */
}
#endif
#define wait(text)
define yesno(text) once--
int once = 1; /* this only works as there is just one yesno call in the program */
#endif
/* #define check(x,y) if (y) { psock_errno(x); exit(1); } else printf(x " OK\n"); */
#define CHECK(x,y) time(&t1) , check(x,y)
time_t t1, t2 ;
int check(char *text, int condition) /* if TRUE, error */
{
    printf("%s",text);
    if (condition) {
        tcperror("error");
        #ifdef MVS
        return errno ;
        #else
        return sock_errno() ;
        #endif
    } else {
        time(&t2) ; /* get timestamp in seconds */
        printf("completed in %i seconds\n",t2-t1);
    }
return 0 ;
} /* endif */

int main(int argc,char**argv)
{
    int socketNumber ;
    int bytesReceived ;
    struct sockaddr_in clientAddr ;
    struct sockaddr_in localAddr ;
    unsigned long bufferSize = 4 ;
    char * buffer ;
    char * bufferChar ;
    struct hostent * hostEnt ;
    unsigned short port = 9999 ;
    time_t ltime ;
    int nameLen = sizeof(struct sockaddr_in);

    setbuf(stdout,NULL); /* don't buffer: don't loose output in case of errors */
    setbuf(stderr,NULL); /* should not be necessary ... */

    time(&ltime); /* Get timestamp in seconds */

    if (argc>1) if (*argv[1]=='?') {
        say(Parameters:
1. port
2. receive buffer size);
        return 0;
    } /* endif */

    if (argc>1) if (*argv[1]!='*') port = atoi(argv[1]); disint(port);
    if (argc>2) if (*argv[2]!='*') bufferSize = atoi(argv[2]); disint(bufferSize);

    if (!(( buffer = (char*)malloc(bufferSize+1) ))) { say(Insufficient storage to allocate receive buffer); return 1; }

#ifndef MVS
    if (CHECK("sock_init",sock_init())) return -1;
#endif

    if (CHECK("socket",(socketNumber=socket(AF_INET,SOCK_DGRAM,0))<0)) return -1; /* create datagram socket */

    localAddr.sin_family = AF_INET ;
    localAddr.sin_addr.s_addr = INADDR_ANY ;
    localAddr.sin_port = htons(port) ;
    if (CHECK("bind",bind(socketNumber,(struct sockaddr*)&localAddr,nameLen)<0)) return -1;

    /* receive data from client */
    while (yesno("RECV")) {
        say(Waiting to receive data);
        if (CHECK("recvfrom",(bytesReceived=recvfrom(socketNumber,buffer,bufferSize,0,(struct sockaddr*)&clientAddr,&nameLen))<0)) return -1;
        printf("Received from %s port %i (%sAF_INET family).
", (hostEnt=gethostbyaddr((char*)&clientAddr.sin_addr,sizeof(clientAddr.sin_addr),AF_INET))?hostEnt->h_name:inet_ntoa(clientAddr.sin_addr), clientAddr.sin_port, clientAddr.sin_family==AF_INET?"":"NOT ");
        dislong(bytesReceived);
        *(buffer+bytesReceived) = 0 ; /* for disstr */
        bufferChar = buffer ;
        while (++bufferChar=*buffer); /* investigate whether all the same character */
        if (bufferChar-buffer==bytesReceived) {
            printf("All characters '%c' (X'%2.2X') received.\n",*buffer,*buffer);
        } else {
            disstr(buffer);
        } /* endif */
    } /* endwhile */
} /* return */
A.4 Datagram Socket C Client Program

#include <stdio.h>
#include <string.h>
#include <time.h> /* time stamp */

#else
/* On OS/2, use S032DLL.LIB TCP32DLL.LIB */
#include <types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <nerrno.h> /* sock_errno() */
#define close soclose
#define tcperror psock_errno
#include "d:\rbb\dis.h"
#endif

#include <netdb.h> /* should not precede #include <manifest.h> on MVS */

#endif WAIT
#endif MVS
#define wait(text) printf("Hit <enter> key to continue with " #text ".\n"); getchar();
int yesno(char * what)
{
    char answ ;
    for (; ; ) {
        printf("Enter \"Y\" to continue, \"N\" to stop with %s:\",what);
        answ = '|'getchar(); /* EBCDIC uppercase translation */
        getchar(); /* absorp enter key */
        switch (answ) {
            case 'Y': return 1; break;
            case 'N': return 0; break;
            default: ;
        } /* endswitch */
    } /* endfor */
}
#endif MVS
#define wait(text) printf("Hit <any> key to continue with " #text ".\n"); getch();
int yesno(char * what)
{
    for (; ; ) {
        printf("Enter \"Y\" to continue, \"N\" to stop with %s\n",what);
        switch (' '|getch()) { /* ASCII lowercase translation */
            case 'Y': return 1; break;
            case 'N': return 0; break;
            default: ;
        } /* endswitch */
    } /* endfor */
}
case 'y': return 1; break;
case 'n': return 0; break;
default: ;
} /* endforeach */
} /* endfor */
#endif
#else
#define wait(text)
define yesno(text) once--
int once = 1; /* this only works as there is just one yesno call in the program */
#endif
/* #define check(x,y) if (y) { psock_errno(x); exit(1); } else printf(x " OK
"); */
#define CHECK(x,y) time(&t1) , check(x,y)

int main(int argc, char**argv )
{
  int socketNumber ;
  int bytesSent = 0 ;
  int bytesToBeSent = 12 ;
  char * hostName ;
  unsigned long binaryAddress ;
  unsigned short serverPort = 9999 ;
  unsigned short clientPort = 0 ;
  struct sockaddr_in serverAddr ;
  struct sockaddr_in fromAddr ;
  char * reason ;
  struct hostent * hostEnt ;
  char * sendBuffer ;
  char * receiveBuffer ;
  int nameLen = sizeof(struct sockaddr_in);
  time_t ltime ;

  setbuf(stdout,NULL); /* don't buffer: don't loose output in case of errors */
  setbuf(stderr,NULL); /* should not be necessary ... */
  time(&ltime); /* Get timestamp in seconds */

  if (argc>1) if (*argv[1]=='?') {
    say(Parameters:
1. address server (dotted or symbolic)
2. serverPort
3. bytes to be sent
4. local port to be used);
return 0;
} /* endif */
A Beginner's Guide to MVS TCP/IP Socket Programming

if (argc>1) if (*argv[1]!='*') hostName = argv[1] ; disstr(hostName );
if (argc>2) if (*argv[2]!='*') serverPort = atoi(argv[2]); disint(serverPort );
if (argc>3) if (*argv[3]!='*') bytesToBeSent = atoi(argv[3]); disint(bytesToBeSent );
if (argc>4) if (*argv[4]!='*') clientPort = atoi(argv[4]); disint(clientPort );

#ifndef MVS
if (CHECK("sock_init",sock_init())) return -1;
#endif

if (( binaryAddress = inet_addr(hostName) )==-1) {
  if (!(hostEnt=gethostbyname(hostName))) {
    switch (h_errno) {
    case HOST_NOT_FOUND : reason = "host not found" ; break;
    case TRY_AGAIN : reason = "try again" ; break;
    case NO_RECOVERY : reason = "no recovery" ; break;
    case NO_DATA : reason = "no data/address" ; break;
    /* case NO_ADDRESS : reason = "no address" ; break; */
    default: disint(h_errno);reason = "?" ;
    } /* endswitch */
    printf("Gethostbyname for host \"%s\" failed, reason: %s.\n",hostName,reason);
    return 0 ;
  } /* endif */
  binaryAddress = *(unsigned long*) *hostEnt->h_addr_list ;
  printf("Host \"%s\" has address %s\n",hostName ,inet_ntoa(*(struct in_addr*)&binaryAddress));
} /* endif */

if (CHECK("socket",(socketNumber=socket(AF_INET,SOCK_DGRAM,0))<0)) return -1; /* create datagram socket */

serverAddr.sin_family = AF_INET ;
serverAddr.sin_addr.s_addr = binaryAddress ;
serverAddr.sin_port = htons(serverPort) ; /* disint(htons(serverAddr.sin_port)); */

/* send message(s) */
while (yesno("SEND")) {
  if (!(( sendBuffer = (char*)malloc(bytesToBeSent) ) ) { say(Insufficient storage to allocate send buffer); return 0 ;
    memset(sendBuffer,'A',bytesToBeSent);  
    if (CHECK("sendto",(bytesSent=sendto(socketNumber,sendBuffer,bytesToBeSent,0,(struct sockaddr*)&serverAddr,sizeof(serverAddr)))<0)) return -1;
    printf("%li bytes have been sent.\n",bytesSent);  
  } /* endwhile */

  wait(CLOSE);
  if (CHECK("close",close(socketNumber))) return -1; /* close socket */
} return 0 ;

B.0 Appendix B. Sample Stream Socket Programs

This appendix contains the following two sets of stream socket programs:

One set is written in COBOL using the Sockets Extended call API. This application consists of a server ("Sample Stream Socket COBOL Server" in topic B.1) and a client ("Sample Stream Socket COBOL Client" in topic B.2). The server is implemented as an iterative server running in a normal MVS address space. This server is referred to from Chapter 5, "Your First Socket Program" in topic 5.0.

Another set is written in C. This application consists of a server ("Sample Stream Socket C Server" in topic B.3) and a client ("Sample Stream Socket C Client" in topic B.4). The C source code is written so the source code can be ported between MVS and OS/2.
B.1 Sample Stream Socket COBOL Server

Identification Division.
*===============================================*
* Name: TPIIESRV - MVS iterative echo server using *
* TCP protocols. Client is TPIIECLN. *
* *
* Function: Each client is required to start the dialog by *
* sending a sign-on message. Information in the *
* sign-on message is used to verify the user and *
* to establish a user security environment via a *
* call to utility routine: TPIRACF. *
* The result of sign-on is returned to the client *
* in a sign-on reply. *
* The client then sends one or more 8K messages *
* to the server that are echoed back to the client.*
* When the client closes the connection, the user *
* security environment is reset, and the server *
* enters a new blocking accept call waiting for *
* the next client. *
* A client may send a Close-down message. If it *
* does, the server asks RACF if the user has *
* authority to do so via READ access to the *
* RACF resource class FACILITY resource name *
* TPIIESRV.CLSDOWN; if OK, the server terminates. *
* *
* Interface: TCP address space name via EXEC PARM field *
* *
* Logic: 1. Establish server setup and listen on *
* port 9997 *
* 2. Accept a connection *
* 3. Receive sign-on message from client *
* 4. Verify user and create task level security *
* environment *
* 5. Receive data message from client *
*   If message is close-down message and client *
*   user is authorized to close down the server, *
*   server terminates itself. *
* 6. Echo data message back to client *
* 7. Wait for new connect *
* *
* Returncode: - none - *
* *
* Written: March 8, 1995 at ITSO Raleigh *
* *
* Modified: *
* *
*---------------------------------------------------------------*

Program-id. tpiiesrv.

*===============================================*

Environment Division.
Data Division.

Working-storage Section.

Socket interface function codes

01 soket-functions.
  02 soket-accept pic x(16) value 'ACCEPT '.
  02 soket-bind pic x(16) value 'BIND '.
  02 soket-close pic x(16) value 'CLOSE '.
  02 soket-connect pic x(16) value 'CONNECT '.
  02 soket-fcntl pic x(16) value 'FCNTL '.
  02 soket-getclientid pic x(16) value 'GETCLIENTID '.
  02 soket-gethostbyaddr pic x(16) value 'GETHOSTBYADDR '.
  02 soket-gethostbyname pic x(16) value 'GETHOSTBYNAME '.
  02 soket-gethostid pic x(16) value 'GETHOSTID '.
  02 soket-gethostname pic x(16) value 'GETHOSTNAME '.
  02 soket-getpeername pic x(16) value 'GETPEERNAME '.
  02 soket-getsockname pic x(16) value 'GETSOCKNAME '.
  02 soket-getsockopt pic x(16) value 'GETSOCKOPT '.
  02 soket-givesocket pic x(16) value 'GIVESOCKET '.
  02 soket-initapi pic x(16) value 'INITAPI '.
  02 soket-ioctl pic x(16) value 'IOCTL '.
  02 soket-listen pic x(16) value 'LISTEN '.
  02 soket-read pic x(16) value 'READ '.
  02 soket-recv pic x(16) value 'RECV '.
  02 soket-recvfrom pic x(16) value 'RECVFROM '.
  02 soket-select pic x(16) value 'SELECT '.
  02 soket-send pic x(16) value 'SEND '.
  02 soket-sendto pic x(16) value 'SENDTO '.
  02 soket-setssockopt pic x(16) value 'SETSOCKOPT '.
  02 soket-shutdown pic x(16) value 'SHUTDOWN '.
  02 soket-socket pic x(16) value 'SOCKET '.
  02 soket-takesocket pic x(16) value 'TAKE_SOCKET '.
  02 soket-termapi pic x(16) value 'TERMAPI '.
  02 soket-write pic x(16) value 'WRITE '.

Work variables

01 errno pic 9(8) binary value zero.
01 retcode pic s9(8) binary value zero.
01 client-ipaddr-dotted pic x(15) value space.
01 client-type pic x value space.
  88 client-is-ascii value 'A'.
  88 client-is-ebcdic value 'E'.
01 client-status pic 9(8) binary value zero.
  88 client-has-closed value 1.

Variables used for the INITAPI call

01 maxsoc pic 9(4) binary value 2.
01 initapi-ident.
  05 tcpname pic x(8) value ' '.
  05 asname pic x(8) value space.
01 subtask pic x(8) value space.
01 maxsno pic 9(8) binary value zero.

Variables returned by the GETCLIENTID Call

A Beginner's Guide to MVS TCP/IP Socket Programming 170
01 clientid.
  05 clientid-domain pic 9(8) Binary.
  05 clientid-name pic x(8) value space.
  05 clientid-task pic x(8) value space.
  05 filler pic x(20) value low-value.

* Variables used for the SOCKET call *

01 afinet pic 9(8) Binary Value 2.
01 soctype-stream pic 9(8) Binary Value 1.
01 proto pic 9(8) Binary Value zero.
01 socket-descriptor pic 9(4) Binary Value zero.

* Variables used for the BIND Call *

01 server-socket-address.
  05 server-afinet pic 9(4) Binary Value 2.
  05 server-port pic 9(4) Binary Value 9997.
  05 server-ipaddr pic 9(8) Binary Value zero.
  05 filler pic x(8) value low-value.

* Variables used by the LISTEN Call *

01 backlog-queue pic 9(8) Binary Value 10.

* Variables used by the ACCEPT Call *

01 client-socket-address.
  05 client-afinet pic 9(4) Binary Value zero.
  05 client-port pic 9(4) Binary Value zero.
  05 client-ipaddr pic 9(8) Binary Value zero.
  05 filler pic x(8) value low-value.
  01 accepted-socket-descriptor pic 9(4) Binary Value zero.

* Variables used by the SETSOCKOPT Linger call *

01 setsockopt-linger pic 9(8) Binary Value 128.
01 setsockopt-value.
  05 linger-on pic 9(8) Binary Value zero.
  05 linger-time pic 9(8) Binary Value 5.
  01 setsockopt-len pic 9(8) Binary Value 8.

* Peek control fields for a peeking RECV call *

01 recv-flag pic 9(8) Binary value zero.
01 recv-flag-read pic 9(8) Binary value zero.
01 recv-flag-peek pic 9(8) Binary value 2.

* Buffer and length field for read operation *

01 read-request-len pic 9(8) Binary Value zero.
01 read-request-read pic 9(8) Binary Value zero.
01 read-request-remaining pic 9(8) Binary Value zero.
01 read-buffer.
  05 read-buffer-total pic x(8192) Value space.
  05 sign-on-message redefines read-buffer-total.
    10 sign-on-id pic x(8).
    88 message-is-signon value '*SIGNON*'.
    10 sign-on-userid pic x(8).
    10 sign-on-pwd pic x(8).
    10 sign-on-new-pwd pic x(8).
A Beginner's Guide to MVS TCP/IP Socket Programming

10 sign-on-group pic x(8).
10 filler pic x(8152).
05 close-down-message redefines read-buffer-total.
10 close-down-id pic x(8).
88 message-is-closedown value '*CLSDWN*'.
10 filler pic x(8184).
05 read-buffer-byte redefines read-buffer-total pic x occurs 8192 times.

* Buffer and length fields for write operation *
*----------------------------------------------------------------------*
01 send-request-len pic 9(8) Binary value zero.
01 send-request-sent pic 9(8) Binary value zero.
01 send-request-remaining pic 9(8) Binary value zero.
01 send-buffer.
05 send-buffer-total pic x(8192) value space.
05 sign-on-reply redefines send-buffer-total.
10 sign-on-reply-id pic x(8).
10 sign-on-rc pic 9(4).
10 filler pic x(8180).
05 send-buffer-byte redefines send-buffer-total pic x occurs 8192 times.

* Fields used for calls to TPIAUTH *
*----------------------------------------------------------------------*
01 tpiiesrv-cls-resource pic x(80) value 'TPIIESRV.CLSDOWN'.
01 tpiauth-read pic x(8) value 'READ'.

* Fields used for calls to TPIRACF *
*----------------------------------------------------------------------*
01 tpiracf-request pic 9(8) Binary value zero.
01 tpiracf-application pic x(8) value 'TPIIESRV'.
01 tpiracf-rc pic 9(8) Binary value zero.

* Error message for socket interface errors *
*----------------------------------------------------------------------*
01 ezaerror-msg.
05 filler pic x(9) Value 'Function='. Value space.
05 ezaerror-function pic x(16) Value space.
05 filler pic x value ' '. Value space.
05 ezaerror-retcode pic ----99. Value space.
05 filler pic x value ' '. Value space.
05 ezaerror-errno pic zzz99. Value space.
05 filler pic x value ' '. Value space.
05 ezaerror-text pic x(50) value ' '. Value space.

Linkage Section.
*--------------------*
01 EXEC-parameter-field.
05 parm-ll pic 9(4) Binary.
05 parm-tcpname pic x(8).

*----------------------------------------------------------------------*
Procedure Division using EXEC-parameter-field.
*----------------------------------------------------------------------*

* Initialize socket API *
*----------------------------------------------------------------------*
If parm-ll < 8 then
    Display 'Invalid or missing TCP address space name'
    Display ' in EXEC PARM field: PARM='xxxxxxxx''
    Go to exit-now
end-if.
Move soket-initapi to ezaerror-function.
Move parm-tcpname to tcpname.
Call 'TPICLNID' using asname, subtask.
Call 'EZASOKET' using soket-initapi
    maxsoc
    initapi-ident
    subtask
    maxsno
    errno
    retcode.
If retcode < 0 then
    move 'Initapi failed' to ezaerror-text
    perform write-ezaerror-msg thru write-ezaerror-msg-exit
    go to exit-now.

*---------------------------------------------------------------*
* Let us see the client ID                                      *
*---------------------------------------------------------------*
move soket-getclientid to ezaerror-function.
Call 'EZASOKET' using soket-getclientid
    clientid
    errno
    retcode.
If retcode < 0 then
    move 'Getclientid failed' to ezaerror-text
    perform write-ezaerror-msg thru write-ezaerror-msg-exit
    go to exit-term-api.
Display 'Our client ID = ' clientid-name ' ' clientid-task.

*---------------------------------------------------------------*
* Get us a socket descriptor                                   *
*---------------------------------------------------------------*
move soket-socket to ezaerror-function.
Call 'EZASOKET' using soket-socket
    afinet
    soctype-stream
    proto
    errno
    retcode.
If retcode < 0 then
    move 'Socket call failed' to ezaerror-text
    perform write-ezaerror-msg thru write-ezaerror-msg-exit
    go to exit-term-api.
Move retcode to socket-descriptor.

*---------------------------------------------------------------*
* Bind socket to our server port number                        *
*---------------------------------------------------------------*
Move soket-bind to ezaerror-function.
Call 'EZASOKET' using soket-bind
    socket-descriptor
    server-socket-address
    errno
retcode.
If retcode < 0 then
move 'Bind call failed' to ezaerror-text
perform write-ezaerror-msg thru write-ezaerror-msg-exit
go to exit-close-socket.

*---------------------------------------------------------------*
* Issue passive open via Listen call *
*---------------------------------------------------------------*

move soket-listen to ezaerror-function.
Call 'EZASOKET' using soket-listen
socket-descriptor
backlog-queue
erro
retcode.
If retcode < 0 then
move 'Listen call failed' to ezaerror-text
perform write-ezaerror-msg thru write-ezaerror-msg-exit
go to exit-close-socket.

*---------------------------------------------------------------*
* Start iterative server loop with a blocking Accept Call *
*---------------------------------------------------------------*

iterative-server-loop.

move soket-accept to ezaerror-function.
Call 'EZASOKET' using soket-accept
socket-descriptor
client-socket-address
erro
retcode.
If retcode < 0 then
move 'Accept call failed' to ezaerror-text
perform write-ezaerror-msg thru write-ezaerror-msg-exit
go to exit-close-socket.
Move retcode to accepted-socket-descriptor.
Call 'TPINTOIA' using client-ipaddr client-ipaddr-dotted.
Display '***** New client connection *****'.
Display 'Client IP address = ' client-ipaddr-dotted.
Display ' and port number = ' client-port.

*---------------------------------------------------------------*
* Peek at first 8 bytes of client data *
*---------------------------------------------------------------*

Move 8 to read-request-len.
Move recv-flag-peek to recv-flag.
Perform read-TCP thru read-TCP-exit.
If retcode = zero then
Go to exit-close-a-socket.
If message-is-signon then
move 'E' to client-type
else
Call 'EZACIC05' using read-buffer
read-request-read
If message-is-signon then
move 'A' to client-type
else
Display 'First message from client is not sign-on'
Go to exit-close-a-socket
end-if
end-if.

*---------------------------------------------------------------*
* Receive signon message and issue RACF Verify via TPIRACF *
*---------------------------------------------------------------*

Move 40 to read-request-len.
Move recv-flag-read to recv-flag.
Perform read-TCP thru read-TCP-exit.
If retcode = zero then
   Go to exit-close-a-socket.
If client-is-ascii then
   Call 'EZACIC05' using read-buffer
      read-request-read
end-if.
Move zero to tpiracf-request.
Call 'TPIRACF' using tpiracf-request
   sign-on-userid
   sign-on-pwd
   sign-on-new-pwd
   sign-on-group
   tpiracf-application.
Move return-code to tpiracf-rc.
Move '*SIGNON*' to sign-on-reply-id.
Move tpiracf-rc to sign-on-rc.
Move 12 to send-request-len.
if client-is-ascii then
   Call 'EZACIC04' using send-buffer
      send-request-len
end-if.
Perform send-TCP thru send-TCP-exit.
if tpiracf-rc > 0 then
   Display 'Sign-on failed for user = ' sign-on-userid
   Display ' RACF RC = ' tpiracf-rc
   Go to exit-close-a-socket
end-if.
Display 'Sign-on successfull for user = ' sign-on-userid.

*---------------------------------------------------------------*
* Read 8192 block of client-data *
*---------------------------------------------------------------*

Move 0 to client-status.
Perform until client-has-closed
   move 8192 to read-request-len
   Perform read-TCP thru read-TCP-exit
   If retcode = 0 then
      Move 1 to client-status
   else
      Display 'Received 8K message from client'
      If client-is-ascii then
         Call 'EZACIC05' using read-buffer
            read-request-read
      end-if
   end-if
else
   Display 'We received a close-down message'
call 'TPIAUTH' using tpiiesrv-cls-resource
      tpiauth-read
   If return-code = 0 then
      Go to exit-close-socket
end-if

end-if
else
  Display 'User is not authorized to close server'
end-if
end-if

*---------------------------------------------------------------*
* Echo data back to client                                    *
*---------------------------------------------------------------*

  Move read-buffer to send-buffer
  If client-is-ascii then
    Call 'EZACIC04' using read-buffer
    read-request-read
  end-if
  move 8192 to send-request-len
  Perform send-TCP thru send-TCP-exit
  Display 'Echoed back 8K message to client'
end-if
end-perform.

*---------------------------------------------------------------*
* Delete security environment and close socket                 *
* Set 5 seconds linger time before close                        *
*---------------------------------------------------------------*

exit-delete-sec-env.
  Move 8 to tpiracf-request.
  Call 'TPIRACF' using tpiracf-request.
exit-close-a-socket.
  move soket-setsockopt to ezaerror-function.
  Call 'EZASOKET' using soket-setsockopt
    accepted-socket-descriptor
    setsockopt-linger
    setsockopt-value
    setsockopt-len
    errno
    retcode.
  If retcode < 0 then
    move 'Setsockopt Linger call failed' to ezaerror-text
    perform write-ezaerror-msg thru write-ezaerror-msg-exit
    Go to exit-close-socket.
  end-if
  move soket-close to ezaerror-function
  Call 'EZASOKET' using soket-close
    accepted-socket-descriptor
    errno
    retcode.
  If retcode < 0 then
    move 'Close call failed' to ezaerror-text
    perform write-ezaerror-msg thru write-ezaerror-msg-exit
    Go to exit-close-socket.
  end-if
  Display 'Finished processing one client'.
  Go to iterative-server-loop.

*---------------------------------------------------------------*
* Close listener socket and terminate                           *
*---------------------------------------------------------------*

exit-close-socket.
  move soket-close to ezaerror-function
  Call 'EZASOKET' using soket-close
    socket-descriptor
errno  
retcode.  
If retcode < 0 then  
    move 'Close call failed' to ezaerror-text  
    perform write-ezaerror-msg thru write-ezaerror-msg-exit.  
Display 'Listener socket closed'.  

*---------------------------------------------------------------*  
* Terminate socket API  
*---------------------------------------------------------------*  

exit-term-api.  
    Call 'EZASOKET' using soket-termapi.  

*---------------------------------------------------------------*  
* Terminate program  
*---------------------------------------------------------------*  

exit-now.  
    move zero to return-code.  
    Goback.  

*---------------------------------------------------------------*  
* Write out an error message  
*---------------------------------------------------------------*  

write-ezaerror-msg.  
    move errno to ezaerror-errno.  
    move retcode to ezaerror-retcode.  
    display ezaerror-msg.  
write-ezaerror-msg-exit.  
    exit.  

*---------------------------------------------------------------*  
* Subroutine:  
* -----------  
* *  
* *  
* * Read data from a TCP connection  
*---------------------------------------------------------------*  

Read-TCP.  
    move soket-recv to ezaerror-function.  
    move zero to read-request-read.  
    move read-request-len to read-request-remaining.  
    Perform until read-request-remaining = 0  
    Call 'EZASOKET' using soket-recv  
        accepted-socket-descriptor  
        recvv-flag  
        read-request-remaining  
        read-buffer-byte(read-request-read + 1)  
        errno  
        retcode  
        If retcode < 0 then  
            move 'Read call failed' to ezaerror-text  
            perform write-ezaerror-msg thru  
            write-ezaerror-msg-exit  
            go to exit-delete-sec-env  
        end-if  
        Add retcode to read-request-read  
        Subtract retcode from read-request-remaining  
        If retcode = 0 then  
            Move zero to read-request-remaining
Display 'Client closed socket connection'
end-if
end-perform.

Read-TCP-exit.
exit.

*---------------------------------------------------------------*
* Subroutine: *
* ----------- *
* * *
* * *
* Send data over a socket connection *
*---------------------------------------------------------------*

Send-TCP.
move soket-write to ezaerror-function.
move send-request-len to send-request-remaining.
move 0 to send-request-sent.
Perform until send-request-remaining = 0
Call 'EZASOKET' using soket-write
accepted-socket-descriptor
send-request-remaining
send-buffer-byte(send-request-sent + 1)
erno
retcode
If retcode < 0 then
move 'Write call failed' to ezaerror-text
perform write-ezaerror-msg thru
write-ezaerror-msg-exit
go to exit-delete-sec-env
end-if
add retcode to send-request-sent
subtract retcode from send-request-remaining
If retcode = 0 then
Display 'Client closed socket connection'
Move zero to send-request-remaining
end-if
end-perform.

Send-TCP-exit.
exit.

B.2 Sample Stream Socket COBOL Client

Identification Division.
*=================================* 
*---------------------------------* 
* Name: TPIIECLN - Client to test MVS iterative
* server TPIIESRV (TCP protocols).
* * *
* Function: This program connects to server on port 9997
* and sends a sign-on message including userid,
* password, optional new password and group id.
* Client receives sign-on reply. If sign-on is OK
* client sends 8K messages and receives them back
* from the server.
* If client EXEC PARM startup option is CLOSE, the
* client sends a server close-down message.
* * *
* Interface: Optional CLOSE option in JCL EXEC PARM field.
*
Program-id. tpiiecln.

*==================================*
Environment Division.
*==================================*

*=================
Data Division.
*=================

Working-storage Section.
*-------------------------------*

* Socket interface function codes *
*-------------------------------*
01 soket-functions.
   02 soket-accept pic x(16) value 'ACCEPT '.
   02 soket-bind pic x(16) value 'BIND '.
   02 soket-close pic x(16) value 'CLOSE '.
   02 soket-connect pic x(16) value 'CONNECT '.
   02 soket-fcntl pic x(16) value 'FCNTL '.
   02 soket-getclientid pic x(16) value 'GETCLIENTID '.
   02 soket-gethostbyaddr pic x(16) value 'GETHOSTBYADDR '.
   02 soket-gethostbyname pic x(16) value 'GETHOSTBYNAME '.
   02 soket-gethostid pic x(16) value 'GETHOSTID '.
   02 soket-gethostname pic x(16) value 'GETHOSTNAME '.
   02 soket-getpeername pic x(16) value 'GETPEERNAME '.
   02 soket-getsockname pic x(16) value 'GETSOCKNAME '.
   02 soket-getsockopt pic x(16) value 'GETSOCKOPT '.
   02 soket-givesocket pic x(16) value 'GIVESOCKET '.
   02 soket-initapi pic x(16) value 'INITAPI '.
   02 soket-ioctl pic x(16) value 'IOCTL '.
   02 soket-listen pic x(16) value 'LISTEN '.
   02 soket-read pic x(16) value 'READ '.
   02 soket-receive pic x(16) value 'RECV '.
   02 soket-receivefrom pic x(16) value 'RECVFROM '.
   02 soket-select pic x(16) value 'SELECT '.
   02 soket-send pic x(16) value 'SEND '.
   02 soket-sendto pic x(16) value 'SENDTO '.
   02 soket-setsockopt pic x(16) value 'SETSOCKOPT '.
   02 soket-shutdown pic x(16) value 'SHUTDOWN '.
   02 soket-socket pic x(16) value 'SOCKET '.
   02 soket-takesocket pic x(16) value 'TAKESOCKET '.
   02 soket-termapi pic x(16) value 'TERMAPI '.
   02 soket-write pic x(16) value 'WRITE '.

* Work variables *

A Beginner's Guide to MVS TCP/IP Socket Programming 179
01 errno pic 9(8) binary value zero.
01 retcode pic s9(8) binary value zero.
01 buffer-element.
   05 buffer-element-nbr pic 9(5).
   05 filler pic x(3) value space.
01 index-counter pic 9(8) binary value zero.
01 connect-status pic 9(4) Binary value zero.
   88 connect-done value 1.
01 server-ipaddr-dotted pic x(15) value space.
01 close-server pic 9(8) Binary value zero.
   88 send-close-server value 1.
*---------------------------------------------------------------*
* Variables used for the INITAPI call *
*---------------------------------------------------------------*
01 maxsoc pic 9(4) Binary Value 1.
01 initapi-ident.
   05 tcpname pic x(8) Value 'T18BTCP'.
   05 asname pic x(8) Value space.
01 subtask pic x(8) value space.
01 maxsno pic 9(8) Binary Value 1.
*---------------------------------------------------------------*
* Variables returned by the GETCLIENTID Call *
*---------------------------------------------------------------*
01 clientid.
   05 clientid-domain pic 9(8) Binary.
   05 clientid-name pic x(8) value space.
   05 clientid-task pic x(8) value space.
   05 filler pic x(20) value low-value.
*---------------------------------------------------------------*
* Variables used for the SOCKET call *
*---------------------------------------------------------------*
01 afinet pic 9(8) Binary Value 2.
01 soctype-stream pic 9(8) Binary Value 1.
01 proto pic 9(8) Binary Value zero.
01 socket-descriptor pic 9(4) Binary Value zero.
*---------------------------------------------------------------*
* Variables used for the GETHOSTBYNAME Call *
*---------------------------------------------------------------*
01 host-namelen pic 9(8) Binary Value 5.
01 host-name pic x(5) Value 'mvs18'.
01 host-entry-addr pic 9(8) Binary Value zero.
*---------------------------------------------------------------*
* Variables used for the call to EZACIC08 *
*---------------------------------------------------------------*
01 host-alias-seq pic 9(4) Binary Value zero.
01 host-addr-seq pic 9(4) Binary Value zero.
01 host-name-length pic 9(4) Binary Value zero.
01 host-name-value pic x(255) Value space.
01 host-alias-count pic 9(4) Binary Value zero.
01 host-alias-length pic 9(4) Binary Value zero.
01 host-alias-value pic x(255) Value space.
01 host-addr-type pic 9(4) Binary Value zero.
01 host-addr-length pic 9(4) Binary Value zero.
01 host-addr-count pic 9(4) Binary Value zero.
01 host-addr-value pic 9(8) Binary Value zero.
01 host-return-code pic s9(8) Binary Value zero.
*---------------------------------------------------------------*
* Variables used for the CONNECT Call *
*---------------------------------------------------------------*
01 server-socket-address.
   05 server-afinet pic 9(4) Binary Value 2.
A Beginner's Guide to MVS TCP/IP Socket Programming

05 server-port pic 9(4) Binary Value 9997.
05 server-ipaddr pic 9(8) Binary Value zero.
05 server-reserved pic x(8) value low-value.

*---------------------------------------------------------------*
* Buffer and length field for read operation                  *
*---------------------------------------------------------------*
01 recv-flag pic 9(8) Binary Value zero.
01 read-request-len pic 9(8) Binary Value zero.
01 read-request-read pic 9(8) Binary Value zero.
01 read-request-remaining pic 9(8) Binary Value zero.
01 read-buffer.
05 read-buffer-total pic x(8192) Value space.
05 sign-on-reply redefines read-buffer-total.
   10 sign-on-reply-id pic x(8).
   88 message-is-reply value '*SIGNON*'.
   10 sign-on-rc pic 9(4).
   10 filler pic x(8180).
05 read-buffer-byte redefines read-buffer-total
   pic x occurs 8192 times.

*---------------------------------------------------------------*
* Buffer and length fields for write operation                 *
*---------------------------------------------------------------*
01 send-request-len pic 9(8) Binary value zero.
01 send-request-sent pic 9(8) Binary value zero.
01 send-request-remaining pic 9(8) Binary value zero.
01 send-buffer.
05 send-buffer-total pic x(8192) value space.
05 sign-on-message redefines send-buffer-total.
   10 sign-on-message-id pic x(8).
   10 sign-on-userid pic x(8).
   10 sign-on-pwd pic x(8).
   10 sign-on-new-pwd pic x(8).
   10 sign-on-group pic x(8).
   10 filler pic x(8152).
05 close-down-message redefines send-buffer-total.
   10 close-down-message-id pic x(8).
   10 filler pic x(8184).
05 send-buffer-seq redefines send-buffer-total
   pic x(8) occurs 1024 times.
05 send-buffer-byte redefines send-buffer-total
   pic x occurs 8192 times.

*---------------------------------------------------------------*
* Error message for socket interface errors                    *
*---------------------------------------------------------------*
01 ezaerror-msg.
   05 filler pic x(9) Value 'Function='.
   05 ezaerror-function pic x(16) Value space.
   05 filler pic x value ' '.
   05 filler pic x(8) Value 'Retcode='.
   05 ezaerror-retcode pic ----99.
   05 filler pic x value ' '.
   05 filler pic x(9) Value 'Errorno='.
   05 ezaerror-errno pic zzz99.
   05 filler pic x value ' '.
   05 ezaerror-text pic x(50) value ' '.

Linkage Section.
*================
01 EXEC-parameter-field.
   05 parm-ll pic 9(4) Binary.
   05 parm-close-option pic x(5).
Procedure Division using EXEC-parameter-field.

If parm-ll < 5 then
  move zero to close-server
else
  If parm-close-option = 'CLOSE' then
    move 1 to close-server
  end-if
end-if.

*---------------------------------------------------------------*
* Initialize socket API  
*---------------------------------------------------------------*

Move soket-initapi to ezaerror-function.
Call 'TPICLNID' using asname subtask.
Call 'EZASOKET' using soket-initapi
  maxsoc
  initapi-ident
  subtask
  maxsno
  errno
  retcode.
If retcode < 0 then
  move 'Initapi failed' to ezaerror-text
  perform write-ezaerror-msg thru write-ezaerror-msg-exit
  go to exit-now.

*---------------------------------------------------------------*
* Let us see the client-id 
*---------------------------------------------------------------*

move soket-getclientid to ezaerror-function.
Call 'EZASOKET' using soket-getclientid
  clientid
  errno
  retcode.
If retcode < 0 then
  move 'Getclientid failed' to ezaerror-text
  perform write-ezaerror-msg thru write-ezaerror-msg-exit
  go to exit-term-api.
Display 'Our client ID = ' clientid-name ' ' clientid-task.

*---------------------------------------------------------------*
* Get host entry structure pointer based on host name 
*---------------------------------------------------------------*

move soket-gethostbyname to ezaerror-function.
Call 'EZASOKET' using soket-gethostbyname
  host-namelen
  host-name
  host-entry-addr
  retcode.
If retcode < 0 then
  move 'Gethostbyname failed' to ezaerror-text
  perform write-ezaerror-msg thru write-ezaerror-msg-exit
  go to exit-term-api.
Display 'Our client ID = ' clientid-name ' ' clientid-task.

*---------------------------------------------------------------*
* Get IP addresses out of the HOST Entry structure. 
*---------------------------------------------------------------*
A Beginner's Guide to MVS TCP/IP Socket Programming

* Loop pulling IP addresses out of the host entry structure, getting a socket and trying to connect to IP address.
* Loop until the returned list of IP addresses is exhausted or a connect is successful.
*
*---------------------------------------------------------------*
Move zero to connect-status.
Perform until ((host-addr-count = host-addr-seq and host-addr-seq > 0) or connect-done)
If host-alias-seq > host-alias-count then subtract 1 from host-alias-seq
end-if
move 'EZACIC08' to ezaerror-function
Call 'EZACIC08' using host-entry-addr
host-name-length
host-name-value
host-alias-count
host-alias-seq
host-alias-length
host-alias-value
host-addr-type
host-addr-length
host-addr-count
host-addr-seq
host-addr-value
host-return-code
If host-return-code < 0 then
move host-return-code to retcode
move 'Host translation failed' to ezaerror-text
perform write-ezaerror-msg thru write-ezaerror-msg-exit
go to exit-close-socket
end-if
Move host-addr-value to server-ipaddr
*
* Get an AF_INET socket to use for connect
*------------------------------------------------------------*
move soket-socket to ezaerror-function
Call 'EZASOKET' using soket-socket
afinet
soctype-stream
proto
errno
retcode
If retcode < 0 then
move 'Socket call failed' to ezaerror-text
perform write-ezaerror-msg thru write-ezaerror-msg-exit
go to exit-term-api
end-if
Move retcode to socket-descriptor
*
* Try to connect to iterative server on returned IP address
*------------------------------------------------------------*
If host-return-code = 0 then
Move socket-connect to ezaerror-function
Call 'TPIINTOA' using server-ipaddr
server-ipaddr-dotted
move 2 to server-afinet
move low-value to server-reserved
move 9997 to server-port
Call 'EZASOKET' using socket-connect
socket-descriptor
server-socket-address
errno
retcode
If retcode < 0 then
move space to ezaerror-text
Call 'TPIINTOA' using server-ipaddr
ezaerror-text
perform write-ezaerror-msg thru
write-ezaerror-msg-exit
move socket-close to ezaerror-function
Call 'EZASOKET' using socket-close
socket-descriptor
errno
retcode
If retcode < 0 then
move 'Close call failed' to ezaerror-text
perform write-ezaerror-msg thru
write-ezaerror-msg-exit
Go to exit-term-api
end-if
else
move 1 to connect-status
end-if
end-if
end-perform.

if not connect-done then
move 'Connection-loop' to ezaerror-function
move 'Connect failed' to ezaerror-text
perform write-ezaerror-msg thru
write-ezaerror-msg-exit
Go to exit-term-api.
end-if

Display 'Connected to server at ' server-ipaddr-dotted.

*---------------------------------------------------------------*
* Send sign-on message to server                                 *
* In this sample code, user id and password are hardcoded.       *
* In a real application, we would prompt the client user for    *
* these values.                                                  *
*---------------------------------------------------------------*
Move '*SIGNON*' to sign-on-message-id.
Move 'USERXX' to sign-on-userid.
Move '??????' to sign-on-pwd.
Move space to sign-on-new-pwd.
Move space to sign-on-group.
Move 40 to send-request-len.
Perform send-TCP thru send-TCP-exit.
If retcode = 0 then
Go to exit-close-socket.

*---------------------------------------------------------------*
* Receive sign-on reply from server                              *
*---------------------------------------------------------------*
Move 12 to read-request-len.
Perform read-TCP thru read-TCP-exit.
If retcode = 0 then
  Go to exit-close-socket.
If sign-on-rc not = 0 then
  Display 'Sign-on was unsuccessful'
  Go to exit-close-socket
else
  Display 'Successful sign-on, user = ' sign-on-userid
end-if.

* Initialize send buffer *
*---------------------------------------------------------------*
perform varying index-counter from 0 by 1
until index-counter > 1023
  move index-counter to buffer-element-nbr
  move buffer-element to send-buffer-seq(index-counter)
end-perform.

* If we are asked to close server down, we send a closedown *
* message and do not expect a response. *
*---------------------------------------------------------------*
If send-close-server then
  Display 'Sending close-down message to server'
  move '*CLSDWN*' to close-down-message-id
end-if.

move 8192 to send-request-len.
Perform send-TCP thru send-TCP-exit.
If send-close-server then
  Go to exit-close-socket.
If retcode = 0 then
  Go to exit-close-socket
else
  Display '8K Message sent to server'
end-if.

* Read server response *
*---------------------------------------------------------------*
move 8192 to read-request-len.
Perform read-TCP thru read-TCP-exit.
If retcode = 0 then
  Go to exit-close-socket
else
  Display 'Server returned 8K message'
end-if.

* Close socket *
*---------------------------------------------------------------*
exit-close-socket.
move soket-close to ezaerror-function
Call 'EZASOKET' using soket-close
    socket-descriptor
    errno
    retcode.
If retcode < 0 then
    move 'Close call failed' to ezaerror-text
    perform write-ezaerror-msg thru write-ezaerror-msg-exit.

*---------------------------------------------------------------*
* Terminate socket API                                         *
*---------------------------------------------------------------*
exit-term-api.
    Call 'EZASOKET' using soket-termapi.

*---------------------------------------------------------------*
* Terminate program                                           *
*---------------------------------------------------------------*
exit-now.
    move zero to return-code.
    Goback.

*---------------------------------------------------------------*
* Subroutine.                                                  *
*  Write out an error message                                 *
*---------------------------------------------------------------*
write-ezaerror-msg.
    move errno to ezaerror-errno.
    move retcode to ezaerror-retcode.
    display ezaerror-msg.
write-ezaerror-msg-exit.
exit.

*---------------------------------------------------------------*
* Subroutine:                                                  *
*  Read data from socket connection                           *
*---------------------------------------------------------------*
Read-TCP.
    move soket-recv to ezaerror-function.
    move zero to read-request-read.
    move read-request-len to read-request-remaining.
    Perform until read-request-remaining = 0
    Call 'EZASOKET' using soket-recv
        socket-descriptor
        recv-flag
        read-request-remaining
        read-buffer-byte(read-request-read + 1)
        errno
        retcode
    If retcode < 0 then
        move 'Read call failed' to ezaerror-text
        perform write-ezaerror-msg thru
            write-ezaerror-msg-exit
        go to exit-close-socket
end-if
Add retcode to read-request-read
Subtract retcode from read-request-remaining
If retcode = 0 then
  Display 'Server closed socket connection'
  Move zero to read-request-remaining
end-if
end-perform.
Read-TCP-exit.
exit.

*---------------------------------------------------------------*
* Subroutine: *
* ----------- *
* *
* *
* Send data over socket connection *
*---------------------------------------------------------------*

Send-TCP.
  move soket-write to ezaerror-function.
  move send-request-len to send-request-remaining.
  move 0 to send-request-sent.
  Perform until send-request-remaining = 0
  Call 'EZASOCKET' using soket-write
    socket-descriptor
    send-request-remaining
    send-buffer-byte(send-request-sent + 1)
    errno
    retcode
  If retcode < 0 then
    move 'Write call failed' to ezaerror-text
    perform write-ezaerror-msg thru
    write-ezaerror-msg-exit
    go to exit-close-socket
end-if
  add retcode to send-request-sent
  subtract retcode from send-request-remaining
If retcode = 0 then
  Display 'Server closed socket connection'
  Move zero to send-request-remaining
end-if
end-perform.
Send-TCP-exit.
exit.

B.3 Sample Stream Socket C Server

/* Portable socket server - (C) IBM - 1995 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef MVS
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>
#include <inet.h>
#include <socket.h>
#include <errno.h> /* required to make "errno" variable available */
#include <tcperrno.h>
#define tcperrno errno
A Beginner's Guide to MVS TCP/IP Socket Programming
#else
/* On OS/2, use SO32DLL.LIB TCP32DLL.LIB */
#define MAX_SEND_RECV 32767
#include <types.h>
#include <sys\socket.h>
#include <netinet\in.h>
#include <nerrno.h> /* sock_errno() */
#define close soclose
#define tcperror psock_errno
#define tcperrno sock_errno()
#endif
#include <netdb.h> /* should not precede #include <manifest.h> on MVS */
int check(char *text, int condition) /* if TRUE, error */
{
printf("%-9s ",text);
if (condition) {
tcperror("error");
return tcperrno ;
} else {
printf("completed OK.\n");
return 0 ;
} /* endif */
}
int sendRecord (int socketId , char * recordBuffer , unsigned long recordLength )
{
int
bytesSent
= 0
;
int
bytesToBeSent
;
char * remainingData = recordBuffer ;
int
remainingBytes = recordLength ;
while ( remainingBytes >0) {
#ifdef MAX_SEND_RECV
bytesToBeSent = min(remainingBytes,MAX_SEND_RECV);
#else
bytesToBeSent = remainingBytes ;
#endif
if (check("send",(bytesSent=send(socketId,remainingData,bytesToBeSent,0))<0)) return 1;
if (!bytesSent) { printf("Connection broken while sending.\n"); return 1 ; }
printf("%i bytes have been sent.\n",bytesSent);
remainingBytes -= bytesSent ;
remainingData += bytesSent ;
} /* endwhile */
printf("Complete record sent.\n");
return 0 ;
}
int receiveRecord (int socketId , char * recordBuffer , unsigned long recordLength )
{
int
bytesReceived
= 0
;
int
bytesToBeReceived
;
char * remainingData
= recordBuffer ;
int
remainingBytes
= recordLength ;
while ( remainingBytes >0) {
#ifdef MAX_SEND_RECV
bytesToBeReceived = min(remainingBytes,MAX_SEND_RECV);
#else

A Beginner's Guide to MVS TCP/IP Socket Programming

188


bytesToBeReceived = remainingBytes;
#endif
if (check("recv",(bytesReceived=recv(socketId,remainingData,bytesToBeReceived,0))<0)) {
    return 1;
} /* endif */
if (!bytesReceived) {
    printf("Connection broken while receiving.\n");
    return 1;
}
printf("%i bytes have been received.\n",bytesReceived);
remainingBytes -= bytesReceived;
remainingData += bytesReceived;
} /* endwhile */
printf("Complete record received.\n");
return 0;
}

int main(int argc,char**argv) {
{
    int socketId ;
    int newSocket ;
    struct sockaddr_in localAddress ;
    struct sockaddr_in clientAddress ;
    char * buffer ;
    unsigned long recordLength = 80;
    unsigned short port=9999;
    struct hostent * hostEnt ;
    int namelen=sizeof(struct sockaddr_in);

    setbuf(stdout,NULL); /* don't buffer: don't loose output in case of errors */

    if (argc>1) if (*argv[1]=='?') {
        printf("Parameters:\n"
            "1. port (default 9999)\n"
            "2. expected number of bytes (default 80).\n");
        return 0;
    } /* endif */

    if (argc>1) if (*argv[1]!=='*') port = atoi(argv[1]);
    if (argc>2) if (*argv[2]!=='*') recordLength = atoi(argv[2]);

    printf("port = %i\n"
        "record length = %i\n"
        , port
        , recordLength);

    if (!(( buffer = (char*)malloc(recordLength+1) ))) {
        printf("Insufficient storage to allocate buffer.\n");
        return 1;
    } /* endif */
#endif

    if (check("sock_init",sock_init())) return 1;

    /* create stream socket */
    if (check("socket",(socketId=socket(AF_INET,SOCK_STREAM,0))<0)) return 1;

    /* bind socket to any local address */
    localAddress.sin_family = AF_INET ;
    localAddress.sin_addr.s_addr = INADDR_ANY ;
    localAddress.sin_port = htons(port) ;

    if (check("bind",bind(socketId,(struct sockaddr*)&localAddress,namelen)<0 ) return 1 ;
if (check("listen",listen(socketId,1))) return 1;

if (check("accept",(newSocket=accept(socketId,(struct sockaddr*)&clientAddress,&namelen))<0)) {
    return 1;
} /* endif */

printf("Client: address: %s, port: %i.\n",
    (hostEnt=
        gethostbyaddr((char*)&clientAddress.sin_addr,sizeof(clientAddress.sin_addr),AF_INET))
    ?hostEnt->h_name/inet_ntoa(clientAddress.sin_addr),
        clientAddress.sin_port);

/* echo data to client */
if (receiveRecord (newSocket , buffer , recordLength )) return 1;
if (sendRecord (newSocket , buffer , recordLength )) return 1;

if (check("close newSocket" , close(newSocket))) return 1;
if (check("close socketId" , close(socketId ))) return 1;
return 0 ;
}

B.4 Sample Stream Socket C Client

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef MVS
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>
#include <inet.h>
#include <socket.h>
#include <tcperrno.h>
#define tcperrno errno
#include <errno.h> /* required to make "errno" variable available */
#else
/* On OS/2, use SO32DLL.LIB TCP32DLL.LIB */
#define MAX_SEND_RECV 32767
#include <types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <nerrno.h> /* sock_errno() */
#define close soclose
#define tcperror psock_errno
#define tcperrno sock_errno()
#endif
#include <netdb.h> /* should not precede #include <manifest.h> on MVS */

int check(char *text, int condition) /* if TRUE, error */
{
    printf("%-9s ",text);
    if (condition) {
        tcperror("error");
        return tcperrno ;
    } else {
        printf("completed OK.\n");
    }
int sendRecord (int socketId, char * recordBuffer, unsigned long recordLength)
{
    int bytesSent = 0;
    int bytesToBeSent;
    char * remainingData = recordBuffer;
    int remainingBytes = recordLength;

    while (remainingBytes > 0) {
        #ifdef MAX_SEND_RECV
            bytesToBeSent = min(remainingBytes, MAX_SEND_RECV);
        #else
            bytesToBeSent = remainingBytes;
        #endif
        if (check("send", (bytesSent = send(socketId, remainingData, bytesToBeSent, 0)) < 0)) return 1;
        if (!bytesSent) { printf("Connection broken while sending.\n"); return 1; }
        printf("%i bytes have been sent.\n", bytesSent);
        remainingBytes -= bytesSent;
        remainingData += bytesSent;
    } /* endwhile */
    printf("Complete record sent.\n");
    return 0;
}

int receiveRecord (int socketId, char * recordBuffer, unsigned long recordLength)
{
    int bytesReceived = 0;
    int bytesToBeReceived;
    char * remainingData = recordBuffer;
    int remainingBytes = recordLength;

    while (remainingBytes > 0) {
        #ifdef MAX_SEND_RECV
            bytesToBeReceived = min(remainingBytes, MAX_SEND_RECV);
        #else
            bytesToBeReceived = remainingBytes;
        #endif
        if (check("recv", (bytesReceived = recv(socketId, remainingData, bytesToBeReceived, 0)) < 0))
            return 1;
        if (!bytesReceived) { printf("Connection broken while receiving.\n"); return 1; }
        printf("%i bytes have been received.\n", bytesReceived);
        remainingBytes -= bytesReceived;
        remainingData += bytesReceived;
    } /* endwhile */
    printf("Complete record received.\n");
    return 0;
}

unsigned long * findAddresses (char * id)
{
    unsigned long binaryAddress;
    unsigned long * binaryAddresses;
    char * reason;
    struct hostent * hostEnt;

    if ((binaryAddress = inet_addr(id)) == INADDR_NONE) {
        if (!(hostEnt = gethostbyname(id))) {
            switch (h_errno) { 
            } /* endif */
        }
    }
}
case HOST_NOT_FOUND : reason = "host not found" ; break;
case TRY_AGAIN : reason = "try again" ; break;
case NO_RECOVERY : reason = "no recovery" ; break;
case NO_ADDRESS : reason = "no data/address" ; break;
default: reason = "?" ;
} /* endswitch */
printf("Gethostbyname for host \"%s\" failed, reason: %s.\n",id,reason);
return 0 ;
} /* endif */
return (unsigned long*) *hostEnt->h_addr_list ;
#else {
    binaryAddresses = (unsigned long *)calloc(2,sizeof(unsigned long));
binaryAddresses[0] = binaryAddress ; /* second entry terminates loop */
    return binaryAddresses ;
} /* endif */
}

int main(int argc, char**argv )
{
    int socketId
    int recordLength = 80 ;
    char * serverId
    unsigned short serverPort = 9999 ;
    struct sockaddr_in serverAddress ;
    unsigned long binaryAddress ;
    unsigned long * binaryAddresses ;
    struct sockaddr_in localAddress ;
    char * sendBuffer
    char * receiveBuffer
    int namelen = sizeof(localAddress);
    char * help =
      "Parameters:\n" "1. address server (dotted or symbolic) (no default)\n" "2. serverPort (default 9999)\n" "3. bytes to be sent (default 80).\n" ;
    setbuf(stdout,NULL); /* don't buffer: don't loose output in case of errors */
    if (argc<2) { printf(help); return 0; }
    if (argc>1) if (*argv[1]=='?') { printf(help); return 0; }
    if (argc>1) if (*argv[1]!='*') serverId = argv[1] ;
    if (argc>2) if (*argv[2]!='*') serverPort = atoi(argv[2]);
    if (argc>3) if (*argv[3]!='*') recordLength = atoi(argv[3]);
    printf("server id = %s\n" "server port = %i\n" "record length = %i\n" , serverId , serverPort , recordLength);
    if (! ( sendBuffer = (char*)malloc(2*recordLength) )) {
        printf("Insufficient storage to allocate buffers.\n");
        return 1;
    } /* endif */
    receiveBuffer = sendBuffer + recordLength ;

    memset( sendBuffer, 'A',recordLength); /* we will send a record full of A's */
    memset(receiveBuffer, 0 ,recordLength); /* blank out - to prevent mistakes */
#endif MVS
if (check("sock_init",sock_init())) return 1;
#endif

/* define fixed portion of server address */
serverAddress.sin_family = AF_INET ;
serverAddress.sin_port = htons(serverPort) ;

if (!(binaryAddresses= findAddresses ( serverId ))) return 1 ;

while ( binaryAddress = *binaryAddresses++ ) {

    /* get a new socket for each connect attempt */
    if (check("socket",(socketId=socket(AF_INET,SOCK_STREAM,0))<0)) return 1;
    serverAddress.sin_addr.s_addr = binaryAddress ;

    /* connect to server */
    printf("Trying to connect address %s\n",inet_ntoa(serverAddress.sin_addr));
    if (!check("connect",
            connect(socketId,(struct sockaddr*)&serverAddress,sizeof(serverAddress))<0)) break ;

    /* socket can not be reused after a failure */
    if (check("close",close(socketId))) return 1; /* close socket */
}

if (!binaryAddress) {
    printf("All known addresses of the specified server host were tried without success.\n") ;
    return 1 ;
} /* endif */

/* Find out where the system bound us */
if (check("getsockname",getsockname(socketId, (struct sockaddr *)&localAddress, &namelen))) {
    return 1 ;
} /* endif */

printf("Our own socket address: %s, port: %i.\n",
        inet_ntoa(localAddress.sin_addr),
        ntohs(localAddress.sin_port));

    /* send a message and receive echo */
if (sendRecord ( socketId , sendBuffer , recordLength )) return 1;
if (receiveRecord ( socketId , receiveBuffer , recordLength )) return 1;

    /* verify we received the same thing we sent */
if (memcmp(sendBuffer,receiveBuffer,recordLength)) {
    printf("Echo *NOT* correct.\n");
} else {
    printf("Echo is correct.\n");
} /* endif */

    if (check("close",close(socketId))) return 1;
    return 0 ;
}

C.0 Appendix C. Sample IMS Socket Programs

This appendix contains sample IMS socket programs that are developed in COBOL.

It also contains the sample IMS listener security exit that was used in the ITSO-Raleigh installation.
C.1 Dual Purpose Implicit Mode IMS Server Program
C.2 C Client Program to Test Dual Purpose IMS Server
C.3 Explicit Mode IMS Server Program
C.4 IMS Listener Security Exit

C.1 Dual Purpose Implicit Mode IMS Server Program

Identification Division.
*====================================================================*

* Name: TPIIMSDP - DI21PART database query program. *
* Function: Receives a part number, fetches data from the *
* DI21PART database and sends a message back. *
* Works for both MFS 3270 and implicit mode *
* IMS sockets. Dual-purpose IMS MPP. *
* *
* Interface: - none - *
* *
* Logic: 1. Receive input message *
* 2. Look up PARTROOT and STANINFO segments *
* 3. Format output message according to *
* defined layout *
* 4. Insert output message and terminate *
* *
* Returncode: - none - *
* *
* Written: March 7, 1995 at ITSO Raleigh *
* *
* Modified: *
* *
*====================================================================*

Program-id. TPIIMSDP.

*----------------------------------------------------------------------*

Environment Division.
*----------------------------------------------------------------------*

*----------------------------------------------------------------------*

Data Division.
*----------------------------------------------------------------------*

Working-storage Section.
*----------------------------------------------------------------------*

*----------------------------------------------------------------------*

Status messages
*----------------------------------------------------------------------*

01 partnumber-unknown pic x(79)
   Value 'Part number is not in database'.
01 staninfo-unknown pic x(79)
   Value 'Only basic information is available for part number'.
01 dli-unknown.
   05 filler pic x(11) Value 'DLI status='.
   05 status-dli pic x(2).
   05 filler pic x(10) Value ' Function='.
   05 status-function pic x(4).
   05 filler pic x(9) Value ' Segment='. 
   05 status-segment pic x(8).
A Beginner's Guide to MVS TCP/IP Socket Programming

05 filler pic x(1) Value space.
05 status-message pic x(34) Value space.
01 ioerr-unknown.
  05 filler pic x(11) Value 'DLI status='.
  05 ioerr-dli pic x(2).
  05 filler pic x(10) Value ' Function='.
  05 ioerr-function pic x(4).
  05 filler pic x(8) Value ' Assist='.
  05 ioerr-status pic x(6) Value space.
  05 filler redefines ioerr-status.
    10 ioerr-char pic x(2).
    10 filler pic x(4).
  05 ioerr-num redefines ioerr-status
    pic -99999.
  05 filler pic x(1) Value space.
  05 ioerr-message pic x(31) Value space.

*---------------------------------------------------------------*
* Work variables                                               *
*---------------------------------------------------------------*
01 dli-gu pic x(4) Value 'GU'.
01 dli-isrt pic x(4) Value 'ISRT'.
01 dli-gn pic x(4) Value 'GN'.
01 dli-gnp pic x(4) Value 'GNP'.

*---------------------------------------------------------------*
* SSA's for PARTROOT and STANINFO segments                    *
*---------------------------------------------------------------*
01 partroot-ssa.
  05 filler pic x(8) Value 'PARTROOT'.
  05 filler pic x(11) Value '(PARTKEY ='.
  05 filler pic x(2) Value '02'.
  05 partroot-key pic x(15) Value Space.
  05 filler pic x(1) Value ')'.
01 staninfo-ssa.
  05 filler pic x(8) Value 'STANINFO'.
  05 filler pic x(1) Value '.

*---------------------------------------------------------------*
* PARTROOT segment IO area                                    *
*---------------------------------------------------------------*
01 partroot-segment.
  05 filler pic x(2).
  05 partroot-partno pic x(15).
  05 filler pic x(9).
  05 partroot-descr pic x(20).
  05 filler pic x(4).

*---------------------------------------------------------------*
* STANINFO segment IO area                                    *
*---------------------------------------------------------------*
01 staninfo-segment.
  05 staninfo-proc-code pic x(2).
  05 staninfo-inv-code pic x(1).
  05 staninfo-rev-number pic x(2).
  05 filler pic x(24).
  05 staninfo-makedept pic x(2).
  05 staninfo-makecost pic x(2).
  05 filler pic x(2).
  05 staninfo-commodity-code pic x(4).
  05 filler pic x(4).
  05 filler pic x(25).
A Beginner's Guide to MVS TCP/IP Socket Programming

*-------------------------------------------------------------------*
* Terminal segment input/output area (MID and MOD) *
*-------------------------------------------------------------------*
  01 buffer.  
    05 buffer-ll pic 9(4) Binary.  
    05 buffer-zz pic 9(4) Binary.  
    05 input-buffer.  
      10 input-trancode pic x(8).  
      10 input-partno pic x(15).  
      10 filler pic x(102).  
    05 output-buffer redefines input-buffer.  
      10 output-partno pic x(15).  
      10 output-descr pic x(20).  
      10 output-proc-code pic x(2).  
      10 output-inv-code pic x(1).  
      10 output-revision-nbr pic x(2).  
      10 output-makedept pic x(2).  
      10 output-makecctr pic x(2).  
      10 output-commodity pic x(2).  
      10 output-status pic x(79).  

Linkage section.  
*-------------------------------------------------------------------*
* Input-Output PCB layout *
*-------------------------------------------------------------------*
  01 iopcb.  
    05 iopcb-lterm pic x(8).  
    05 iopcb-assist-status-bin pic s9(4) comp.  
    05 iopcb-assist-status-char redefines iopcb-assist-status-bin pic x(2).  
      88 iopcb-assist-aib-error value 'EA'.  
      88 iopcb-assist-buffer-full value 'EB'.  
      88 iopcb-assist-tim-only value 'EC'.  
    05 iopcb-status pic x(2).  
      88 iopcb-dli-stop value 'QC'.  
      88 iopcb-dli-ok value ' '.  
      88 iopcb-assist-error value 'ZZ'.  
    05 iopcb-cdate pic s9(7) comp-3.  
    05 iopcb-ctime pic s9(7) comp-3.  
    05 iopcb-input-msgno pic 9(8) binary.  
    05 iopcb-output-mod pic x(8).  
    05 iopcb-userid pic x(8).  

  01 altpcb1.  
    05 altpcb1-lterm pic x(8).  
    05 filler pic x(2).  
    05 altpcb1-status pic x(2).  

  01 altpcb2.  
    05 altpcb2-lterm pic x(8).  
    05 filler pic x(2).  
    05 altpcb2-status pic x(2).  

*-------------------------------------------------------------------*
* DI21PART PCB layout *
*-------------------------------------------------------------------*
  01 di21part-pcb.  
    05 filler pic x(10).  
    05 dbpcb-status pic x(2).  
      88 dbpcb-dli-ok Value ' '.  
      88 dbpcb-dli-not-found Value 'GE'.  

A Beginner's Guide to MVS TCP/IP Socket Programming
Procedure Division using iopcb, altpcb1, altpcb2, di21part-pcb.

*---------------------------------------------------------------*
* Receive one input segment.                                     *
*---------------------------------------------------------------*

Get-unique.

Call 'CBLADLI' using dli-gu
   iopcb
   buffer.
   If iopcb-dli-stop then
      go to exit-now.
   if not iopcb-dli-ok then
      move dli-gu to ioerr-function
      Perform io-error thru io-error-exit
      go to exit-now.

   Display 'buffer-ll = ' buffer-ll.
   Display 'buffer-zz = ' buffer-zz.
   Display 'input-trancode = ' input-trancode.
   Display 'input-partno = ' input-partno.

*---------------------------------------------------------------*
* Origin of input may be determined by analyzing the buffer-zz field. If it is zero, input has not been processed by MFS and originates from a socket client. If buffer-zz is 1,2 or 3 input has been processed by MFS and the value corresponds to the MFS option in effect. *
*---------------------------------------------------------------*

If buffer-zz = 0 then
   Display 'Input originates from socket client'
else
   Display 'Input originates from 3270 terminal'.
   Display 'iopcb-lterm = ' iopcb-lterm.
   Display 'iopcb-userid = ' iopcb-userid.

*---------------------------------------------------------------*
* Look up info in PARTROOT                                      *
*---------------------------------------------------------------*

move input-partno to partroot-key.
move space to output-buffer.
Call 'CBLADLI' using dli-gu
di21part-pcb
   partroot-segment
   partroot-ssa.
   Display 'GU partroot status = ' dbpcb-status.
   if dbpcb-dli-not-found then
      move partnumber-unknown to output-status
      go to isrt-output.
   if not dbpcb-dli-ok then
      move dli-gu to status-function
      perform db-error thru db-error-exit
      go to isrt-output.

*---------------------------------------------------------------*
* Look up info in STANINFO

Call 'CBLADLI' using dli-gnp
di21part-pcb
staninfo-segment
staninfo-ssa.
Display 'GNP staninfo status = ' dbpcb-status.
if dbpcb-dli-not-found then
  move partroot-partno to output-partno
  move partroot-descr to output-descr
  move staninfo-unknown to output-status
  go to isrt-output.
if not dbpcb-dli-ok then
  move dli-gnp to status-function
  perform db-error thru db-error-exit
  go to isrt-output.

* Build output segment

move partroot-partno to output-partno.
move partroot-descr to output-descr.
move staninfo.proc-code to output-proc-code.
move staninfo-rev-number to output-revision-nbr.
move staninfo-inv-code to output-inv-code.
move staninfo-makedept to output-makedept.
move staninfo-makecost to output-makecctr.
move staninfo-commodity-code to output-commodity.
move space to output-status.

* Send output segment

isrt-output.
move 150 to buffer-ll.
move zero to buffer-zz.
Display 'buffer-ll = ' buffer-ll.
Display 'buffer-zz = ' buffer-zz.
Display 'output buffer = ' output-buffer.

Call 'CBLADLI' using dli-isrt
iopcb
buffer.
Display 'ISRT output-buffer iopcb-status = ' iopcb-status.
if not iopcb-dli-ok then
  move dli-isrt to ioerr-function
  perform io-error thru io-error-exit
  go to exit-now.
  go to get-unique.

* Handle bad DLI status from a DB call

db-error.
  move dbpcb-status to status-dli.
  move dbpcb-segment-feedback to status-segment.
move 'DLI Call failed' to status-message.
move dli-unknown to output-status.
db-error-exit.
exit.

*---------------------------------------------------------------*
* Handle bad DLI status from an IO call *
*---------------------------------------------------------------*

io-error.
move iopcb-status to ioerr-dli.
move space to ioerr-status.
If iopcb-assist-error then
  if iopcb-assist-status-bin < 0 then
    move iopcb-assist-status-bin to ioerr-num
  else
    move iopcb-assist-status-char to ioerr-char
  end-if
move 'Socket error' to ioerr-message
else
  move space to ioerr-char
  move 'IO PCB call failed' to ioerr-message
end-if.
Display ioerr-unknown.
io-error-exit.
exit.

*---------------------------------------------------------------*
* Terminate program *
*---------------------------------------------------------------*
exit-now.
Goback.

C.2 C Client Program to Test Dual Purpose IMS Server

 campuses

/* TPIIMCDP - C client to test IMS dual-purpose implicit mode
 server program TPIIMSDP */
/*
 * Include Files.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
define lim 200
/*
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <netinet/socket.h>
#include <stdio.h>
*/
#ifndef __OS2__
#include <types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <string.h>
#else
#include <unistd.h> /* sock_errno() */
define close socket
#define tcerror psocket_errno
#endif

A Beginner's Guide to MVS TCP/IP Socket Programming
/* Client Main. */
main(int argc, char**argv) {
    /*
     * Transaction Request Message (TRM)
     * Sent by us to the IMS listener to initiate IMS transaction
     */
    struct TRM_message {
        unsigned short ll;
        unsigned short zz;
        char trnreq [8];
        char trancode [8];
        char userid [8];
        char pwd [8];
    } TRM;
    /*
     * Request Status Message (RSM)
     * Sent by the IMS Listener
     */
    struct RSM_message {
        unsigned short ll;
        unsigned short zz;
        char id [8];
        unsigned long rc;
        unsigned long reason;
    } RSM;
    /*
     * Completed Status Message (CSM)
     * Sent by the assist code
     */
    struct CSM_message {
        unsigned short ll;
        unsigned short zz;
        char csmoky [8];
    } CSM;
    /*
     * Segment buffer for sending and receiving data
     */
    struct segment_buffer {
        unsigned short ll;
        unsigned short zz;
        char buf [200];
    } segment;
    /*
     * Segment buffer for input segment to IMS
     */
    struct input_segment_buffer {
        unsigned short ll;
        unsigned short zz;
        char trancode [8];
        char partno [15];
    } input_segment;
/*
 * Segment buffer for output segment from IMS
 */
struct output_segment_buffer {
    unsigned short ll;
    unsigned short zz;
    char    partno [15];
    char    descr [20];
    char    proccode [2];
    char    invcode [1];
    char    revnbr [2];
    char    makedept [2];
    char    makecctr [2];
    char    commodity [2];
    char    status [79];
} output_segment;

unsigned short port; /* port client will connect to */
char buf [lim]; /* send receive buffer */
unsigned short lenbytes; /* Length field */
struct hostent *hostnm; /* server host name information */
struct sockaddr_in server; /* server address */
int s; /* client socket */
struct clientid ourclientid; /* Client ID structure */

/*
 * Check Arguments Passed. Should be hostname and port.
 */
if (argc != 3) {
    printf("Usage: %s hostname port\n", argv[0]);
    exit(1);
}
printf("Usage: %s hostname port\n", argv[0]);

/*
 * The host name is the first argument. Get the server address.
 */
hostnm = gethostbyname(argv[1]);
if (hostnm == (struct hostent *) 0) {
    printf("Gethostbyname failed\n");
    exit(2);
}

/*
 * The port is the second argument.
 */
port = (unsigned short) atoi(argv[2]);

/*
 * Build the TRM
 */
TRM.ll = htons(36);
TRM.zz = 0;
strcpy(TRM.trnreq, "*TRNREQ*");
strcpy(TRM.trancode,"TCP3");
strcpy(TRM.userid,"USER01");
strcpy(TRM.pwd,"????????");

/*
 * Put the server information into the server structure.
 * The port must be put into network byte order.
 */
server.sin_family = AF_INET;
server.sin_port = htons(port);
server.sin_addr.s_addr = *(unsigned long *)hostnm->h_addr;

/*
 * Get a stream socket.
*/
if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    tcperror("Socket()");
    exit(3);
}
printf("Socket sd = %d\n", s);

/*
 * Let us see our Client ID
*/
if (getclientid(AF_INET, &ourclientid) < 0) {
    tcperror("Getclientid()");
    exit(4);
}
printf("ClientID Jobname = %s\n", ourclientid.name);
printf("ClientID Subtaskname = %s\n", ourclientid.subtaskname);

/*
 * Connect to the server and send TRM
*/
if (connect(s, (struct sockaddr*) &server, sizeof(server)) < 0) {
    tcperror("Connect()");
    exit(4);
}
printf("Connected\n");
if (send(s, (char*) &TRM, sizeof(TRM), 0) < 0) {
    tcperror("Send() of TRM trancode = %s\n", TRM.trancode);
    exit(5);
}
printf("Send of TRM complete\n");
printf("TRM trancode = %s\n", TRM.trancode);

/*
 * Build transation input_segment and send it
*/
input_segment.ll = htons(sizeof(input_segment));
input_segment.zz = 0;
memcpy(input_segment.partno, "250794 ", 15);
if (send(s, (char*) &input_segment, sizeof(input_segment), 0) < 0) {
    tcperror("Send() of input_segment");
    exit(7);
}
printf("Send data complete\n");

/*
 * Send End of Message segment
*/
segment.ll = htons(4);
segment.zz = 0;
if (send(s, (char*) &segment, 4, 0) < 0) {
    tcperror("Send() of segment, 4, 0 < 0");
    exit(7);
}
printf("EOM segment sent\n");

/*
  * Receive first segment into buffer
  */
if (recv(s, (char*) &segment, 4, MSG_PEEK) < 0) {
tcperror("Recv() Peek for 4 bytes");
  exit(6);
}
lenbytes = ntohs(segment.ll);
printf("Bytes ready to read is %d\n", lenbytes);
if (recv(s, (char*)&segment, lenbytes, 0) < 0) {
tcperror("Recv()");
  exit(6);
}

if (!memcmp(buf, "*REQSTS*", 8)) {
  memcpy(&RSM, &segment, ntohs(segment.ll));
  printf("Receive of RSM complete\n");
  RSM.rc = ntohl(RSM.rc);
  RSM.reason = ntohl(RSM.reason);
  printf("RSM rc = %d\n", RSM.rc);
  printf("RSM reason code = %d\n", RSM.reason);
  if (RSM.rc > 0) {
    printf("Negative response in RSM message - rc=%d\n", RSM.rc);
    exit(12);
  }
  if (recv(s, (char*) &segment, 4, MSG_PEEK) < 0) {
    tcperror("Recv() Peek for 4 bytes");
    exit(6);
  }
  lenbytes = ntohs(segment.ll);
  printf("Bytes ready to read is %d\n", lenbytes);
  if (recv(s, (char*) &segment, lenbytes, 0) < 0) {
    tcperror("Recv()");
    exit(6);
  }
}

printf("Full output segment is %s\n", segment.buf);
memcpy(&output_segment, &segment, ntohs(segment.ll));
printf("Output data received\n");
printf("Partno = %s\n", output_segment.partno);
printf("Descr = %s\n", output_segment.descr);
printf("Proccode = %s\n", output_segment.proccode);
printf("Invcode = %s\n", output_segment.invcode);
printf("Revnbr = %s\n", output_segment.revnbr);
printf("Makedept = %s\n", output_segment.makedept);
printf("Makecctr = %s\n", output_segment.makecctr);
printf("Commodity = %s\n", output_segment.commodity);
printf("Status = %s\n", output_segment.status);

/*
  * Receive EOM message
  */
if (recv(s, (char*) &segment, 4, 0) < 0) {
tcperror("Recv()");
  exit(6);
}
printf("Receive of EOM segment complete");
/* Receive CSM message */
if (recv(s, (char*) &CSM, sizeof(CSM), 0) < 0) {
    tcperror("Recv()");
    exit(6);
}
printf("recv returned %s\n", CSM.csmoky);
if (!memcmp(CSM.csmoky, "*CSMOKY*", 8)) {
    printf("Receive of CSM complete: %s\n", CSM.csmoky);
}
/*
 * Close the socket.
 */
close(s);
printf("Client Ended Successfully\n");
exit(0);

C.3 Explicit Mode IMS Server Program

Identification Division.
*===========================================*
* Name: TPIIMSSE - IMS echo server, started via the *
* IMS Listener. *
* *
* Function: This program works as an echo server under IMS. *
* The program uses TCP protocols and is coded *
* in explicit mode. *
* The client used to test both EIMSTSRI and *
* TPIIMSSE is the same: EIMSTCLI. In order to *
* that, this explicit mode server uses the same *
* application protocol as the IMS assist modules *
* implement for an implicit mode server. *
* *
* Input messages are preceeded by a two byte *
* binary length field followed by two bytes with *
* binary zeroes. *
* The last input message is an EOM message and has *
* length field of 4. *
* llzz-message data- *
* llzz-more message data- *
* llzz (ll=4, EOM message) *
* *
* Output from this server uses the same format - *
* 2 length bytes in front of message and signals *
* end of output via an EOM segment with a length *
* of four. *
* First output message is a Request Status *
* Message with an rc=0. *
* It terminates the connection by sending a *
* CSM (Completed Status Message) to the client. *
* llzz-RSM- *
* llzz-message data- *
* llzz-more message data-
Program-id. TPIIMSSE.

Environment Division.

Data Division.

Working-storage Section.

* Socket interface function codes

01 soket-functions.
  02 soket-accept pic x(16) value 'ACCEPT'.
  02 soket-bind pic x(16) value 'BIND'.
  02 soket-close pic x(16) value 'CLOSE'.
  02 soket-connect pic x(16) value 'CONNECT'.
  02 soket-fcntl pic x(16) value 'FCNTL'.
  02 soket-getclientid pic x(16) value 'GETCLIENTID'.
  02 soket-gethostbyaddr pic x(16) value 'GETHOSTBYADDR'.
  02 soket-gethostbyname pic x(16) value 'GETHOSTBYNAME'.
  02 soket-gethostid pic x(16) value 'GETHOSTID'.
  02 soket-gethostname pic x(16) value 'GETHOSTNAME'.
  02 soket-getpeername pic x(16) value 'GETPEERNAME'.
  02 soket-getsockname pic x(16) value 'GETSOCKNAME'.
  02 soket-getsockopt pic x(16) value 'GETSOCKOPT'.
  02 soket-givesocket pic x(16) value 'GIVESOCKET'.
  02 soket-initapi pic x(16) value 'INITAPI'.
  02 soket-ioctl pic x(16) value 'IOCTL'.
  02 soket-listen pic x(16) value 'LISTEN'.
  02 soket-read pic x(16) value 'READ'.
  02 soket-recv pic x(16) value 'RECV'.
  02 soket-recvfrom pic x(16) value 'RECVFROM'.
  02 soket-select pic x(16) value 'SELECT'.
  02 soket-send pic x(16) value 'SEND'.
  02 soket-sendto pic x(16) value 'SENDTO'.
  02 soket-setsockopt pic x(16) value 'SETSOCKOPT'.
A Beginner’s Guide to MVS TCP/IP Socket Programming

02 soket-shutdown   pic x(16) value 'SHUTDOWN  '.
02 soket-socket     pic x(16) value 'SOCKET   '.
02 soket-takesocket pic x(16) value 'TAKESOCKET'.
02 soket-termapi    pic x(16) value 'TERMAPI  '.
02 soket-write      pic x(16) value 'WRITE   '.

*---------------------------------------------------------------*
* Work variables                                              *
*---------------------------------------------------------------*
01 errno           pic 9(8) binary value zero.
01 retcode         pic s9(8) binary value zero.
01 ezacic-len      pic 9(8) Binary Value zero.
01 dli-gu          pic x(4) Value 'GU'.
01 dli-isrt        pic x(4) Value 'ISRT'.

*---------------------------------------------------------------*
* Variables used for the INITAPI call                         *
*---------------------------------------------------------------*
01 maxsoc          pic 9(4) Binary Value 50.
01 apitype         pic 9(4) Binary Value 2.
01 initapi-ident.
  05 tcpname        pic x(8) Value space.
  05 myjobname      pic x(8) Value space.
01 subtask         pic x(8) value space.
01 maxsno          pic 9(8) Binary Value 1.

*---------------------------------------------------------------*
* Variables returned by the GETCLIENTID Call                  *
*---------------------------------------------------------------*
01 clientid-area.
  05 clientid-domain pic 9(8) Binary.
  05 clientid-name  pic x(8) value space.
  05 clientid-task  pic x(8) value space.
  05 filler         pic x(20) value low-value.

*---------------------------------------------------------------*
* Variables used by the TAKESOCKET Call                       *
*---------------------------------------------------------------*
01 take-from-clientid.
  05 take-from-domain pic 9(8) Binary Value 2.
  05 take-from-name  pic x(8) value space.
  05 take-from-task  pic x(8) value space.
  05 filler         pic x(20) value low-value.
01 socket-descriptor pic 9(4) Binary value zero.

*---------------------------------------------------------------*
* Transaction Initiation Message segment                      *
*---------------------------------------------------------------*
01 TIM-message.
  05 TIM-len        pic 9(4) Binary Value zero.
  05 filler         pic x(2) value low-value.
  05 TIM-id         pic x(8) value space.
  05 TIM-lstn-name  pic x(8) value space.
  05 TIM-lstn-task  pic x(8) value space.
  05 TIM-srv-name   pic x(8) value space.
  05 TIM-srv-task   pic x(8) value space.
  05 TIM-lstn-socketid pic 9(4) Binary value zero.
  05 TIM-tcpip-name pic x(8) value space.
  05 TIM-data-type  pic 9(4) Binary value zero.
  88 TIM-ascii      value 0.
  88 TIM-ebcdic     value 1.

*---------------------------------------------------------------*
* Transaction Request Status message segment                  *
*---------------------------------------------------------------*
01 RSM-message.
  05 RSM-len        pic 9(4) Binary Value 20.
  05 filler         pic x(2) value low-value.
05 RSM-oky   pic x(8) value 'REQSTS'.
05 RSM-return-code   pic 9(8) Binary Value zero.
05 RSM-reason-code   pic 9(8) Binary Value zero.

*---------------------------------------------------------------*
* Complete Status Message segment *
*---------------------------------------------------------------*
01 CSM-message.
 05 CSM-len   pic 9(4) Binary Value 12.
 05 filler   pic x(2) Value low-value.
 05 CSM-oky   pic x(8) value 'CSMOKY'.

*---------------------------------------------------------------*
* Peek buffer and length fields for RECV peek call *
*---------------------------------------------------------------*
01 recv-flag-read   pic 9(8) Binary value zero.
01 recv-flag-peek   pic 9(8) Binary value 2.
01 recv-flag   pic 9(8) Binary value 2.

*---------------------------------------------------------------*
* Buffer and length fields for read operation *
*---------------------------------------------------------------*
01 read-request-len   pic 9(8) Binary Value zero.
01 read-request-read   pic 9(8) Binary value zero.
01 read-request-remaining   pic 9(8) Binary Value zero.
01 read-buffer.
  05 read-buffer-total   pic x(8192) Value space.
  05 read-buffer-byte redefines read-buffer-total   pic x occurs 8192 times.
  05 read-buffer-segment redefines read-buffer-total.
     10 read-buffer-seg-len pic 9(4) Binary.
        88 EOM-segment value 4.
     10 read-buffer-seg-data pic x(8190).

*---------------------------------------------------------------*
* Buffer and length fields for write operation *
*---------------------------------------------------------------*
01 send-request-len   pic 9(8) Binary Value zero.
01 send-request-sent   pic 9(8) Binary value zero.
01 send-request-remaining   pic 9(8) Binary Value zero.
01 send-buffer.
  05 send-buffer-total   pic x(8192) value space.
  05 send-buffer-seq redefines send-buffer-total   pic x(8) occurs 1024 times.
  05 send-buffer-byte redefines send-buffer-total   pic x occurs 8192 times.

*---------------------------------------------------------------*
* Error message for socket interface errors *
*---------------------------------------------------------------*
01 ezaerror-msg.
  05 filler   pic x(9) Value 'Function='.
  05 ezaerror-function   pic x(16) Value space.
  05 filler   pic x value ' '.
  05 ezaerror-retnode   pic x(8) Value 'Retcode='.
  05 filler   pic x value ' '.
  05 ezaerror-errno   pic ---99.
  05 filler   pic x value ' '.
  05 ezaerror-errno   pic x(9) Value 'Errorno='.
  05 filler   pic x value ' '.
  05 ezaerror-text   pic x(11) Value 'DLI-status='.
  05 filler   pic x(2) value space.
  05 filler   pic x value ' '.
  05 ezaerror-dli-status   pic x(50) value ' '.

Linkage section.
<pre>01 iopcb. 
  05 iopcb-lterm pic x(8).
  05 filler pic x(2).
  05 iopcb-status pic x(2).
  05 iopcb-cdate pic s9(7) comp-3.
  05 iopcb-ctime pic s9(7) comp-3.
  05 iopcb-input-msgno pic 9(8) binary.
  05 iopcb-output-mod pic x(8).
  05 iopcb-userid pic x(8).

01 altpcb1.
  05 altpcb1-lterm pic x(8).
  05 filler pic x(2).
  05 altpcb1-status pic x(2).

*========================================*
Procedure Division using iopcb, altpcb1.
*========================================*

* Receive TIM from listener *
*----------------------------------*

Get-unique.
  Call 'CBLTDLI' using dli-gu
    iopcb
    TIM-message.
  If iopcb-status = 'QC' then
    go to exit-now.
  if iopcb-status not equal ' ' then
    move 'IOPCB Get-Unique' to ezaerror-function
    move iopcb-status to ezaerror-dli-status
    perform write-ezaerror-msg thru write-ezaerror-msg-exit
    go to exit-now.

* Initialize socket API with the values we got from *
* the IMS Listener *
*----------------------------------*

  Move soket-initapi to ezaerror-function.
  Move TIM-srv-name to myjobname.
  Move TIM-srv-task to subtask.
  Move TIM-tcip-name to tcpname.
  Display 'Initapi myjobname=' myjobname
     ' subtask=' subtask.
  Call 'EZASOKET' using soket-initapi 
    maxsoc
    initapi-ident
    subtask
    maxsno
    errno
    retcode.
  If retcode < 0 then
    move 'Initapi failed' to ezaerror-text
    perform write-ezaerror-msg thru write-ezaerror-msg-exit
    go to exit-now.
</pre>
* Issue a getclientid to take a look at the actual * 
* clientid we are running under  *
*---------------------------------------------------------------*
move soket-getclientid to ezaerror-function.
Call 'EZASOKET' using soket-getclientid
  clientid-area
  errno
  retcode.
If retcode < 0 then
  move 'Getclientid failed' to ezaerror-text
  perform write-ezaerror-msg thru write-ezaerror-msg-exit
go to exit-term-api.
Display 'Getclientid returned Domain=' clientid-domain.
Display ' Address space name=' clientid-name.
Display ' Subtask name=' clientid-task.
*---------------------------------------------------------------*
* Issue a take-socket with the values we got from *
* the IMS Listener *
*---------------------------------------------------------------*
move soket-takesocket to ezaerror-function.
move TIM-lstn-name to take-from-name.
move TIM-lstn-task to take-from-task.
Display 'TIM-message=' TIM-message.
Display 'Takesocket from-name=' take-from-name
  ' from-task=' take-from-task.
Call 'EZASOKET' using soket-takesocket
  TIM-lstn-socketid
  take-from-clientid
  errno
  retcode.
If retcode < 0 then
  move 'Takesocket failed' to ezaerror-text
  perform write-ezaerror-msg thru write-ezaerror-msg-exit
go to exit-term-api.
move retcode to socket-descriptor.
*---------------------------------------------------------------*
* Send an OK Request Status Message to the client. *
* We have been started via the IMS Listener *
*---------------------------------------------------------------*
If TIM-ascii then
  Move 8 to ezacic-len
  Call 'EZACIC04' using RSM-oky
  ezacic-len.
  Move RSM-message to send-buffer.
  Move RSM-len to send-request-len.
  Perform send-tcp thru send-tcp-exit.
If send-request-sent < 0 then
  move 'Write RSM failed' to ezaerror-text
  perform write-ezaerror-msg thru write-ezaerror-msg-exit
go to exit-close-socket.
*---------------------------------------------------------------*
* Peek at first bytes of client data *
*---------------------------------------------------------------*
Perform until EOM-segment
Move 2 to read-request-len
Move recv-flag-peek to recv-flag
Perform read-tcp thru read-tcp-exit
if read-request-read < 0 then
  Display 'Peek failed'
go to exit-close-socket
end-if

*---------------------------------------------------------------*
* Read client data                                           *
*---------------------------------------------------------------*
move read-buffer-seg-len to read-request-len
move recv-flag-read to recv-flag
Perform read-tcp thru read-tcp-exit
If read-request-read < 0 then
  move 'Read failed' to ezaerror-text
  perform write-ezaerror-msg thru
  write-ezaerror-msg-exit
  go to exit-close-socket
end-if

*---------------------------------------------------------------*
* Echo data back to client                                    *
*---------------------------------------------------------------*
move read-buffer to send-buffer
move read-request-read to send-request-len
Perform send-tcp thru send-tcp-exit
If send-request-sent < 0 then
  move 'Send failed' to ezaerror-text
  perform write-ezaerror-msg thru
  write-ezaerror-msg-exit
  go to exit-close-socket
end-if
end-perform.

*---------------------------------------------------------------*
* Send a transaction Completed Status Message to the        *
* client                                                    *
*---------------------------------------------------------------*
If TIM-ascii then
  Move 8 to ezacic-len
  Call 'EZACIC04' using CSM-oky
  ezacic-len.
move CSM-len to send-request-len.
move CSM-message to send-buffer.
Perform send-tcp thru send-tcp-exit
If send-request-sent < 0 then
  move 'Send CSM failed' to ezaerror-text
  perform write-ezaerror-msg thru
  write-ezaerror-msg-exit
  go to exit-close-socket.

*---------------------------------------------------------------*
* Close the socket                                           *
*---------------------------------------------------------------*
exit-close-socket.
move soket-close to ezaerror-function
Call 'EZASOKET' using soket-close
   socket-descriptor
   errno
   retcode.
If retcode < 0 then
   move 'Close call failed' to ezaerror-text
   perform write-ezaerror-msg thru write-ezaerror-msg-exit.

*---------------------------------------------------------------*
* Terminate socket API and request next TIM from IMS          *
*---------------------------------------------------------------*
exit-term-api.
   Call 'EZASOKET' using soket-termapi.
   Go to get-unique.

*---------------------------------------------------------------*
* Terminate program                                          *
*---------------------------------------------------------------*
exit-now.
   Goback.

*---------------------------------------------------------------*
* Subroutine                                                 *
* Write out an error message                                 *
*---------------------------------------------------------------*
write-ezaerror-msg.
   move errno to ezaerror-errno.
   move retcode to ezaerror-retcode.
   display ezaerror-msg.
write-ezaerror-msg-exit.
   exit.

*---------------------------------------------------------------*
* Subroutine                                                 *
* Read data from a TCP socket                                *
*---------------------------------------------------------------*
Read-TCP.
   move soket-recv to ezaerror-function.
   move zero to read-request-read.
   move read-request-len to read-request-remaining.
   Perform until read-request-remaining = 0
      Display 'Ready for new read'
      Display 'Number of bytes remaining=' read-request-remaining
      Display 'Number of bytes read until now=' read-request-read
   Call 'EZASOKET' using soket-recv
      socket-descriptor
      recv-flag
      read-request-remaining
      read-buffer-byte(read-request-read + 1)
      errno
      retcode
      Display 'Read returned rc=' retcode
If retcode < 0 then
  move 'Read call failed' to ezaerror-text
  perform write-ezaerror-msg thru
    write-ezaerror-msg-exit
  go to exit-close-socket
end-if
Display 'Number of bytes read=' retcode
Add retcode to read-request-read
Subtract retcode from read-request-remaining
If retcode = 0 then
  Display 'End-of-data received too early'
  Display 'Server probably closed socket'
  Move zero to read-request-remaining
end-if
end-perform.
Read-TCP-exit.

*---------------------------------------------------------------*
* Subroutine *                                                   *
* ---------- *                                                   *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* Send data over a TCP socket *                                  *
* --------------------------------------------------------------*

Send-TCP.
  move soket-write to ezaerror-function.
  move send-request-len to send-request-remaining.
  move 0 to send-request-sent.
  Perform until send-request-remaining = 0
    Display 'Ready for new write'
    Display 'Number of bytes remaining='
      send-request-remaining
    Display 'Number of bytes sent until now='
      send-request-sent
    Call 'EZASOKET' using soket-write
      socket-descriptor
      send-request-remaining
      send-buffer-byte(send-request-sent + 1)
     erno
      retcode
    Display 'Write returned an rc=' retcode
  If retcode < 0 then
    move 'Write call failed' to ezaerror-text
    perform write-ezaerror-msg thru
      write-ezaerror-msg-exit
    go to exit-close-socket
  end-if
  Display 'Number of bytes written=' retcode
  add retcode to send-request-sent
  subtract retcode from send-request-remaining
end-perform.
Send-TCP-exit.
exit.

C.4 IMS Listener Security Exit

**************************************************************************
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* Name: IMLSECX - IMS Sockets Listener security exit. *               *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
A Beginner's Guide to MVS TCP/IP Socket Programming 212
* Function: Validate stream socket connections to IMS. *
* Interface: R1 -> parameter list with eight pointers: *
  +0 -> Fullword IP Address (In)
  +4 -> Halfword port number (In)
  +8 -> 8 byte IMS transaction code name (In)
  +12 -> Halfword datatype (0, ASCII, 1 EBCDIC) (In)
  +16 -> Fullword length of user data in TRM (In)
  +20 -> User data (In)
  +24 -> Fullword return code (Out)
  +28 -> Fullword reason code (Out)
* Security exit interface contains user data. User data is installation-defined, in our case as 32 bytes with the following layout:
  8 bytes user ID
  8 bytes password
  8 bytes new password (optional)
  8 bytes RACF group ID (optional)
* Logic: 1. Validates if all required parms are present.
  2. Calls TPIRACF for user authentication and creation of task level security environment.
  3. Authorizes user's access to requested IMS tran code via call to TPIAUTH for resource class FACILITY and resource TPI.IMSSOCK.trancode.
  4. Deletes user security environment again and returns to IMS listener.
* Abends: User abend 1001: If RACROUTE REQUEST=DELETE fails and we do not know if we are continuing under a user security environment, we abend.
* Return codes: Return and reason codes set in the IMS listener security exit interface area.
  RC=000 Reason=000: User authenticated OK and user's access to tran code authorized OK.*
  RC=008 Reason=101: UserID and password missing in TRM*
  RC=008 Reason=102: Invalid length of userdata in TRM *
  RC=008 Reason=103: UserID not defined to RACF *
  RC=008 Reason=104: Invalid password *
  RC=008 Reason=105: Password has expired *
  RC=008 Reason=106: New password is not valid *
  RC=008 Reason=107: User does not belong to group *
  RC=008 Reason=108: User is revoked *
  RC=008 Reason=109: Access to group is revoked *
  RC=008 Reason=110: User not authorized to IMS Sockets*
  RC=008 Reason=111: TPIRACF internal error *
  RC=008 Reason=112: TPIRACF internal error *
  RC=008 Reason=113: User not authorized to tran code *
* Written: ITSO, Raleigh April 16, 1995

*******************************************************************************

INTFAREA DSECT
LSIPADDR DC A(0)  *-- Client IP address
LSPORT  DC A(0)  *-- Client port number
LSTRNNAM DC A(0)  *-- IMS transaction code name
LSDATTyp DC A(0)  *-- Datatype (0 ASCII, 1 EBCDIC)
A Beginner's Guide to MVS TCP/IP Socket Programming

LSDATLEN DC A(0) *-> Length of user data in TRM
LSUSRDAT DC A(0) *-> User data area
LSRETCOD DC A(0) *-> Return code field
LSREACOD DC A(0) *-> Reason code field

USERDATA DSECT
LSUSERID DC CL8' ' *User ID
LSPWD DC CL8' ' *Password
LSNPWD DC CL8' ' *New Password (Optional)
LSGROUP DC CL8' ' *Group ID (Optional)

IMSLSECX INIT 'IMS Sockets Listener security exit',MODE=31

LR R11,R1 *Parameter pointer
USING INTFAREA,R11 *Addressability of parameters

* ---------------------------------------------------------------------
* Check passed parameters and do any necessary conversion from
* ASCII to EBCDIC.
* ---------------------------------------------------------------------

L R2,LSDATLEN *-> Fullword with L'userdata
L R9,0(R2) *L'userdata
C R9,=A(16) *User ID and Password must be there
BL TOFEWPRM *Too few parameters passed
BE PARMOK *Only user ID and password is OK
C R9,=A(24) *Exactly 24 bytes long
BE PARMOK *- is OK - new password
C R9,=A(32) *Or exactly 32 bytes long
BNE LENERR *- is OK, if anything else: error

PARMOK EQU *

L R3,LSDATTPY *-> Halfword with datatype
L R4,LSUSRDAT *-> Userdata
LH R9,0(R3) *Datatype
LTR R9,R9 *Is data ASCII ?
BNZ ISEBCDIC *- No, data is EBCDIC
CALL EZACIC05,((R4),(R2)),VL *Translate ASCII to EBCDIC

ISEBCDIC EQU *

* Build TPIRACF parameters and call TPIRACF to verify user.
* ---------------------------------------------------------------------

MVC USERID(32),=CL32' ' *Initialize all parms to space
MVC REQCODE,=A(REQVER) *Issue RACROUTE REQUEST=VERIFY
USING USERDATA,R4 *User data area addressability
MVC USERID,LSUSERID *User ID from TRM
MVC PWD,LSPWD *Password from TRM
L R9,0(R2) *L'userdata
C R9,=A(16) *Is there a new password ?
BNH DOVER *- No, all parms are set
MVC NPWD,LSNPWD *New password from TRM
C R9,=A(24) *Is there a group ID ?
BNH DOVER *- No, all parms are set
MVC GROUP,LSGROUP *Group ID from TRM

DOVER EQU *

CALL TPIRACF, *
(REQCODE, *RACROUTE REQUEST=VERIFY C
A Beginner's Guide to MVS TCP/IP Socket Programming

USERID, * C
PWD, * C
NPWD, * C
GROUP, * C
APPLNAME),VL
LTR R15,R15 *Was VERIFY Successful?
BNZ VERFAIL *-- No, return error to client.

*----------------------------------------------------------------------------*
* Build TPIAUTH parameters and call TPIAUTH to test if user is authorized to TPI.IMSSOCK.trancode in the FACILITY resource class.
*----------------------------------------------------------------------------*

L R2,LSTRNNAM *-> IMS Transaction code name
MVC RESTRNNM,0(R2) *Move to FACILITY Class resource nm.
CALL TPIAUTH, *Authorize call C (RESNAME, *Resource name C AUTHACC),VL *Test for read access
ST R15,AUTHRC *Save RC for a little later

*----------------------------------------------------------------------------*
* Delete user security environment again, so we restore address space security environment before we return to the IMS Listener.
*----------------------------------------------------------------------------*

MVC PWD(24),=CL24' ' *Space out unneeded parms
MVC REQCODE,=A(REQDEL) *We want to delete sec. environment
LTR R15,R15 *This should only give RC=0
BZ DELOK *-- which it did
LR R9,R15 *Save RC
CVD R15,DORD *Convert
OI DORD+7,X'0F' *- to something
UNPK WTODELRC,DORD *- readable in a WTO
WTO MF=(E,WTODEL) *Tell about it
CH R9,=AL2(253) *Does not leave a security env.
BE DELOK *-- which is OK
WTO 'IMSLSECX - User abend 1001 due to above return code'
ABEND 1001,DUMP *Others may leave user sec. active.
DELOK EQU *
ICM R9,B'1111',AUTHRC *Return code from AUTH call
BNZ AUTHFAIL *If not zero, auth failed.
SR R15,R15 *Set RC=0
SR R10,R10 *-- and Reason code=0
B RETURN *And exit

*----------------------------------------------------------------------------*
* Error exit routines. Set R15 to return code and R10 to reason code and go to common exit code.
*----------------------------------------------------------------------------*
* AUTHFAIL EQU * *User is not authorized
  LA R10,NOTAUTH *Not authorized to tran code
  LA R15,8 *This is an error
  B RETURN *And exit
* VERFAIL EQU * *User did not verify successfully
  LM R5,R7,RCBXLE *Prepare to set reason code
* RLOOP EQU *
  CH R15,0(R5) *This TPIRACF Return code ?
  BE SETREAS *-- Yes, set corresponding reason
  BXLE R5,R6,RCLOOP *We use last entry as garbage can
* SETREAS EQU *
  LH R10,2(R5) *Here is corresponding reason code
  LA R15,8 *This is an error
  B RETURN *And exit
* TOFEWRM EQU *
  LA R15,8 *This is an error
  LA R10,PARMERR1 *To few parameters
  B RETURN *Exit
* LENERR EQU *
  LA R15,8 *This is an error
  LA R10,PARMERR2 *Wrong length
* RETURN EQU *
  L R2,LSRETCOD *--> Return code field
  ST R15,0(R2) *Pass back return code
  L R2,LSREACOD *--> Reason code field
  ST R10,0(R2) *Pass back reason code
  TERM RC=0 *Return to IMS Listener
* LTORG

* Work areas and constants.
* 
* Reason codes
*
* PARMERR1 EQU 101 *At least user ID and password req.
* PARMERR2 EQU 102 *Length must be 16, 24 or 32.
* NOTAUTH EQU 113 *User not authorized to tran code
* 
* RCBXLE DC A(START,4,LAST) *TPIRACF RC to Reason code convert.
* START DC AL2(4,103) *User ID not defined to RACF
  DC AL2(8,104) *Invalid password
  DC AL2(12,105) *Password has expired
  DC AL2(16,106) *New password is not valid
  DC AL2(20,107) *User ID does not belong to group
  DC AL2(24,108) *User ID is revoked
  DC AL2(28,109) *Access to group is revoked
  DC AL2(32,110) *User ID is not authorized to appl
  DC AL2(254,111) *Internal error
  LAST DC AL2(255,112) *Some other error
* 
* REQCODE DC A(0) *TPIRACF Request Code
* REQVER EQU 0 *REQUEST=VERIFY
* REQDEL EQU 8 *REQUEST=DELETE
* USERID DC CL8' ' *User ID
* PWD DC CL8' ' *Password
* NPWD DC CL8' ' *New password
* GROUP DC CL8' ' *Group ID
D.0 Appendix D. Sample CICS Socket Program

This appendix contains sample CICS socket programs that are developed in COBOL and C.

D.1 Stream Socket COBOL Program for CICS

D.2 C Version of EZACICSC

D.1 Stream Socket COBOL Program for CICS

Identification Division.
*========================*
*---------------------------------------------------------------*
* 
* Name: TPICICSS - CICS echo server program that is *
* started via the CICS Listener. *
* CICS transaction code TPIE. *
* 
* Function: This is a stream socket program. The server *
* is started via the TPIE CICS transaction code. *
* It will echo back to the client any data the *
* client sends to it. It will close the socket *
* and terminate when the client closes its socket. *
* If the client is quiet for more than 30 seconds, *
* the server will timeout and close the *
* connection. *
* 
* Interface: CICS Listener Transaction Initiation Message *
* 
* Logic: 1. Receive TIM from CICS listener *
* 2. Initialize API and takesocket *
* 3. Enter a read/write loop where data will be *
* echoed back to the client *
* Socket is set to non-blocking in order to *
* control own timeout logic *
* 4. If no data from client within 30 seconds, the *
* server closes the connection and terminates *
* 
* Returncode: - none - *
* 
* Written: March 8, 1995 at ITSO Raleigh *
* 
* Modified: 
* 
A Beginner's Guide to MVS TCP/IP Socket Programming 217
Program-id. tpicicss.

Environment Division.

Data Division.

Working-storage Section.

* Socket interface function codes *

01 soket-functions.
  02 soket-accept pic x(16) value 'ACCEPT'.
  02 soket-bind pic x(16) value 'BIND'.
  02 soket-close pic x(16) value 'CLOSE'.
  02 soket-connect pic x(16) value 'CONNECT'.
  02 soket-fcntl pic x(16) value 'FCNTL'.
  02 soket-getclientid pic x(16) value 'GETCLIENTID'.
  02 soket-gethostbyaddr pic x(16) value 'GETHOSTBYADDR'.
  02 soket-gethostbyname pic x(16) value 'GETHOSTBYNAME'.
  02 soket-gethostid pic x(16) value 'GETHOSTID'.
  02 soket-gethostname pic x(16) value 'GETHOSTNAME'.
  02 soket-getpeername pic x(16) value 'GETPEERNAME'.
  02 soket-getsockname pic x(16) value 'GETSOCKNAME'.
  02 soket-getsockopt pic x(16) value 'GETSOCKOPT'.
  02 soket-givesocket pic x(16) value 'GIVESOCKET'.
  02 soket-initapi pic x(16) value 'INITAPI'.
  02 soket-ioctl pic x(16) value 'IOCTL'.
  02 soket-listen pic x(16) value 'LISTEN'.
  02 soket-read pic x(16) value 'READ'.
  02 soket-recev pic x(16) value 'RECV'.
  02 soket-recevfrom pic x(16) value 'RECVFROM'.
  02 soket-select pic x(16) value 'SELECT'.
  02 soket-send pic x(16) value 'SEND'.
  02 soket-sendto pic x(16) value 'SENDTO'.
  02 soket-setssockopt pic x(16) value 'SETSOCKOPT'.
  02 soket-shutdown pic x(16) value 'SHUTDOWN'.
  02 soket-socket pic x(16) value 'SOCKET'.
  02 soket-takesocket pic x(16) value 'TAKEAPI'.
  02 soket-termapi pic x(16) value 'TERAPI'.
  02 soket-write pic x(16) value 'WRITE'.

* Work variables *

01 errno pic 9(8) binary value zero.
01 retcode pic s9(8) binary value zero.
01 cleng pic s9(4) binary value zero.
01 socket-to-take pic s9(4) binary value zero.
01 client-ipaddr-dotted pic x(15) value space.
01 client-status pic 9(8) binary value zero.
  88 client-has-closed Value 1.
01 timer-accum pic 9(8) binary value zero.

* Variables used for the INITAPI call *

01 maxsoc pic 9(4) binary Value 2.
01 initapi-ident.
05 tcpname  pic x(8) Value ' '.
05 asname  pic x(8) Value space.
01 subtask.
  05 init-cics-task  pic 9(7).
  05 filler  pic x value 'I'.
01 maxsno  pic 9(8) Binary Value zero.

*---------------------------------------------------------------*
* Variables returned by the GETCLIENTID Call - our clientid    *
*---------------------------------------------------------------*
01 clientid.
  05 clientid-domain  pic 9(8) Binary.
  05 clientid-name  pic x(8) value space.
  05 clientid-task  pic x(8) value space.
  05 filler  pic x(20) value low-value.

*---------------------------------------------------------------*
* Variables used for the IOCTL call                            *
*---------------------------------------------------------------*
01 ioctl-command-fionbio  pic x(4).
01 ioctl-command-string  pic x(16) value 'FIONBIO'.
01 ioctl-reqarg-non-blocking  pic 9(8) Binary value 1.
01 ioctl-retarg  pic 9(8) binary value zero.

*---------------------------------------------------------------*
* CICS Listener client ID used in the TAKESOCKET call          *
*---------------------------------------------------------------*
01 clientid-lstn.
  05 cid-domain-lstn  pic 9(8) binary.
  05 cid-name-lstn  pic x(8) value space.
  05 cid-subtask-lstn  pic x(8) value space.
  05 cid-res-lstn  pic x(20) value low-value.

*---------------------------------------------------------------*
* Variables used for the SOCKET call                           *
*---------------------------------------------------------------*
01 afinet  pic 9(8) Binary Value 2.
01 soctype-stream  pic 9(8) Binary Value 1.
01 proto  pic 9(8) Binary Value zero.
01 socket-descriptor  pic 9(4) Binary Value zero.

*---------------------------------------------------------------*
* Buffer and length fields for read operation                  *
*---------------------------------------------------------------*
01 recv-flag  pic 9(8) Binary value zero.
01 read-request-len  pic 9(8) Binary Value zero.
01 read-request-read  pic 9(8) Binary Value zero.
01 read-request-remaining  pic 9(8) Binary Value zero.
01 read-buffer.
  05 read-buffer-total  pic x(8192) Value space.
  05 read-buffer-byte redefines read-buffer-total  pic x occurs 8192 times.

*---------------------------------------------------------------*
* Buffer and length fields for write operation                 *
*---------------------------------------------------------------*
01 send-request-len  pic 9(8) Binary value zero.
01 send-request-sent  pic 9(8) Binary value zero.
01 send-request-remaining  pic 9(8) Binary value zero.
01 send-buffer.
  05 send-buffer-total  pic x(8192) value space.
  05 send-buffer-byte redefines send-buffer-total  pic x occurs 8192 times.

*---------------------------------------------------------------*
* Error message for socket interface errors                    *
*---------------------------------------------------------------*
01 ezaerror-msg.
  05 filler  pic x(9) Value 'Function='.
**Procedure Division.**

*--------------------------------------------------------------------------*
* Receive TIM from the CICS Listener                                       *
*--------------------------------------------------------------------------*

A Beginner's Guide to MVS TCP/IP Socket Programming 220
move 72 to cleng.

exec cics retrieve
    into(CICS-listener-TIM)
    length(cleng)
end-exec.

move give-take-sd to startup-old-socket.
move lstn-asname to startup-lstn-asname.
move lstn-subtask to startup-lstn-subtask.
move sin-family to startup-sin-family.
move sin-port to startup-sin-port.
call 'TPIINTOA' using sin-addr startup-sin-addr.

exec cics writeq td
    queue('CSMT')
    from(cics-startup-msg-area)
    length(cics-startup-msg-len)
    nohandle
end-exec.

*---------------------------------------------------------------*
* Initialize socket API                                        *
*---------------------------------------------------------------*

move space to asname.
move eibtaskn to init-cics-task.
Call 'EZASOKET' using soket-initapi
    maxsoc
    initapi-ident
    subtask
    maxsno
    errno
    retcode.
if retcode < 0 then
    move 'Initapi failed' to ezaerror-text
    perform write-ezaerror-msg thru
    write-ezaerror-msg-exit.

*---------------------------------------------------------------*
* Let us see the client-id                                      *
*---------------------------------------------------------------*

move soket-getclientid to ezaerror-function.
Call 'EZASOKET' using soket-getclientid
    clientid
    errno
    retcode.
If retcode < 0 then
    move 'Getclientid failed' to ezaerror-text
    perform write-ezaerror-msg thru
    write-ezaerror-msg-exit
go to exit-term-api.
move clientid-name to clientid-msg-asname.
move clientid-task to clientid-msg-subtask.
exec cics writeq td
    queue('CSMT')
    from(cics-clientid-msg-area)
    length(cics-clientid-msg-len)
    nohandle
end-exec.
* Take the socket from the CICS Listener *

move lstn-asname to cid-name-lstn.
move lstn-subtask to cid-subtask-lstn.
move sin-family to cid-domain-lstn.
move low-value to cid-res-lstn.
move give-take-sd to socket-to-take.
move soket-takesocket to ezaerror-function.
Call 'EZASOKET' using soket-takesocket
    socket-to-take
    clientid-lstn
    errno
    retcode.
If retcode < 0 then
    move 'Takesocket failed' to ezaerror-text
    perform write-ezaerror-msg thru
    write-ezaerror-msg-exit
    go to exit-term-api.
move retcode to socket-descriptor.

* Start read/write loop *

move zero to client-status.
move zero to timer-accum.

Perform until (client-has-closed or
    timer-accum > 30)

* First we turn the socket into non-blocking mode *

Move soket-ioctl to ezaerror-function
Call 'TPIIOCTL' using ioctl-command-string
    ioctl-command-fionbio
If return-code > zero then
    move 'Call to TPIIOCTL failed' to ezaerror-text
    perform write-ezaerror-msg thru
    write-ezaerror-msg-exit
    go to exit-close-socket
end-if
Call 'EZASOKET' using soket-ioctl
    socket-descriptor
    ioctl-command-fionbio
    ioctl-reqarg-non-blocking
    ioctl-retarg
    errno
    retcode
If retcode < 0 then
    move 'IOCCTL call failed' to ezaerror-text
    perform write-ezaerror-msg thru
    write-ezaerror-msg-exit
    go to exit-close-socket
end-if

* Then we issue a read for 8192 bytes *
* If we receive any, we echo back what we got *
Move 8192 to read-request-len
Move zero to recv-flag
Perform read-TCP thru read-TCP-exit
If retcode = zero then
   Go to exit-close-socket
end-if
if read-request-read > 0 then
   move read-request-read to send-request-len
   move read-buffer to send-buffer
   Perform send-TCP thru send-TCP-exit
   move zero to timer-accum
else
   if errno = 35 then
      add 2 to timer-accum
      exec cics delay
      for seconds(2)
      end-exec
   else
      move 1 to client-status
   end-if
end-if
end-perform.

If timer-accum > 30 then
   move '30 second timeout' to ezaerror-text
   move 'Timeout' to ezaerror-function
   perform write-ezaerror-msg thru
   write-ezaerror-msg-exit
else
   move 'Client closed socket' to ezaerror-text
   move 'Client-close' to ezaerror-function
   perform write-ezaerror-msg thru
   write-ezaerror-msg-exit
end-if.

exit-close-socket.
   move soket-close to ezaerror-function.
   Call 'EZASOKET' using soket-close
       socket-descriptor
       errno
       retcode.
   If retcode < 0 then
      move 'Close call failed' to ezaerror-text
      perform write-ezaerror-msg thru
      write-ezaerror-msg-exit.

exit-term-api.
   Call 'EZASOKET' using soket-termapi.

*---------------------------------------------------------------*
* Terminate program *
*---------------------------------------------------------------*

exit-now.
  exec cics return
  end-exec.
  Goback.

*---------------------------------------------------------------*
* Write out an error message to CSMT *
*---------------------------------------------------------------*

write-ezaerror-msg.
  move errno to ezaerror-errno.
  move retcode to ezaerror-retcode.
  exec cics writeq td
    queue('CSMT')
      from(ezaerror-msg)
      length(ezaerror-msg-len) nohandle
    end-exec.
  write-ezaerror-msg-exit.
  exit.

*---------------------------------------------------------------*
* Subroutine: *
* ----------- *
* *
* Read data from a TCP connection *
*---------------------------------------------------------------*

Read-TCP.
  move soket-recv to ezaerror-function.
  move zero to read-request-read.
  move read-request-len to read-request-remaining.
  Perform until read-request-remaining = 0
    Call 'EZASOKET' using soket-recv
      socket-descriptor
      recv-flag
      read-request-remaining
      read-buffer-byte(read-request-read + 1)
      errno
      retcode
    If retcode < 0 and errno not = 35 then
      move 'Read call failed' to ezaerror-text
      perform write-ezaerror-msg thru
        write-ezaerror-msg-exit
      go to exit-close-socket
    end-if
    If retcode > 0 then
      Add retcode to read-request-read
      Subtract retcode from read-request-remaining
    end-if
    If retcode = 0 or errno = 35 then
      Move zero to read-request-remaining
    end-if
  end-perform.
Read-TCP-exit.
  exit.

*---------------------------------------------------------------*
* Subroutine: *
Send-TCP.
move socket-write to ezaerror-function.
move send-request-len to send-request-remaining.
move 0 to send-request-sent.
Perform until send-request-remaining = 0
   Call 'EZASOCKET' using socket-write
   send-descriptor
   send-request-remaining
   send-buffer-byte(send-request-sent + 1)
   errno
   retcode
   If retcode < 0 then
      move 'Write call failed' to ezaerror-text
      perform write-ezaerror-msg thru
         write-ezaerror-msg-exit
      go to exit-close-socket
   end-if
   add retcode to send-request-sent
   subtract retcode from send-request-remaining
   If retcode = 0 then
      Move zero to send-request-remaining
   end-if
end-perform.
Send-TCP-exit.
exit.

D.2 C Version of EZACICSC

/* This is a C version of EZACICSC */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>
#include <inet.h>
#include <socket.h>
#include <tcperrno.h>
#include <errno.h> /* required to make "errno" variable available */
#include <netdb.h> /* should not precede #include <manifest.h> on MVS */
#define recv read /* DOCUMENTATION ERROR! */
#define ebcdic2ascii(buffer,length)
ezacic04(buffer,(long*)(0x80000000|(long)&length));
#define ascii2ebcdic(buffer,length)
ezacic05(buffer,(long*)(0x80000000|(long)&length));

long retcode ;
/* DIS.H ALTERNATIVE */
char disMsgBuffer[300];
ifndef __stdio_h
#include <stdio.h>
#endif
#include <stdio.h>
A Beginner's Guide to MVS TCP/IP Socket Programming

A Beginner's Guide to MVS TCP/IP Socket Programming 226
/* program's variables */
/*===================================================================*/

struct clientid clientidLstn;

struct TcpSocketParm
{
    unsigned long giveTakeSocket;
    unsigned char lstnName[8];
    unsigned char lstnSubtaskname[8];
    unsigned char lstnText[6];
    unsigned char filler[29];
    unsigned char alignchar;
    struct sockaddr_in socketAddress;
} *pTcpSocketParm;

receiveBuffer = (char*)malloc(receiveBufferSize);

/* exec CICS handle condition not supported by CICS C support */
/* invreq (invreqErrSec) */
/* ioerr (ioerrSec) */
/* enddata (enddataSec) */
/* lengerr (lengerrSec) */
/* nospace (nospaceErrSec) */
/* qiderr (qiderrSec) */
/* itemerr (itemerrSec) */

writeCics("TRBB transaction start up");

exec CICS address eib(dfheiptr); /* Not automatic for C CICS */
checkEib("ADDRESS");

exec CICS retrieve set(pTcpSocketParm) length(length);
checkEib("RETRIEVE");
/* disshort(length); /* is set by the call */
/* disint(pTcpSocketParm);*/

disstr8(pTcpSocketParm->lstnName);
disstr8(pTcpSocketParm->lstnSubtaskname);
disint(pTcpSocketParm->giveTakeSocket);
disint(pTcpSocketParm->socketAddress.sin_family);
disint(pTcpSocketParm->socketAddress.sin_port);
dishex(pTcpSocketParm->socketAddress.sin_addr);

/*===================================================================*/
/* issue "takesocket" to acquire a socket */
/* which was given by listen program. */
/*===================================================================*/

memset((void*)&clientidLstn,0,sizeof(clientidLstn));

clientidLstn.domain = AF_INET;
memcpy(clientidLstn.name, pTcpSocketParm->lstnName, 8);
memcpy(clientidLstn.subtaskname, pTcpSocketParm->lstnSubtaskname, 8);

retcode = takesocket(&clientidLstn, pTcpSocketParm->giveTakeSocket);

if (retcode < 0) {
    writeCics("takesocket fail");
}
pgmExit();
}
else {
    writeCics("takesocket successful");
} /* endif */

sockid = retcode;

sendMessage = "Task starting thru CICS/TCPIP interface";
messageLength = strlen(sendMessage);
ascii = memcmp(pTcpSocketParm->lstnText,"EBCDIC",6);
disstr(pTcpSocketParm->lstnText);
disint(ascii);
if ( ascii ) ebcdic2ascii( sendMessage ,messageLength);

retcode = write (sockid , sendMessage , messageLength ) ;
if (retcode < 0) { writeCics("write socket fail"); pgmExit(); }

sentBytes = retcode ; disint(sentBytes);

recvBuffer = malloc(receiveBufferSize) ; disstr(recvBuffer);

receivedBytes = recv(sockid,receiveBuffer,receiveBufferSize,recvFlag);
disint(receivedBytes);
retcode = receivedBytes ;
*(receiveBuffer+receivedBytes) = 0 ; /* disstr requirement */
say(Before translation);
disstr(receiveBuffer);
if (retcode < 0) { writeCics("read socket fail"); pgmExit(); }

if ( ascii ) ascii2ebcdic(receiveBuffer,receivedBytes);
say(After translation);
disstr(receiveBuffer);

endTestField [3] = 0 ;
memcpy(endTestField,receiveBuffer,3);
if (!strcmp(endTestField,"end") ) { /*
    say(end indication received);
    taskFlag = 0 ; /* false */
    sendMessage = "connection end" ;
    messageLength = strlen(sendMessage) ;
    / *if ( ascii ) ebcdic2ascii(sendMessage,messageLength) ;*
    retcode = write (sockid ,sendMessage, messageLength ) ;
    if (retcode < 0) {
        writeCics("write socket fail pgm end msg");
        pgmExit();
    } /* endif */
    */ endif */
    */ @ echo message here with "data received" text? */
#endif NOT
    sendMessage = "data received ";
    messageLength = strlen(sendMessage) ;
    if ( ascii ) ebcdic2ascii(sendMessage,messageLength);
    retcode = write (sockid ,sendMessage, messageLength ) ;
} /* endif */
A Beginner's Guide to MVS TCP/IP Socket Programming

E.0 Appendix E. Sample REXX Socket Programs

This appendix contains the following two sets of sample REXX socket programs:

1. A sample iterative server and associated client. The programs use stream sockets, and they are written in REXX.

2. A sample implementation of a NETSTAT command in NetView. The NETSTAT REXX program is invoked from the NetView operator screen, it connects to the NETSTATS REXX server that runs in a batch TSO job. The NETSTAT parameters are passed to the NETSTATS REXX that issues the actual NETSTAT command with a STACK option, collects the output lines, and transfers them back to the NETSTAT REXX client in the NetView address space.

E.1 REXX Client
E.2 REXX Server
E.3 NetView NETSTAT Client REXX
E.4 NETSTAT Server REXX

E.1 REXX Client

/* REXX - Simple MVS TCP/IP Client */
if ,arg(1,'E') then do
  say 'Please specify hostname port bytesToSend'
  exit
end

parse arg hostname port bytesToSend .

service = 'T18ATCP' /* MUST be TCP/IP jobname */
socketsetsize = 10 /* number of preallocated sockets */
subtaskid = 'RBB' /* any name */

"alloc f(systcpd) da('sys1.tcpparms(tcpdata)') shr"
/* Prevent message EZY1372W Dataset *.TCPIP.DATA not found */

parse value check('socketsetlist',socket('socketsetlist')) with ids
do while ids,=''
    parse var ids id ids
    parse value socket('terminate',id ) with rc .
    if rc,=0 then say 'No cleanup was needed for id=' id
end

call check 'initialize',socket('initialize',subtaskid,socketsetsize,service )

/*af_inet = 2*/ /* not required */
parsed value check('socket',socket('socket',af_inet,'SOCK_STREAM','TCP')) with socket .
say 'socket id is' socket

parse value check('gethostbyname',socket('gethostbyname',hostname)) with ipaddress otheraddresses
say 'ipaddress="'ipaddress'"'
if otheraddresses,='' then say 'otheraddresses="'otheraddresses'"'

call check 'connect',socket('connect',socket,af_inet port ipaddress)

/* send a message of just 'A' characters */
parsed value check('send',socket('send',socket,copies('A',bytesToSend))) with bytesSent .
left = bytesToSend-bytesSent
if left>0 then say left 'bytes left.'

call check 'close socket', socket('close',socket)
call check 'terminate' , socket('terminate',subtaskid ) with rc .
exit

check: procedure
parse arg callname,returnstring
parse var returnstring rc rest
if rc=0 then do;say callname 'call successfull.'; return rest;end
else do;say callname 'call failed, rc='rc', reason='rest;exit;end

E.2 REXX Server

/* REXX - Simple MVS TCP/IP Server */
if ,arg(1,'E') then do
    say 'Please specify hostname port bytesExpected .
    exit
end
parse arg hostname port bytesExpected .

service = 'T18ATCP' /* MUST be TCP/IP jobname */
socketsetsize = 10 /* number of preallocated sockets */
subtaskid = 'RBB' /* any name */

"alloc f(systcpd) da('sys1.tcpparms(tcpdata)') shr"
/* Prevent message EZY1372W Dataset *.TCPIP.DATA not found */

parse value check('socketsetlist',socket('socketsetlist')) with ids
do while ids,=''
    parse var ids id ids
    parse value socket('terminate',id ) with rc .
    if rc,=0 then say 'No cleanup was needed for id=' id
end

A Beginner's Guide to MVS TCP/IP Socket Programming 230
call check 'initialize', socket('initialize', subtaskid, socketsetsize, service )

af_inet = 2
parse value check('socket', socket('socket', af_inet, 'SOCK_STREAM', 'TCP')) with socket .
say 'socket is' socket

ipaddress = 0 /* equivalent of INADDR_ANY */
call check 'bind', socket('bind', socket, af_inet port ipaddress)

backlog = 3 ; /* or any other value you'd like */
call check 'listen', socket('listen', socket, backlog)

say 'Waiting for connection request from client.'
parse value check('socket', socket('accept', socket)) with newsocket clientdomain clientport clientaddress .
say 'newsocket = '''newsocket'''
say 'clientdomain = '''clientdomain'''
say 'clientaddress = '''clientaddress'''
say 'clientport = '''clientport'''

parse value check('gethostbyaddr', socket('gethostbyaddr', clientaddress)) with clientname
say 'clientname = '''clientname'''

say 'Waiting for message to be received.'
message = '
receivedsofar = 0
do while receivedsofar<bytesExpected
    parse value check('read', socket('read', newsocket)) with length data
    if length ,=length(data) then say 'length discrepancy.'
    message = message||data
    receivedsofar = receivedsofar + length
end

E.3 NetView NETSTAT Client REXX

A Beginner's Guide to MVS TCP/IP Socket Programming

A Beginner's Guide to MVS TCP/IP Socket Programming 231
/* Interface: Same as NETSTAT command - except STACK and REPORT */
/* */
/* Logic: This REXX is used as a frontend rexx to */
/* the TCP/IP NETSTAT command from NetView. */
/* */
/* 1. Connects to netstat server at TCP port 6000 */
/* 2. Sends netstat parameters to server */
/* 3. Receives response from netstat server and */
/* displays result lines */
/* */
/* Returncode: RC = 0, processing OK */
/* Everything else is non-successful returncode from */
/* socket interface. */
/* */
/* Written: April 25, 1995 at ITSO Raleigh */
/* */
/* Modified: */
/* */
/*------------------------------------------------------------------*/
dotrace = 0 /*Controls tracing */
/*dotrace = 1 for trace */
netport = '6000' /*Server port number */
netserver = 'mvs18' /*Server host name */
subtaskid = opid() /*Subtask id = operator */
if dotrace then say 'Subtaskid = 'subtaskid
parse arg p0 p1 p2 p3 p4 p5 p6 p7 p8 p9
if dotrace then say 'Arguments passed = ' p0 p1 p2 p3 p4 p5 p6 p7 p8 p9
/*------------------------------------------------------------------*/
/* */
/* All socket calls are performed by subroutine DoSocket */
/* */
/*------------------------------------------------------------------*/
sockval = DoSocket('Terminate') /*Ensure clean interface*/
if dotrace then say 'Terminate returned: 'sockval
/*------------------------------------------------------------------*/
/* */
/* Initialize REXX socket interface */
/* */
/*------------------------------------------------------------------*/
sockval = DoSocket('Initialize', subtaskid)
if dotrace then say 'Initialize returned: 'sockval
if sockrc <> 0 then do
    say 'Socket initialize failed, rc='sockrc
    say sockval
    exit(sockrc)
end
/*------------------------------------------------------------------*/
/* */
/* Get IP address(es) of server host */
/* */
/*------------------------------------------------------------------*/
servipaddr = DoSocket('Gethostbyname', netserver)
if dotrace then say 'Gethostbyname returned: 'servipaddr
if sockrc <> 0 then do
    say 'Gethostbyname failed, rc='sockrc
    say sockval
    x=Doclean
    exit(sockrc)
end
parse value servipaddr with s1 s2 s3 s4 s5 s6 s7 s8 s9
y=0
do i = 1 to 9
    mystring = 'sipaddr.i = s'||i
interpret mystring
if sipaddr.i <> '' then y=y+1
if dotrace then say 'sipaddr.'i' = 'sipaddr.i
end
sipaddr.0 = y
if dotrace then say 'Number of IP addresses = 'sipaddr.0
/*----------------------------------------------------------------*/
/* */
/* Get a socket and try to connect to the server */
/* */
/* If connect fails (ETIMEDOUT), we must close the socket, */
/* get a new one and try to connect to the next IP address */
/* in the list, we received on the gethostbyname call. */
/* */
/* *----------------------------------------------------------------*/
i = 1
connected = 0
do until (i > sipaddr.0 | connected)
sockdescr = DoSocket('Socket')
if sockrc <> 0 then do
    say 'Socket failed, rc='sockrc
    x=Doclean
    exit(sockrc)
end
name = 'AF_INET '||netport||' '||sipaddr.i
sockval = DoSocket('Connect', sockdescr, name)
if sockrc = 0 then do
    connected = 1
end
else do
    sockval = DoSocket('Close', sockdescr)
    if sockrc <> 0 then do
        say 'Close failed, rc='sockrc
        x=Doclean
        exit(sockrc)
    end
end
i = i + 1
end
if .not.connected then do
    say 'Connect failed, rc='sockrc
    say sockval
    x=Doclean
    exit(sockrc)
end
netstatcmd = p0 p1 p2 p3 p4 p5 p6 p7 p8 p9
if dotrace then say 'Command='netstatcmd
/*----------------------------------------------------------------*/
/* */
/* Send the NETSTAT command to the NETSTAT server */
/* */
/* *----------------------------------------------------------------*/
sockval = DoSocket('Write', sockdescr, netstatcmd)
if dotrace then say 'Write returned: 'sockval
if sockrc <> 0 then do
    say 'Write failed, rc='sockrc
    x=Doclean
    exit(sockrc)
end
/*----------------------------------------------------------------*/
/* */
/* Read the response from the NETSTAT server */
/* */
/* Display output lines from the netstat command */
/* */
/*----------------------------------------------------------------*/
readlen = 1
resplen = 0
respdata = ''
parse upper value p0 with closedown
do until readlen = 0
    readdata = DoSocket('Read', sockdescr)
    if sockrc <> 0 then do
        say 'Read failed, rc='sockrc
        x=Doclean
        exit(sockrc)
    end
    if dotrace then say 'Server returned ' readdata
    parse value readdata with readlen readrest
    if readlen > 0 then do
        respdata = respdata||readrest
        resplen = resplen + readlen
    end
    if closedown = 'CLOSE' then readlen = 0
end
If dotrace then do
    say 'Total length = 'resplen
    say 'Total data = 'respdata
end
If resplen > 0 then do until resplen ,> 0
    eol = pos('00'X, respdata)
    len = eol - 1
    line = substr(respdata, 1, len)
    resplen = (resplen - eol)
    respdata = substr(respdata, eol+1, resplen)
    say line
end
/*----------------------------------------------------------------*/
/* */
/* Terminate socket interface */
/* */
/*----------------------------------------------------------------*/
sockval = DoSocket('Terminate')
if dotrace then say 'Terminate returned; 'sockval
if sockrc <> 0 then do
    say 'Socket Close failed, rc='sockrc
    say sockval
    exit(sockrc)
end
Exit(0)
/*----------------------------------------------------------------*/
/* */
/* Doclean Procedure. */
/* */
/*----------------------------------------------------------------*/
Doclean:
    sockval = DoSocket('Close', sockdescr)
    sockval = DoSocket('Terminate')
return sockres
/*---------------------------------------------------------------- */
/* */
DoSocket:
numargs = ARG() /*Number of passed args*/
argstring = '' /*Init arg string*/
if dotrace then do /*Tracepoint*/
say 'DoSocket subroutine' /*Trace entry to routine*/
say ' - Number of args = 'numargs /*Trace number of args*/
end /**/
do subix=1 to numargs /*Build argument string*/
if dotrace then do /*Tracepoint*/
say ' - arg('subix') = 'arg(subix) /*Trace each argument*/
end /**/
argstring = argstring||'arg('subix')' /*for the socket call*/
if subix<numargs then do /*If not last argument*/
argstring = argstring||',' /*add a comma*/
end /**/
interpret 'Parse value Socket('||argstring||') with sockrc sockres'
if dotrace then do /*Tracepoint*/
say ' - return code = 'sockrc /*Trace returncode*/
say ' - return string = 'sockres /*Trace return string*/
end /**/
return sockres /*Return socket result*/

E.4 NETSTAT Server REXX

/* REXX */
/*------------------------------------------------------------------*/
/* */
/* Name: NETSTATS - NETSTAT server */
/* */
/* Function: REXX Socket NETSTAT server. This is an iterative */
/* socket server, that serves netstat command requests */
/* from clients. Clients send the netstat parameters, */
/* this server does the actual netstat command, picks */
/* up the netstat output and returns it to the client. */
/* */
/* Client example is NETSTAT REXX in NetView. */
/* */
/* Interface: - none - */
/* */
/* Logic: This server binds to TCP port 6000. */
/* Processing is done in a never ending loop: */
/* 1. Accept connection request */
/* 2. Receive netstat parameters from client */
/* 3. Invoke netstat command with stack option */
/* 4. Pull out stacked netstat output lines */
/* 5. Send lines back to client - each line terminated */
/* by a X'00' byte */
/* 6. Go and wait for anew connection request */
/* */
/* Returncode: RC = 0, processing OK */
/* Everything else is non-successful returncode from */
/* socket interface. */
/* */
/* Written: April 25, 1995 at ITSO Raleigh */
/* */
/* Modified: */
/* */
/* */
/*------------------------------------------------------------------*/
dotrace = 0 /*Controls tracing*/
/*dotrace = 1 for trace*/
servport = '6000' /*Server port number*/
subtaskid = 'netstats' /*Subtask id*/
/*------------------------------------------------------------------*/
/* */
/* All socket calls are performed by subroutine DoSocket */
/* */
/* */
/*------------------------------------------------------------------*/
sockval = DoSocket('Terminate') /*Ensure clean interface*/
/*------------------------------------------------------------------*/
/* */
/* Initialize REXX socket interface */
/* */
/* */
/*------------------------------------------------------------------*/
sockval = DoSocket('Initialize', subtaskid)
if sockrc <> 0 then do
  say 'Initialize failed, rc='sockrc
  exit(sockrc)
end
/*------------------------------------------------------------------*/
/* */
/* Obtain a socket, bind it to our server port on INADDR_ANY and */
/* issue a listen call. */
/* */
/* */
/*------------------------------------------------------------------*/
sockdescr = DoSocket('Socket')
if sockrc <> 0 then do
  say 'Socket failed, rc='sockrc
  x=Doclean
  exit(sockrc)
end
sockval = DoSocket('Bind', sockdescr, 'AF_INET' servport 0)
if sockrc <> 0 then do
  say 'Bind failed, rc='sockrc
  x=Doclean
  exit(sockrc)
end
sockval = DoSocket('Listen', sockdescr)
if sockrc <> 0 then do
  say 'Listen failed, rc='sockrc
  x=Doclean
  exit(sockrc)
end
/*------------------------------------------------------------------*/
/* */
/* Enter iterative server loop, waiting for a connection request */
/* */
/* */
/*------------------------------------------------------------------*/
Do forever
  sockval = DoSocket('Accept', sockdescr)
  if sockrc <> 0 then do
    say 'Accept failed, rc='sockrc
    x=Doclean
    exit(sockrc)
  end
  parse value sockval with newsock.
/*------------------------------------------------------------------*/
/* Read netstat parameters from client */
/* If client sends a CLOSE command, we will terminate the */
/* iterative server loop. */
/* We will add a stack parameter to the passed parameters. */
/* Ensure that client did not send a stack or a report option. */
/* */
/*------------------------------------------------------------------*/
sockval = DoSocket('Read', newsock)
if sockrc <> 0 then do
  say 'Read failed, rc='sockrc
  x=Doclean2
  exit(sockrc)
end
parse upper value sockval with resplen netcmdi
If substr(netcmdi,1,5) = 'CLOSE' then do
  say 'Server is terminating as result of a CLOSE command'
  retstring = 'Server Closing Down'||'00'X
  sockval = DoSocket('Write', newsock, retstring)
  sockval = DoSocket('Shutdown', newsock, read)
  sockval = DoSocket('Close', sockdescr)
  sockval = DoSocket('Terminate')
  exit(0)
end
if dotrace then say 'Received data = 'netcmdi
antparms = words(netcmdi)
netcmd = ''
if antparms > 0 then do i=1 to antparms
  subparm = word(netcmdi,i)
  select
    when substr(subparm,1,4) = 'STAC' then nop
    when substr(subparm,1,3) = 'REP' then nop
    Otherwise netcmd = netcmd||' '||word(netcmdi,i)
  end
end
netcmd = 'STACK '||netcmd
if dotrace then say 'Command = NETSTAT 'netcmd
/*------------------------------------------------------------------*/
/* */
/* Do the actual NETSTAT command with the passed parameters */
/* Pull out the output lines from the stack, add a line */
/* terminating character to each line including the last and */
/* send the full return buffer back to the client. */
/* */
/*------------------------------------------------------------------*/
msgstat=MSG()
z=MSG("OFF")
address tso "NETSTAT" netcmd
xy = queued()
If dotrace then say 'Number of lines returned='xy
retstring = ''
doi=1 to xy
  index = xy-i+1
  pull lin.index
end
doi = 1 to xy
  retstring=retstring||lin.i||'00'X
end
z=MSG(msgstat)
if retstring = '' then do
  retstring = 'No response from NETSTAT command'||'00'X
end
sockval = DoSocket('Write', newsock, retstring)
if dotrace then say 'Write returned: 'sockval
if sockrc <> 0 then do
   say 'Write failed, rc='sockrc
   x=Doclean2
   exit(sockrc)
end

/*------------------------------------------------------------------*/
/* */
/* Close the socket and wait for a new connection */
/* */
/* *------------------------------------------------------------------*/
sockval = DoSocket('Close', newsock)
if sockrc <> 0 then do
   say 'Socket Close failed, rc='sockrc
   x=Doclean
   exit(sockrc)
end
end

/*------------------------------------------------------------------*/
/* */
/* Doclean Procedure. */
/* */
/* */
/* If a socket call failed, and we are about to exit this */
/* Rexx application, close the socket and terminate the */
/* socket interface. */
/* */
/* *------------------------------------------------------------------*/
Doclean:
   if dotrace then do
      say 'Cleaning up socket descriptor = 'sockdescr
   end
   sockval = DoSocket('Close', sockdescr)
   sockval = DoSocket('Terminate')
return sockres
Doclean2:
   if dotrace then do
      say 'Cleaning up socket descriptor = 'sockdescr
      say ' and socket descriptor = 'newsock
   end
   sockval = DoSocket('Close', sockdescr)
   sockval = DoSocket('Close', newsock)
   sockval = DoSocket('Terminate')
return sockres

/*------------------------------------------------------------------*/
/* */
/* DoSocket procedure. */
/* */
/* */
/* Do the actual socket call, and parse the return code. */
/* */
/* Return rest of string returned from socket call. */
/* */
/* */
/* *------------------------------------------------------------------*/
DoSocket:
   numargs = ARG()       /*Number of passed args */
   argstring = ''       /*Init arg string */
   if dotrace then do
      say 'DoSocket subroutine'    /*Trace entry to routine*/
      say ' - Number of args = 'numargs /*Trace number of args */
   end
   /* */
   do subix=1 to numargs
   /*Build argument string */
A Beginner's Guide to MVS TCP/IP Socket Programming 238
if dotrace then do /* Tracepoint */
  say ' - arg('subix') = 'arg(subix) /* Trace each argument */
end /* */
argstring = argstring||'arg('subix')' /* for the socket call */
if subix<numargs then do /* If not last argument */
  argstring = argstring||',' /* add a comma */
end /* */
end /* */
msgstat = msg() /* Save message status */
z = msg("OFF") /* Turn messages off */
interpret 'Parse value Socket('||argstring||') with sockrc sockres'
z = msg(msgstat) /* Restore message status */
if dotrace then do /* Tracepoint */
  say ' - return code = 'sockrc /* Trace returncode */
  say ' - return string = 'sockres /* Trace return string */
end /* */
return sockres /* Return socket result */

F.0 Appendix F. Sample PL/I Socket Programs

This appendix contains a sample iterative PL/I server and associated client. The programs use stream sockets, and they are written in PL/I.

F.1 PL/I Server
F.2 PL/I Server

/* PL/I Stream Socket Server */
pserver: proc(parm)options(main);

/* The extended sockets API routine */
dcl ezasoket entry options(retcode,asm,inter) ext;
dcl function char(16) ;

/* Regular C routines to convert Internet addresses */
dcl inet_addr entry(char(16) ) options(retcode,asm,inter) ext('INET@ADD');
dcl inet_ntoa entry(fixed(31)bin) options(retcode,asm,inter,byvalue) ext('INET@NTA');

/***************************************************************************/
/* */
/* Subroutines */
/* */
/***************************************************************************/

/* Routine to parse the parameterlist */
word:procedure(string,wordno)returns(char(255)var);
dcl string char(*)var;
dcl wordno fixed(31)bin;
dcl i fixed(31)bin;
dcl pl1 fixed(31)bin;
dcl pl2 fixed(31)bin init(0);
do i=1 to wordno;
  if p2>length(string) then return('');
  pl1=verify(substr(string,p2+1)," ");
  if pl1=0 then return('');
  pl1=pl1+p2;
  p2=index(substr(string,pl1),' ');
  if p2=0 then p2=length(string)+1;else p2=p2+pl1-1;
end;
return(substr(string,p1,p2-p1));
end word;

/* Routine to check parameter validity */
numeric:procedure(num_string)returns(bit(1));
dcl num_string char(100)var;
return(verify(num_string,'0123456789')=0);
end numeric;

/* Routines to convert strings */
z2var : procedure(zstring)returns(char(255)var);
dcl zstring char(256);
dcl p fixed(31)bin;
p=index(zstring,low(1));
if p=0 then return(zstring);
else return(substr(zstring,1,p-1));
end z2var;
var2z : procedure(varstring)returns(char(256));
dcl varstring char(255)var;
return(varstring||low(1));
end var2z;

*********************************************************************/
/* */
/* Subroutine to check results of all socket API calls */
/* */
*********************************************************************/
sock_check:procedure(function,errno,retcode) returns(bit(1));
dcl function char(16) ; /* function name */
dcl retcode fixed bin(31) ; /* return code */
dcl errno fixed bin(31) ; /* error number */
put skip edit(function) (a);
if retcode >=0 then do;
put edit(' completed OK.')(A) ;
return('0'B);
end; else do;
put edit(' failed, errno=',errno) (a,f(9)) ;
return('1'B);
end;
end sock_check;

/* Routine to send records */
sendRecord : procedure (socket , recordBuffer , recordLength )
returns(bit(1));

/* parameter declarations */
dcl socket fixed(15)bin ;
dcl recordBuffer char(*) ;
dcl recordLength fixed(31)bin ;

/* internal variable declarations */
dcl bytesSent fixed(31)bin init(0) ;
dcl bytesToBeSent fixed(31)bin ;
dcl remainingBytes fixed(31)bin init(recordLength);
function = 'WRITE' ;
sendloop : do while ( remainingBytes >0);
bytesToBeSent = remainingBytes ;
call ezasoket( function ,

A Beginner's Guide to MVS TCP/IP Socket Programming
socket, bytesToBeSent, substr(recordBuffer,bytesSent+1), errno, retcode);
if sock_check(function,errno,retcode) then stop;
bytesSent = retcode;
if bytesSent=0 then do;
  put skip list('Connection broken while sending.');
  return ('1'b);
end;
put skip edit(bytesSent,' bytes have been sent.')($9,a);
remainingBytes = remainingBytes - bytesSent;
end sendloop;
put skip list('Complete record sent.');
return ('0'b);
end sendRecord;

/* Routine to receive records */
receiveRecord : procedure (socket , recordBuffer , recordLength )
returns(fixed(31)bin);

/* parameter declarations */
dcl socket fixed(15)bin ;
dcl recordBuffer char(*) ;
dcl recordLength fixed(31)bin ;

/* internal variable declarations */
dcl bytesReceived fixed(31)bin init(0);
dcl bytesReceivedNow fixed(31)bin;
dcl bytesToBeReceived fixed(31)bin;
dcl remainingBytes fixed(31)bin init(recordLength) ;

begin;
dcl receiveBuffer char(recordLength) ;
function = 'READ' ;
receiveloop : do while ( remainingBytes >0);
  bytesToBeReceived = remainingBytes;
call ezasoket(
    function ,
    socket ,
    bytesToBeReceived ,
    receiveBuffer ,
    errno ,
    retcode);
if sock_check(function,errno,retcode) then stop;
bytesReceivedNow = retcode;
put skip edit(bytesReceivedNow,' bytes have been received.'($9,a);
if bytesReceivedNow=0 then do;
  put skip list('Connection broken while receiving.');
  return ('1'b);
end;
substr(recordBuffer,bytesReceived+1,bytesReceivedNow) = receiveBuffer;
remainingBytes = remainingBytes - bytesReceivedNow;
bytesReceived = bytesReceived + bytesReceivedNow;
end receiveloop;
put skip list('Complete record received.');
return ('0'b);
end;
end receiveRecord ;

******************************************************************************
/* */
/* Obtain information from JCL parameterlist (PARM=) */
/* */

/*************************************************************/
dcl parm char(100) var; /* passed in JCL */
dcl tcpipjobname char(8); /* T18ATCP in Raleigh, ZTCP/IP in La Hulpe */
dcl ownport fixed(15) bin init(0);
dcl recordlen fixed(31) bin init(0);
dcl cownport char(100) var;
dcl crecordlen char(100) var;

tcpipjobname = word(parm,1);
cownport = word(parm,2);
crecordlen = word(parm,3);

if tcpipjobname=' ' | numeric(cownport) | numeric(crecordlen) then do;
    put skip list('Usage: parm= /tcp/ip-jobname ownport recordlength''');
call pliretc(4);
    return;
end;

ownport = cownport;
recordlen = crecordlen;
put skip data(ownport);
put skip data(recordlen);

/*************************************************************/
/* */
/* INITAPI call defines TCP/IP subsystem in this MVS to be used */
/* NOTE: "tcpname" should be your TCP/IP's jobname */
/* */
/*************************************************************/
dcl maxsoc fixed bin(15) init(255); /* largest socket # checked */
dcl 1 id, /* */
2 tcpname char(8) , /* TCP/IP jobname */
2 adsname char(8) init(' '); /* local address space */
dcl subtask char(8) init('SUBTASK'); /* task/path identifier */
dcl maxsno fixed bin(31) init(0); /* max descriptor assigned */
dcl retcode fixed bin(31) init(0); /* return code */
dcl errno fixed bin(31) init(0); /* error number */

id.tcpname = tcpipjobname;

function = 'INITAPI';
call ezaosoket(function, maxsoc, id, subtask, maxsno, errno, retcode);
if sock_check(function,errno,retcode) then stop;

/*************************************************************/
/* */
/* SOCKET call obtains a "socket" descriptor, no comms yet. */
/* */
/*************************************************************/

function = 'SOCKET';
dcl af_inet fixed bin(31) init(2); /* internet domain */
dcl type_stream fixed bin(31) init(1); /* two-way byte stream */
dcl proto fixed bin(31) init(0); /* prototype default */
call ezaosoket(function, af_inet, type_stream, proto, errno, retcode);
if sock_check(function,errno,retcode) then stop;

socket fixed bin(15); /* socket descriptor */
socket = retcode;
function = 'BIND' ;
dcl 1 local_address, /* our socket address */
 2 family fixed bin(15) init(2), /* AF_INET = TCP/IP */
 2 port fixed bin(15), /* our own port */
 2 address fixed bin(31) init(0), /* accept any address */
 2 reserved char(8); /* reserved */
local_address.port = cownport; /* can not through init */
call ezasoket(function, socket, local_address, errno, retcode);
if sock_check(function,errno,retcode) then stop ;

function = 'LISTEN' ;
dcl backlog fixed bin(31) init(5); /* max length of pending queue */
call ezasoket(function, socket, backlog, errno, retcode);
if sock_check(function,errno,retcode) then stop ;

function = 'ACCEPT' ;
dcl 1 client_address like local_address ; /* our socket address */
call ezasoket(function, socket, client_address, errno, retcode);
if sock_check(function,errno,retcode) then stop ;
dcl newsocket fixed bin(15); /* new socket */
newsocket = retcode;

begin;
dcl record char(recordlen);
if receiveRecord(newsocket,record,recordlen) then put skip list('receive failed.);
if sendRecord(newsocket,record,recordlen) then put skip list('send failed.);
end;

/*******************************************************************/
/* */
/* Issue CLOSE calls for both sockets */
/* */
/*******************************************************************/

function = 'CLOSE' ;
call ezasoket(function, newsocket, errno, retcode);
if sock_check(function,errno,retcode) then stop ;
call ezasoket(function, socket, errno, retcode);
if sock_check(function,errno,retcode) then stop ;

/*******************************************************************/
/* */
/* TERMAPI call terminates the connection between this address space */
/* and the TCP/IP address space chosen by INITAPI */
/* */
/*******************************************************************/

function = 'TERMAPI' ;
call ezasoket(function);
call pliretc(0);
end pserver;

F.2 PL/I Server

/* PL/I Stream Socket Client */
pclient: proc(parm)options(main);
/* The extended sockets API routine */
dcl ezasoket entry options(retcode,asm,inter) ext;
dcl function char(16) ;
/* Regular C routines to convert Internet addresses */
dcl inet_addr entry(char(16) ) options(retcode,asm,inter) ext('INET@ADD');
dcl inet_ntoa entry(fixed(31)bin) options(retcode,asm,inter,byvalue) ext('INET@NTA');

/*******************************************************************/
/* */
/* Subroutines */
/* */
/*******************************************************************/
/* Routine to parse the parameterlist */
word:procedure(string,wordno)returns(char(255)var);
dcl string char(*)var;
dcl wordno fixed(31)bin;
dcl i fixed(31)bin;
dcl p1 fixed(31)bin;
dcl p2 fixed(31)bin init(0);
do i=1 to wordno;
if p2>length(string) then return('');
p1=verify(substr(string,p2+1),' ');
if p1=0 then return('');
p1=p1+p2;
p2=index(substr(string,p1),' ');
if p2=0 then p2=length(string)+1; else p2=p2+p1-1;
end;
return(substr(string,p1,p2-p1));
end word;

/* Routine to check parameter validity */
numeric:procedure(num_string)returns(bit(1));
dcl num_string char(100)var;
return(verify(num_string,'0123456789')=0);
end numeric;

/* Routines to convert strings */
z2var : procedure(zstring)returns(char(255)var);
dcl zstring char(256);
dcl p fixed(31)bin;
p=index(zstring,low(1));
if p=0 then return(zstring);
else return(substr(zstring,1,p-1));
end z2var;
var2z : procedure(varstring)returns(char(256));
dcl varstring char(255)var;
return(varstring||low(1));
end var2z;

******************************************************************************
/* */
/* Subroutine to check results of all socket API calls */
/* */
******************************************************************************

sock_check:procedure(function,errno,retcode) returns(bit(1));
dcl function char(16) ; /* function name */
dcl retcode fixed bin(31) ; /* return code */
dcl errno fixed bin(31) ; /* error number */
put skip edit(function) (a);
if retcode >=0 then do;
   put edit(' completed OK.')(A) ;
   return('0'B);
end; else do;
   put edit(' failed, errno=',errno) (a,f(9)) ;
   return('1'B);
end;
end sock_check;

/* Routine to send records */
sendRecord : procedure (socket , recordBuffer , recordLength )
returns(bit(1));

/* parameter declarations */
dcl socket fixed(15)bin ;
dcl recordBuffer char(*) ;
dcl recordLength fixed(31)bin ;

/* internal variable declarations */
dcl bytesSent fixed(31)bin init(0) ;
dcl bytesToBeSent fixed(31)bin ;
dcl remainingBytes fixed(31)bin init(recordLength);
function = 'WRITE' ;
sendloop : do while ( remainingBytes >0);
    bytesToBeSent = remainingBytes ;
call ezasoket(  
        function      ,  
        socket        ,  
        bytesToBeSent ,  
        substr(recordBuffer,bytesSent+1) ,  
        errno         ,  
        retcode);  
    if sock_check(function,errno,retcode) then stop ;  
    bytesSent = retcode ;
    if bytesSent=0 then do ;
        put skip list('Connection broken while sending.');  
        return ('1'b) ;
    end ;
    put skip edit(bytesSent,' bytes have been sent.')(f(9),a);  
    remainingBytes = remainingBytes - bytesSent ;
end sendloop ;
put skip list('Complete record sent.');  
return ('0'b);
end sendRecord ;

/* Routine to receive records */
receiveRecord : procedure (socket , recordBuffer , recordLength )
returns(fixed(31)bin);

/* parameter declarations */
dcl socket fixed(15)bin ;
dcl recordBuffer char(*) ;
dcl recordLength fixed(31)bin ;

/* internal variable declarations */
dcl bytesReceived fixed(31)bin init(0);  
dcl bytesReceivedNow fixed(31)bin;  
dcl bytesToBeReceived fixed(31)bin ;
dcl remainingBytes fixed(31)bin init(recordLength) ;

begin;
dcl receiveBuffer char(recordLength) ;
function = 'READ' ;  
receiveloop : do while ( remainingBytes >0);  
    bytesToBeReceived = remainingBytes ;
call ezasoket(  
        function      ,  
        socket        ,  
        bytesToBeReceived ,  
        receiveBuffer   ,  
        errno         ,  
        retcode);  
    if sock_check(function,errno,retcode) then stop ;  
    bytesReceivedNow = retcode ;
    put skip edit(bytesReceivedNow,' bytes have been received.')(f(9),a);  
    if bytesReceivedNow=0 then do ;
        put skip list('Connection broken while receiving.');  
        return ('1'b) ;
    end ;
    put skip edit(bytesReceived,' bytes have been received.')(f(9),a);  
    remainingBytes = remainingBytes - bytesReceived ;
    substr(recordBuffer,bytesReceived+1,bytesReceivedNow) = receiveBuffer;  
    remainingBytes = remainingBytes - bytesReceivedNow ;
    bytesReceived = bytesReceived + bytesReceivedNow ;
end receiveloop ;
put skip list('Complete record received.');
return ('0'b);
end;
end receiveRecord ;

/***************************************************************************/
/* */
/* Obtain information from JCL parameterlist (PARM=) */
/* */
/***************************************************************************/
dcl parm     char(100)var; /* passed in JCL */
dcl tcpipjobname char(8); /* T18ATCP in Raleigh, ZTCPIP in La Hulpe */
dcl servername char(100);
dcl serverport fixed(15)bin init(0);
dcl namelen    fixed(31)bin init(0);
dcl recordlen fixed(31)bin init(0);
dcl crecordlen char(100)var;
dcl cserverport char(100)var;
tcpipjobname  = word(parm,1);
servername    = word(parm,2); namelen=length(servername);
cserverport  = word(parm,3);
crecordlen   = word(parm,4);
if tcpipjobname=' '|servername=' '|,numeric(cserverport)|,numeric(crecordlen) then do;
  put skip list('Usage: parm='/tcp/ip-jobname servername serverport recordlength''');
call pliretc(4);
  return ;
end;
serverport = cserverport ;
recordlen = crecordlen ;
put skip data(servername);
put skip data(serverport);
put skip data(recordlen );

/***************************************************************************/
/* */
/* INITAPI call defines TCP/IP subsystem in this MVS to be used */
/* NOTE: "tcpname" should be your TCP/IP's jobname */
/* */
/***************************************************************************/
dcl maxsoc fixed bin(15) init(255); /* largest socket # checked */
dcl 1 id , /* */
2 tcpname char(8), /* TCP/IP jobname */
2 adname char(8) init(' '), /* local address space */
dcl subtask char(8) init('SUBTASK'); /* task/path identifier */
dcl maxsno fixed bin(31) init(0); /* max descriptor assigned */
dcl retcode fixed bin(31) init(0); /* return code */
dcl errno fixed bin(31) init(0); /* error number */
id.tcpname = tcpipjobname ;

function = 'INITAPI' ;
call ezasoket(function, maxsoc, id, subtask, maxsno, errno, retcode);
if sock_check(function,errno,retcode) then stop ;

/***************************************************************************/
/* */
/* SOCKET call obtains a "socket" descriptor, no comms yet. */
/* */
/***************************************************************************/
function = 'SOCKET';
dcl af_inet fixed bin(31) init(2); /* internet domain */
dcl type_stream fixed bin(31) init(1); /* two-way byte stream */
dcl proto fixed bin(31) init(0); /* prototype default */
call ezasoket(function, af_inet, type_stream, proto, errno, retcode);
if sock_check(function, errno, retcode) then stop;
dcl socket fixed bin(15); /* socket descriptor */
socket = retcode;

/*********************************************************************/
/* Prepare for CONNECT - common part */
/*********************************************************************/

dcl 1 name_id, /* server address in reqd. fmt. */
    2 family fixed bin(15) init(2), /* AF_INET: TCP/IP */
    2 port fixed bin(15), /* port ) of the server */
    2 address fixed bin(31), /* address ) to be contacted */
    2 reserved char(8); /* reserved */

name_id.port = serverport;

/*********************************************************************/
/* Find out how the server address is specified: either/or: */
/* 1. as a "dotted decimal" address */
/* 2. as a symbolic address */
/*********************************************************************/

dcl hostaddr fixed(31)bin;
call inet_addr(var2z(servername));
hostaddr = pliretv();
if hostaddr = -1 then do; /* this means the address was symbolic */

/*********************************************************************/
/* Find server address by means of GETHOSTBYNAME */
/*********************************************************************/

dcl 1 hostent based(hostentptr),
    2 nameptr ptr,
    2 aliaslist ptr,
    2 family fixed(31)bin,
    2 hostaddrlen fixed(31)bin,
    2 hostaddrlist ptr;
function = 'GETHOSTBYNAME';
call ezasoket(function, namelen, servername, hostentptr, retcode);
if sock_check(function, errno, retcode) then stop;

dcl hostname char(256) based(hostent.nameptr);
dcl alias char(256) based;
dcl aliasptr (99) ptr based(hostent.aliaslist);
dcl hostaddrptr (99) ptr based(hostent.hostaddrlist);
dcl hostaddrn fixed(31)bin based;
put skip edit('Full name of server host: ',z2var(hostname),'')(a,a,a);
function = 'CONNECT' ;
dcl addressIndex fixed(31)bin ;
dcl dotted_address char(16) based(dotted_address_pointer);
do addressIndex = 1 by 1 while (unspec(hostaddrptr(addressIndex)),=0) ;
   name_id.address = hostaddrptr(addressIndex)->hostaddrn ;
call inet_ntoa(name_id.address); /* call C inet_ntoa routine */
   unspec(dotted_address_pointer) = unspec(pliretv());
put skip edit('Trying to contact TCP/IP address "',z2var(dotted_address),'"') (a,a,a);
call ezasoket(function, socket, name_id, errno, retcode);
if ,sock_check(function,errno,retcode) then leave ;
end ;
if unspec(hostaddrptr(addressIndex))=0 then do;
   put skip list('Unable to contact any of the addresses of the specified server.'); stop;
end;
end;else do;

function = 'CONNECT' ;
name_id.address = hostaddr ;
call ezasoket(function, socket, name_id, errno, retcode);
if sock_check(function,errno,retcode) then stop ;
end;

function = 'GETSOCKNAME' ;
dcl 1 local_address like name_id ; /* to get our local address */
call ezasoket(function, socket, local_address, errno, retcode);
if ,sock_check(function,errno,retcode) then do;
   call inet_ntoa(local_address.address); /* call C inet_ntoa routine */
   unspec(dotted_address_pointer) = unspec(pliretv());
   put skip edit('Our local TCP/IP address "',z2var(dotted_address),'" - port',local_address.port) (a,a,a,f(5));
   if local_address.family,=2 then put skip list('Not AF_INET family.'); /* very unlikely */
end;

begin;
dcl echoRecord char(recordlen);
dcl recordSent char(recordlen);
recordSent=repeat('A',recordlen-1);

if sendRecord (socket,recordSent,recordlen) then put skip list('send failed.');
if receiveRecord(socket,echoRecord,recordlen) then put skip list('receive failed.');

if recordSent=echoRecord then put skip list('Echoed record identical.');</p>
else put skip list('Echoed record *not* identical.');
end;

/*****************************************************************************************************/
/* */
/* SHUTDOWN call terminates the connection with the server */
/* */
/*****************************************************************************************************/

function = 'CLOSE';
call ezasoket(function, socket, errno, retcode);
if sock_check(function,errno,retcode) then stop;

/*****************************************************************************************************/
/* */
/* TERMAPI call terminates the connection between this address space */
/* and the TCP/IP address space chosen by INITAPI */
/* */
/*****************************************************************************************************/

function = 'TERMAPI';
call ezasoket(function);
call pliretc(0);
end pclient;

G.0 Appendix G. Socket Utilities for Sockets Extended Programs

This appendix contains a number of handy utility programs that were developed during the creation of this book. We document them here because they may come in handy when you begin to develop your own Sockets Extended programs.

G.1 TPICLNID Obtain Values for TCP/IP Client ID
G.2 TPIINTOA Convert IP Address to Character String
G.3 TPIIADDR Convert IP Address Character String to Full-word
G.4 TPIIOCTL Convert IOCTL Command Name to Command
G.5 TPWAIT Place Calling Process in Wait
G.6 TPIRACF Interface to RACROUTE REQUEST=VERIFY User SVC
G.7 User SVC for RACROUTE REQUEST=VERIFY
G.8 TPIAUTH Issue RACROUTE REQUEST=AUTH for FACILITY Class

G.1 TPICLNID Obtain Values for TCP/IP Client ID

***********************************************************************************************
* *
* Name: TPICLNID *
* *
* Function: Return address space name and TCB address as two 8-character fields. *
* *
***********************************************************************************************
* Interface: R1 -> parameter list *
* 0 Pointer to address space name field (Out) *
* 4 Pointer to TCB address name field (Out) *
* *
* Logic: Find address space name via ASCB and TCB address via *
* CVT. Format TCB address into 8 bytes character field, *
* and return address space name and TCB address. *
* *
* Returncode: - none - *
* *
* Written: March 27'th 1994 at ITSO Raleigh *
* *
* Modified: *
* *
**********************************************************************
PRINT NOGEN
IHAASCB *ASCB layout
TPIWORK DSECT
DC 18F'0' *Save Area
DORD DC D'0' *Decimal work word *
*
ASNAMED DSECT
ASNAME DC CL8' ' *Address space name
TCBNAMED DSECT
TCBNAME DC CL8' ' *TCB address *
*
TPICLNID INIT 'Find Address space name and TCB address',RENT=YES, C
WORKLEN=256,MODE=31
*
USING TPIWORK,R13
L R9,0(R1) *-> Address space name return field
USING ASNAMED,R9
L R10,4(R1) *-> TCB address name return field
USING TCBNAMED,R10
L R3,'10' *-> CVT
L R3,0(R3) *-> TCB Words
L R15,12(R3) *-> Current ASCB (My ASCB)
USING ASCB,R15
L R3,4(R3) *-> Current TCB (My TCB)
SR R2,R2 *Make ready for double shift
SLDL R2,4 *00000000 xxxxxxxx0
STM R2,R3,DORD *Store for Unpack
UNPK TCBNAME,DORD *Unpack
NC TCBNAME,=8X'0F' *Remove F's
TR TCBNAME,TRHEX *Translate to EBCDIC
ICM R14,15,ASCBJNBI *-> Jobname if initiated
BNZ INITJBN *If not zero, pointer is OK
ICM R14,15,ASCBJNBS *-> Jobname if start/logon
BNZ INITJBN *If not zero, pointer is OK
MVC ASNAME,=CL8' ' *We did not find a jobname
B INITJOBS *Job name is initialized
INITJBN EQU *
MVC ASNAME,0(R14) *Move in job name
INITJOBS EQU *
TERM RC=0 *And out we go
LTORG
TRHEX DC C'0123456789ABCDEF' *Hex to char translation
END

G.2 TPIINTOA Convert IP Address to Character String
**Name:** TPIINTOA

**Function:** Convert an IP address from a fullword network byte order to 15 characters dotted decimal notation.

**Interface:** R1 -> parameter list with two pointers:
+0 -> fullword with IP address in network byte format
+4 -> 15 character return area, where IP address will be returned in dotted decimal format.

**Logic:** This module is called from whenever another module needs to convert an IP address to dotted decimal format. Output is returned in the format a human would type it in. No leading zeroes if a part of the address is less than 3 characters in length (a value above 99).

**Example:**
Input: X'09180221'
Output: CL15'9.24.2.33'

**Abends:** - none -

**Returncode:** - none -

**Written:** May 28'th 1994 at ITSO Raleigh

**Modified:**

**WORKAREA DSECT**

```
 DC 18F'0'           *Save area
DORD DC D'0'
WORK DC CL4' '
INITSTR DC CL15' '  *Init string
DOTSTR DC CL15' . . . '
WORKSTR DC CL15' '
```

**TPIINTOA INIT 'Build dotted string from fullword IP address', C**

```
RENT=YES,WORKLEN=512
```

**USING WORKAREA,R13**

```
L R9,0(R1)           *-> Fullword with IP address
L R10,4(R1)         *-> 15 character string
```

```
MVC INITSTR,=CL15' ' *Initial string of spaces
MVC DOTSTR,=CL15' . . . ' *Initial dotted string
MVC 0(L'INITSTR,R10),INITSTR *Initialize to space
MVC WORKSTR,DOTSTR   *Initialize workstring
```

```
LA R2,WORKSTR       *-> First part goes here
LR R3,R9           *-> First byte of fullword
```

A Beginner's Guide to MVS TCP/IP Socket Programming

252
LA R4,1  
LR R5,R3  
LA R5,3(R5)  

BLDSTR EQU *  
SR R1,R1  
IC R1,0(R3)  
CVD R1,DORD  
OI DORD+7,X'0F'  
MVC WORK,=XL4'40202120'  
ED WORK,DORD+6  
LA R2,4(R2)  
BXLE R3,R4,BLDSTR  

*-----------------------------------------------*  
*  Reduce the intermediate result, so leading zeroes are *  
*  removed:  *  
*  00x.0xx.xxx.0xx => x.xx.xxx.xx  *  
*-----------------------------------------------*  

LA R2,WORKSTR  
LA R3,L'WORKSTR  
LR R6,R2  
AR R6,R3  
BCTR R6,0  

SPCLOOK EQU *  
CLI 0(R2),C' '  
BNE SPCADV  
BCTR R3,0  
LTR R3,R3  
BZ SPCEND  
LR R4,R3  
BCTR R4,0  
EX R4,MVCSTR  
MVI 0(R6),C' '  
BCTR R6,0  
B SPCLOOK  

SPCADV EQU *  
LA R2,1(R2)  
BCTR R3,0  
CH R3,=AL2(0)  
BH SPCLOOK  

SPCEND EQU *  
MVC 0(L'WORKSTR,R10),WORKSTR  
TERM RC=0  
MVCSTR MVC 0(*-*,R2),1(R2)  
END  

G.3 TPIIADDR Convert IP Address Character String to Full-word  

******************************************************************************  
* Name: TPIIADDR  
* Function: Convert an IP address from a 15 character dotted decimal format to a fullword network byte format.  
* Interface: R1 -> parameter list with two pointers:  
* +0 -> 15 character area with IP address in dotted
decimal format.
+4    ->  fullword return area for IP address in network
byte format.

Logic: This module is called from other TPI modules
to convert an IP address from dotted decimal format to
a fullword network byte format.

Example:
Input:  CL15'9.24.2.33'
Output: X'09180221'

Abends:  - none -

Returncode:  RC = 00: Conversion OK
RC = 08: A part is longer than 3 characters
(9.1234.2.3)
RC = 12: A part has a zero length (9..2.3)
RC = 16: Non numeric data (9.A.B.2)
RC = 20: A part has a value greater than 255
(9.340.2.1)
RC = 24: The IP address has more than four parts
(9.2.3.2.4)
RC = 28: The IP address has less than four parts
(9.2.3)

When the return code is 0, a valid IP address is
returned. When the return code > 0, a return field
of binary zero is returned.

Written:  May 28'th 1994 at ITSO Raleigh

Modified:

***********************************************************************
WORKAREA DSECT
 DC 18F'0'  *Save area
DORD DC D'0'  *Work
STARTOUT DC A(0)  *Work
WORK DC CL4' '  *Work
NUMTEST DC CL4' '  *Work

TPIIADDR INIT 'Convert IP address from text to fullword', C
RLEN=YES,WORKLEN=64

* USING WORKAREA,R13
L  R9,0(R1)  *-- 15 bytes text string IP addr
L  R10,4(R1)  *-- Fullword return field
ST  R10,STARTOUT  *Save start of return field
LR  R2,R9  *Passed string starts here
LR  R8,R9
LA  R8,15(R8)  *First byte after string
LR  R3,R2  *Our advance pointer
NEXTCHAR EQU *
CLI 0(R3),C'.'  *A separator ?
BE  SEPFND  *-- Yes, we found one
CLI 0(R3),C' '  *A separator (The last one) ?
BE  SEPFND  *-- Yes, process element
CONTLOOP EQU *
LA  R3,1(R3)  *Advance one byte
CR  R3,R8  *Still inside string ?
BL  NEXTCHAR  *-- Yes, look on
SEPFND EQU *  
   *- No, treat as separator found
LR R4,R3  
   *Do not destroy advance pointer
SR R4,R2  
   *Where we started = length
CH R4,=AL2(3)  
   *Max 3 is allowed
BH SETRC8  
   *If more - RC=8
LTR R4,R4  
   *But must also be > 0
BNP SETRC12  
   *If not - RC=12
MVC WORK,=4C'0'  
   *Initialize work field
LA R7,WORK  
   *-> Start of work field
LA R15,4  
   *Length of work field
SR R15,R4  
   *Offset into work field
AR R7,R15  
   *-> Here to move into workfield
BCTR R4,0  
   *Length of bytes for Execute
EX R4,MVCBYTES  
   *Move bytes to work field
XC NUMTEST,NUMTEST  
   *Make ready for MVZ
MVZ NUMTEST,WORK  
   *Let us see zones
CLC NUMTEST,=4X'F0'  
   *Are we numeric ?
BNE SETRC16  
   *- No - RC=16
PACK DORD,WORK  
   *Pack to decimal
CVB R1,DORD  
   *Into binary form.
CH R1,=AL2(255)  
   *Max value
BH SETRC20  
   *If higher - RC=20
STCM R1,B'0001',0(R10)  
   *Return byte
LA R10,1(R10)  
   *--> Next return byte
CLI 0(R3),C' '  
   *Any more data?
BE SETRC0  
   *- No, we are done
LA R3,1(R3)  
   *--> Start of next string part
LR R2,R3  
   *New start base
CR R3,R8  
   *Are we still inside ?
BNL SETRC0  
   *- No, consider end of string
L R14,STARTOUT  
   *- -> First return byte
LA R14,4(R14)  
   *- -> first byte after return field
CR R10,R14  
   *We will only return 4 bytes
BL NEXTCHAR  
   *- OK, we are not there yet
B SETRC24  
   *- We will not return 5!!

* SETRC0 EQU *  
   *- -> First return byte
L R14,STARTOUT  
   *--> First return byte
LA R14,4(R14)  
   *- -> first byte after return field
CR R10,R14  
   *Did we return exact 4 bytes?
BNE SETRC28  
   *- No, set RC=28
SR R15,R15  
   *Go and return valid data
B RETURNDT

SETRC8 EQU *  
   *One part > 3 characters
B GETOUT

SETRC12 EQU *  
   *Zero length part
B GETOUT

SETRC16 EQU *  
   *Part is not numeric
B GETOUT

SETRC20 EQU *  
   *Part value > 255
B GETOUT

SETRC24 EQU *  
   *More than 4 parts
B GETOUT

SETRC28 EQU *  
   *Less than 4 parts
B GETOUT

GETOUT EQU *  
   *Return field
L R1,STARTOUT  
   *-> Return field
**G.4 TPIIOCTL Convert IOCTL Command Name to Command**

```
*****************************************************************************
** Name: TPIIOCTL **
** Function: Build COMMAND parameter for IOCTL call **
** COBOL has some problems with the command bitstrings. **
** Interface: R1 -> parameter list with two pointers: **
** +0 -> 16 char command string name (In) **
** +0 -> ioctl command fullword value (Out) **
** Logic: Build ioctl command based on command string **
** Abends: - none - **
** Returncode: - none - **
** Written: April 8'th 1995 at ITSO Raleigh **
** Modified: **
*****************************************************************************

WORKAREA DSECT
DC 18F'0' *Save area

TPIIOCTL INIT 'Build IOCTL command code', C
RENT=YES

USING WORKAREA, R13

L R2,0(R1) *-> 16 char command string
L R3,4(R1) *-> 4 byte return area
LM R5,R7,CMDBXLE
LOOP EQU *
CLC 0(16,R2),0(R5) *This command ?
BE FOUNDIT *- Yes
BXLE R5,R6,LOOP *Look them all
LA R15,8 *Set RC=8
B GETBACK *And return to caller
FOUNDIT EQU *
MVC 0(4,R3),16(R5) *Move back command value
SR R15,R15 *Set RC=0
GETBACK EQU *
TERM RC=R15 *Use value in R15 as RC

CMDBXLE DC A(FIRST,20,LAST)
FIRST DC CL16 'FIONBIO ',X'8004A77E'
DC CL16 'FIONREAD ',X'4004A77F'
DC CL16 'SIOCADDRT ',X'8030A70A'
DC CL16 'SIOCATMARK ',X'4004A707'
DC CL16 'SIOCDELRT ',X'8030A70B'
```
G.5 TPIWAIT Place Calling Process in Wait

*****************************************************
* * 
* Name: TPIWAIT * 
* * 
* Function: Wait a specified amount of time. * 
* * 
* Interface: R1 -> parameter list with one pointer: * 
* +0 -> fullword with waittime in milliseconds * 
* * 
* Logic: Wait the requested amount of time and return. * 
* * 
* Abends: - none - * 
* * 
* Returncode: - none - * 
* * 
* Written: April 8'th 1995 at ITSO Raleigh * 
* * 
* Modified: * 
* * 
*****************************************************
* 
WORKAREA DSECT
DC 18F'0' *Save area
WAITTIME DC A(0) *Wait interval
*
TPIWAIT INIT 'Wait a specified amount of time', C
RENT=YES,WORKLEN=256
*
USING WORKAREA,R13
*
L R9,0(R1) *-> Fullword with waittime in msec
L R7,0(R9) *Milliseconds
SR R6,R6 *Prepare for division
D R6,-A(10) *To get in 1/100 seconds
ST R7,WAITTIME *Store for STIMER
LA R2,WAITTIME *-> fullword with 1/100 seconds
STIMER WAIT, *Wait C
BINTVL=(R2)
*
TERM RC=0
LTORG
END

G.6 TPIRACF Interface to RACROUTE REQUEST=VERIFY User SVC

*****************************************************
* * 
* Name: TPIRACF * 
* * 
* Function: Interface routine to user SVC 236 for verification of * 
* user and construction of task level security * 
* * 
* Written: * 
* * 
*****************************************************
* 
A Beginner's Guide to MVS TCP/IP Socket Programming 257
* Interface: R1 -> Parameter list with 6 pointers: *
* +0 -> 4 byte request kode (In) *
*  0: Verify user and establish task level security environment *
*  4: Verify user; do not establish task level security environment *
*  8: Reset security environment for this task to address space environment *
* +4 -> 8 byte userid (In) *
* +8 -> 8 byte password (In) *
* +12-> 8 byte new password (In) *
* +16-> 8 byte RACF group (In) *
* +20-> 8 byte application (In) *
* *
* New password, RACF group and application must be be passed as space if they are not relevant for this call.
* *
* Password and new password must be passed in clear.
* *
* Logic: 1. Validate parameters *
*  2. Build TPIRACFA area *
*  3. Issue SVC236 with R1 pointing to TPIRACFA *
*  4. Convert return codes to something understandable *
*  5. Return to caller *
* *
* Abend: none *
* *
* Returncode: 0 : Everything is OK *
*  4 : Userid is not defined to RACF *
*  8 : Password is invalid *
* 12 : Password is expired - new password required *
* 16 : New password not a valid password *
* 20 : Userid is not part of the passed group *
* 24 : Userid is revoked *
* 28 : Access to group is revoked *
* 32 : Userid is not authorized to application *
* 252 : Caller not authorized to use SVC 236 *
* 253 : There is no task level env. to delete *
* 254 : Error in passed parameters *
* 255 : Request in error - of other reasons. *
* *
* Written: April 7'th, 1995 - ITSO Raleigh *
* *
***********************************************************************
* TPIWORK DSECT
  DS 104X'00' *Save Area
MACWORK DC X128'00' *Macro work area
PARMPTR DC A(0) *Parameter pointer at entry
TPIRACFA TPIRACFA *Interface area to SVC 236
WTOWORK DS 0F *WTO Message build area
  DC XL4'00',C'TPIRACF - ' *Placeholder
  DC C'Function=' *Placeholder
RACREQ DC CL4' ' *Request kode
  DC C' User=' *Placeholder
RACUID DC CL8' ' *Userid
  DC C' Group=' *Placeholder
RACGRP DC CL8' ' *Group
  DC C' Appl=' *Placeholder
RACAPP DC CL8' ' *Application
DC C' SAF RC=' *Placeholder
RACSAF DC CL4' ' *SAF RC
DC C' RACF RC=' *Placeholder
RACRC DC CL4' ' *RACF RC
DC C' RACF Reason=' *Placeholder
RACREAS DC CL4' ' *RACF Reason
DC C' TPIRACF RC=' *Placeholder
RACMYRC DC CL4' ' *Return code from TPIRACF
DC XL4'00' *Placeholder
DORD DC D'0' *For work
WORKRC DC A(0) *For work

* TPIRACF INIT 'TPI - RACF interface',RENT=YES,WORKLEN=512, 
   C MODE=31
 *
   USING TPIWORK,R13
   ST R1,PARMPTR
   LR R11,R1 *Save parms pointer
 *
   MVC TPIRACFA(TPIRACFL),=XL(TPIRACFL)'00'
   MVC TPIREYEC,=CL8'TPIRACFA' *Eyecatcher
   MVC WTOWORK(WTOLISTL),WTOLIST *Init WTO area
 *
* ---------------------------------------------------------------------
* Check all parameters and build TPIRACFA area
* ---------------------------------------------------------------------
*
   L R2,0(R1) *-> request code
   L R2,0(R2) *Request code
   MVC RACREQ,=CL4'CRE' *Create
   CH R2,=AL2(0) *Code = 0 is OK
   BE REQOK
   MVC RACREQ,=CL4'TEST' *Just test
   CH R2,=AL2(4) *Code = 4 is OK
   BE REQC CRE
   MVC RACREQ,=CL4'N/A' *Unknown code
   CH R2,=AL2(8) *Code = 8 is OK
   BNE BADPARMS
   MVC RACREQ,=CL4'DELE' *Delete task level env.
   B REQOK
REQCRE EQU *
SR R2,R2 *TEST starts with a create
REQOK EQU *
ST R2,TPIRREQ *Request code for interface
CH R2,=AL2(8) *ENVIR=DELETE?
BE PARMSOK *No further parms needed
L R2,4(R1) *-> 8 bytes userid
MVC TPIRUID,0(R2) *Userid
MVC RACUID,0(R2) *Trace line
LA R2,TPIRUIDL
BAL R14,CALLE N *Get l userid
L R2,8(R1) *-> 8 bytes password
MVC TPIRPWD,0(R2) *Password
LA R2,TPIRPWDL
BAL R14,CALL EN *Get l password
L R2,12(R1) *-> 8 bytes new password
MVC TPIRN PW,0(R2) *New password
LA R2,TPIRNPW L
BAL R14,CALL EN *Get l new password
L R2,16(R1) *-> 8 bytes group
MVC TPIRGRP,0(R2) *Group
MVC RACGRP,0(R2) *Trace line
LA R2,TPIRGRPL
BAL R14,CALLEN *Get l'group
L R2,20(R1) *-> 8 bytes application name
MVC TPIRAP,0(R2) *Application name
MVC RACAPP,0(R2) *Trace line
LA R2,TPIRAPPL
BAL R14,CALLEN *Get l'application
*
CLC TPIRREQ,=A(4) *0 and 4 require certain parms
BH PARMSOK *8 requires no special parms
CLI TPIRUIDL,0 *We must have a user ID
BE BADPARMS
CLI TPIRPWDL,0 *And a password
BE BADPARMS
PARMSOK EQU *
* *---------------------------------------------------------------------
* *---------------------------------------------------------------------
* Issue SVC call with R1 pointing to TPIRACFA
* Create trace line after return from user SVC 236
* *---------------------------------------------------------------------
* *---------------------------------------------------------------------
* LA R1,TPIRACFA *-> TPIRACFA interface area
SVC 236 *User SVC 236
*
RCTEST EQU *
CH R15,=AL2(250) *Is it our own RC?
BH SETRCOWN *- Yes, pass it back
LR R2,R15 *Save it for later use
TPIHEX TPIRSAF+2,RACSAF *SAF returncode
TPIHEX TPIRRC+2,RACRC *RACF returncode
TPIHEX TPIRREAS+2,RACREAS *RACF reasoncode
LR R15,R2 *SVC236 R15
*
* *---------------------------------------------------------------------
* *---------------------------------------------------------------------
* Analyze returncodes from SAF and RACF; and set returncode from this
* routine
* *---------------------------------------------------------------------
* CLC TPIRSAF,=A(0) *RC=0 from SAF is OK
BE SETRC0 *OK
CLC TPIRSAF,=A(4) *SAF RC=4 ?
BNE TSAF08 *- no, test for SAF RC=8
CLC TPIRRC,=A(4) *RACF RC=4 ?
BE SETRC4 *User is unknown
B SETRC255 *Garbage can
TSAF08 EQU *
CLC TPIRSAF,=A(8) *SAF RC=8 ?
BNE SETRC255 *- No, Garbage can
CLC TPIRRC,=A(8) *RACF RC=8
BE SETRC8 *- Yes, Password invalid
CLC TPIRRC,=A(12) *RACF RC=12 X'0C'
BE SETRC12 *- Yes, Password expired
CLC TPIRRC,=A(16) *RACF RC=16 X'10'
BE SETRC16 *- Yes, New password invalid
CLC TPIRRC,=A(20) *RACF RC=20 X'14'
Subroutine for calculation of length byte in interface to RACROUTE

CALLEN EQU *
LA R3,1(R2) *-Start field
LA R4,8 *Max length

CALLOOP EQU *
CLI 0(R3),0 *X'00' terminates
BE CALEND
CLI 0(R3),X'40' *X'40' terminates also
BE CALEND
LA R3,1(R3) *Advance pointer
BCT R4,CALLOOP

CALEND EQU *
LA R3,8 *max length
SR R3,R4 *no. loops=length
STC R3,0(R2) *Length field in here
BR R14 *back to mainline

Returncode settings

SETRC0 EQU *
LA R6,0 *OK
CLC RACREQ,=CL4'TEST' *If call was for test
BNE RETUR
CLC TPIRREQ,=AL4(TPIRALL) *and we did a CREATE
BNE RETUR
MVC TPIRREQ,=AL4(TPIRDEL) *Then we must delete it again
MVI TPIRUIDL,0 *No user ID
MVI TPIRPWDL,0 *No password
MVI TPIRNFWL,0 *No new password
MVI TPIRGRPL,0 *No group ID
MVI TPIRAPPPL,0 *No appl ID
LA R1,TPIRACFA *-> TPIRACFA interface area
SVC 236 *User SVC 236
LTR R15,R15 *Should only be OK?
BZ RETUR *- Yes, it is
B RCTEST *Redrive return code testing

SETRC4 EQU *
LA R6,4 *User unknown
B RETUR

SETRC8 EQU *
LA R6,8 *password invalid
B RETUR

SETRC12 EQU *
LA R6,12 *password expired
B RETUR

SETRC16 EQU *
LA R6,16 *new password invalid
B RETUR

SETRC20 EQU *
LA R6,20 *user not in group
B RETUR

SETRC24 EQU *
LA R6,24 *user revoked
B RETUR

SETRC28 EQU *
LA R6,28 *access to group revoked
B RETUR

SETRC32 EQU *
LA R6,32 *not auth. to appl.
B RETUR

SETRCOWN EQU *
LR R6,R15 *Pass unchanged back to caller
B RETUR

SETRC255 EQU *
LA R6,255 *garbage can returkode
B RETUR

BADPARMS EQU *
LA R6,254 *Error in passed params
B RETUR

RETUR EQU *
ST R6,WORKRC *Just for formatting
TPIHEX WORKRC+2,RACMYRC *RC from TPIRACF
WTO MF=(E,WTOWORK) *Put out a WTO
TERM RC=R6 *Return with RC as set in R6
LTORG

WTOLIST WTO 'TPIRACF - Function=n/a User=n/a Group=n/a ApC
pl=n/a SAF RC=n/a RACF RC=n/a RACF Reason=n/a TPC
IRACF RC=n/a ','MF=L

WTOLISTL EQU *-WTOLIST
END

MACRO
&NAME TPIRACFA &TYPE=CSECT
AIF ('&TYPE' EQ 'DSECT').DSECT
&NAME DS 0F (*TPIRACFA section
AGO .GENCODE
.DSECT ANOP
&NAME DSECT (*TPIRACFA section
 .GENCODE ANOP
TPIREYEC DC CL8'TPIRACFA' *Eye catcher
TPIRSAF DC AL4(0) *SAF RC
TPIRRC DC AL4(0) *RACF RC
TPIRREAS DC AL4(0) *RACF Reason code
TPIRREQ DC F'0' *Request code
TPIRALL EQU 0 *Verify and build sec. env
TPIREDL EQU 8 *Delete sec env.
TPIRUIDL DC CL8' ' *L'userid
TPIRUID DC CL8' ' *Userid
TPIRPWDL DC AL1(0) *L'password
TPIRPWD DC CL8' ' *password
TPIRNPWL DC CL8' ' *L'new password
TPIRNPW DC CL8' ' *new password
TPIRGRPL DC CL8' ' *L'groupid
TPIRGRP DC CL8' ' *groupid
TPIRAPPL DC AL1(0) *L'application

A Beginner's Guide to MVS TCP/IP Socket Programming
G.7 User SVC for RACROUTE REQUEST=VERIFY

**---------------------------------------------**
* *
* Name: IGC00236 - SVC 236 - IGC0023F *
* *
* Function: User SVC 236 type 4 - Do RACROUTE REQUEST=VERIFY *
* for verification of userid/pw and creation of a *
* task level security environment. *
* Address space user ID of calling task must be *
* authorized to use this SVC by having read access to *
* FACILITY resource TPI.RACINIT *
* *
* Interface: R1 -> Interface area, mapped with macro TPIRACFA. *
* Area has been constructed by interface module *
* TPIRACF, which issues the SVC 236. *
* R3 -> CVT *
* R4 -> TCB for calling task *
* R5 -> SVRB *
* R6 -> Entry Point address *
* R7 -> ASCB *
* R14 -> Return address *
* *
* Register contents has been saved before entry to *
* this SVC routine. *
* R2-R14 will be restored by MVS before control is *
* passed back to program that issued the SVC 236. *
* *
* Logic: 1. Verify that R1 points to a valid TPIRACFA control *
* block and that caller has access to it. *
* 2. Copy TPIRACFA control to our getmained storage *
* 3. Issue RACROUTE REQUEST=AUTH to see if calling *
* address space user ID has read access to *
* FACILITY resource TPI.RACINIT *
* 4. Initialize SAFP parameter list with the passed *
* values for userid, password etc. *
* TPIRACF has verified that the needed parameters *
* for the request is included in TPIRACFA. *
* 5. Issue RACROUTE REQUEST=VERIFY *
* 6. Set key to callers key and store RACROUTE return *
* codes back into callers TPIRACFA *
* 7. Return to caller *
* *
* Abends: none (hopefully!) *
* *
* Returncode: 0-250: Return code from SAF *
* 252: Calling address space user is not authorized to *
* use this user SVC. *
* 253: No task level security environment to delete. *
* 254: R1 does not point to a valid TPIRACFA control *
* block on entry to SVC routine. *
* *
* Written: ITSO Raleigh April 10, 1995. *
* *
**---------------------------------------------**
ICHSAFP *SAF Parameter list
CVT   DSECT=YES *CVT
IHAPS A *PSA
IHARB  *RB
IKJTCB *TCB
IHAASCB *ASCB
IHAASXB *ASXB

* TPIRACFA TPIRACFA TYPE=DSECT
* TPIWORK DSECT
MACWORK DC 256'00' *Macro work area
GETMADR DC A(0) *Getmained storage address
GETMLEN DC A(0) *Getmained storage length
PARMPTR DC A(0) *-> passed user parmlist
PARMCOPY DC (TPIRACFL)'00' *Copy of user parmlist
SAFWORK DC 512'00' *SAF router workarea
WORKLEN EQU -TPIWORK *Length to getmain

IGC00236 TITLE 'ITSO User type 4 SVC number 236 - RACROUTE VERIFY'
IGC00236 CSECT
IGC00236 AMODE 31
IGC00236 RMODE ANY

* GENERAL PURPOSE REGISTER EQUATES
*
R0 EQU 0
R1 EQU 1
R2 EQU 2
R3 EQU 3
R4 EQU 4
R5 EQU 5
R6 EQU 6
R7 EQU 7
R8 EQU 8
R9 EQU 9
R10 EQU 10
R11 EQU 11
R12 EQU 12
R13 EQU 13
R14 EQU 14
R15 EQU 15

* LR R12,R6 *Entry point address
USING IGC00236,R12 *Addressability
LR R8,R14 *Save return address ptr.
USING CVT,R3 *Comm. Vector Table
USING TCB,R4 *Task Control Block
USING RBBASIC,R5 *Request Block common part
USING ASCB,R7 *Address Space Control Block
LR R9,R1 *-> TPIRACFA interface area
LA R0,WORKLEN *L'Workarea
GETMAIN R,LA=(0) *Getmain workarea storage
LR R11,R1 *Here it is
USING TPIWORK,R11 *Workarea addressability
ST R11,GETMADR *For later freemain
ST R0,GETMLEN *--do--

* * * * * * * * *----------------------------------------------------------------
 *
* To be sure that caller does not pass a pointer to storage that he/she
* has no access to, we use modeset to set key to user key before we
reference all bytes in the passed area.
* We then switch back to our own key and copy the parameter area into
* our own getmained storage, so the caller is not able to modify them
* on the fly.
* If interface area does not have a valid eyecatcher, we return
* to the caller with RC=254
* ---------------------------------------------------------------------

PRINT GEN
ST R9,PARMPTR *Save pointer to user parmlist
MODESET EXTKEY=RBT234, *Ensure proper fetch protect C
          WORKREG=2
MVC 0(TPIRACFL,R9),0(R9) *Copy to itself for byte ref.
MODESET EXTKEY=ZERO, *Back to SVC key C
          WORKREG=2
PRINT NOGEN
MVC PARMCOPY(TPIRACFL),0(R9) *Copy interface area to us
LA R9,PARMCOPY *-- Copy of interface area
USING TPIRACFA,R9 *Hereafter we access our copy
CLC TPIREYEC,=CL8' TPIRACFA' *Valid Eyecatcher?
BNE RETUR254 *-- No, return with RC=254
* ---------------------------------------------------------------------

* To control who is using this SVC, we ask RACF if address space
* user ID has READ access to FACILITY class resource TPI.RACINIT
* If AS user is not authorized or no AS ACEE exists, we return
* to caller with RC=252
* *---------------------------------------------------------------------

INTFOK EQU *
L R2,ASC BASXB *-- ASCB Extension
USING ASXB,R2 *ASXB
ICM R2,15,ASXBSENV *-- Address Space ACEE
BZ RETUR252 *No AS ACEE exists, RC=252
MVC MACWORK(AUTHPL),AUTHP *RACROUTE AUTH Parm list
LA R10,MACWORK *-- SAFP
USING SAFP,R10 *Addressability RACROUTE parms
RACROUTE REQUEST=AUTH, *Authorization request C
        ATTR=READ, *We want READ access to C
        ENTITYX=TPIRES, *TPI.RACINIT C
        ACEE=(R2), *Check against AS user C
        LOGSTR=LOGSTR, *For logging purposes C
        WORKA=SAFWORK, *512 bytes work area C
        RELEASE=1.9, *Required for ENTITYX keyword C
        MF=(E,MACWORK) *Use prebuilt parmlist
LTR R15,R15 *We will only accept SAF RC=0
BNZ RETUR252 *Else return with RC=252
* *---------------------------------------------------------------------

* Build parameter list for RACROUTE REQUEST=VERIFY call based
* on the passed values in the interface area from the caller.
* *---------------------------------------------------------------------

CLC TPIRREQ, =A(TPIRDEL) *Delete request?
BNE NOTDEL *-- No
ICM R14,15,TCBENV *Do we have a TCB ACEE?
BZ RETUR253 *-- No, return with RC=253

NOTDEL EQU *

MVC MACWORK(VERPL),VERP *VERIFY parameter list
CLI TPIRUIDL,0 *Do we have user ID?
BE NOUID *-- No, no user ID passed

RACROUTE REQUEST=VERIFY,
USERID=TPIRUIDL,
MF=(M,MACWORK)
* Put in user ID

NOUID EQU *

CLI TPIRPWDL,0 *Do we have a password?
BE NOPWD *-- No, no password passed

RACROUTE REQUEST=VERIFY,
PASSWRD=TPIRPWDL,
MF=(M,MACWORK)
* Put in password

NOPWD EQU *

CLI TPIRNPWL,0 *Do we have new password?
BE NONPW *-- No, no new password passed

RACROUTE REQUEST=VERIFY,
NEWPASS=TPIRNPWL,
MF=(M,MACWORK)
* Put in new password

NONPW EQU *

CLI TPIRGRPL,0 *Do we have a group ID
BE NOGRP *-- No, no group id passed

RACROUTE REQUEST=VERIFY,
GROUP=TPIRGRPL,
MF=(M,MACWORK)
* Put in group id

NOGRP EQU *

CLI TPIRAPPL,0 *Do we have an appl name?
BE NOAPP *-- No, no appl name passed

RACROUTE REQUEST=VERIFY,
APPL=TPIRAPPL,
MF=(M,MACWORK)
* Put in application name

NOAPP EQU *

CLC TPIRREQ,=A(TPIRALL) *ENVIR=CREATE?
BNE NOTCREAT *-- No.

RACROUTE REQUEST=VERIFY,
ENVIR=CREATE,
MF=(M,MACWORK)
* Put in CREATE

NOTCREAT EQU *

RACROUTE REQUEST=VERIFY,
ENVIR=DELETE,
MF=(M,MACWORK)
* Put in DELETE

* * * *---------------------------------------------------------------------
* * Issue the RACROUTE REQUEST=VERIFY call.
* * Use modeset to set key to callers key, before we store return
* * code values back into the caller's interface area.
* *
* *---------------------------------------------------------------------
* *
DORAC EQU *

RACROUTE REQUEST=VERIFY,
LOGSTR=LOGSTR,
WORKA=SAFWORK,
RELEASE=1.9,
MF=(E,MACWORK)
* Do VERIFY request
* Identify us as caller
* Work area
* SAF release

LR R7,R15 *Save SAF RC
L R9,PARMPTR *Restore pointer to user parms
MODESET EXTTKEY=RBT234,*Ensure proper store protect
WORKREG=2 *
STCM R7,'1111',TPIRSAF *SAF RC
MVC TPIRRC,SAPPRRET *RACF RC
MVC TPIREAS,SAPPRREA *RACF Reason kode
MODESET EXTKEY=ZERO, *Reset to SVC key C
WORKREG=2 *
B FREESTOR *Go to freemain
RETUR252 EQU * *Not authorized to use SVC
LA R7,252 *RC=252
B FREESTOR *Go to freemain
RETUR253 EQU * *No task level env. to delete
LA R7,253 *RC=253
B FREESTOR *Go to freemain
RETUR254 EQU * *Invalid eyecatcher
LA R7,254 *RC=254
FREESTOR EQU *
L R1,GETMADR *This to freemain
L R0,GETMLEN *Length to freemain
FREEMAIN R,A=(R1),LV=(R0) *Freemain storage
LR R15,R7 *Return code to R15
LR R14,R8 *Restore return address ptr.
BR R14 *Get back.
LTORG

VERP RACROUTE REQUEST=VERIFY,MF=L
VERPL EQU *-VERP
AUTHP RACROUTE REQUEST=AUTH,CLASS='FACILITY',MF=L
AUTHPL EQU *-AUTHP
LOGSTR DC AL1(L'LOGTXT)
LOGTXT DC C'TPI Routines - RACROUTE VERIFY'
DS 0F
TPIRES DC AL2(20,0),CL20'TPI.RACINIT'
END

G.8 TPIAUTH Issue RACROUTE REQUEST=AUTH for FACILITY Class

***********************************************************************
* *
* Name: TPIAUTH *
* *
* Function: Issue a RACROUTE REQUEST=AUTH for a FACILITY class *
* resource *
* *
* Interface: R1 -> parameter list *
* +0 Pointer to 80 char resource name (In) *
* +4 Pointer to 8 char access intent (In) *
* *
* Logic: Issue a RACROUTE REQUEST=AUTH for the resource name *
* passed. Authorization will be done via std. ACEE *
* search order: Use TCBSENV if it is non-zero. If *
* TCBSENV is zero, use address space security environment.*
* *
* Returncode: 0: Access is allowed *
* 4: Access is not allowed *
* *
* Written: March 27'th 1994 at ITSO Raleigh *
* *
* Modified: *
* *
***********************************************************************
*
PARMS DSECT
PNAME DC A(0) *-> 80 char resource name
PACCESS DC A(0) *-> 8 char access intent
*
TPIWORK DSECT
DC 18F'0' *Save area
RACLWORK RACROUTE REQUEST=AUTH, *RACROUTE Macro expansion
  ENTITY=(0),
  CLASS='FACILITY',
  ATTR=READ,
  WORKA=0,
  RELEASE=2.1,
  MF=L
WTOWORK WTO 'TPIAUTH SAF RC=xxxx RACF RC=xxxx RACF Reason=xxxx',
  MF=L
WTOSAFRC EQU WTOWORK+19,4 *SAF RC in WTO line
WTORACRC EQU WTOWORK+32,4 *RACF RC in WTO line
WTORACRS EQU WTOWORK+49,4 *RACF Reason in WTO line
DORD DC D'0' *Unpack and edit work
RACFWORK DC 512X'00' *SAF work area
ENTITYBF DC AL2(80,0) *ENTITYX buffer
ENTITYNM DC CL80' ' *Resource name
*
TPIAUTH INIT 'Issue RACROUTE REQUEST=AUTH for a FACILITY resource',
  MODE=31,RENT=YES,WORKLEN=1024
  USING TPIWORK,R13 *Adressability work areas
  LR R2,R1 *Save parm pointer
  USING PARMS,R2 *Adressability parameters
  L R3,PACCESS *-> 8 char access intent
  LM R7,R9,BXLEINT *Access intent table
INTLOOP EQU *
  CLC 0(8,R3),0(R7) *This access intent ?
  BE GOTINT *- Yes.
  BXLE R7,R8,INTLOOP *Look through them all
  LA R7,INTST *Use READ as default
GOTINT EQU *
  SR R3,R3
  IC R3,8(R7) *Access intent code
  L R4,PNAME *-> resource name
  MVC ENTITYBF(4),=AL2(80,0) *Initialize buffer header
  MVC ENTITYNM,0(R4) *Move name to 80 byte buffer
*
  MVC RACFWORK(RACLISTL),RACLIST
  RACROUTE REQUEST=AUTH, *Authorize request
    ENTITYX=ENTITYBF,
    ATTR=(R3),
    WORKA=RACFWORK,
    RELEASE=2.1,
    MF=(E,RACFWORK)
*
  LR R11,R15 *Save return code from SAF
  MVC WTOWORK(WTOLIST),WTOLIST
  LR R7,R11 *Prepare for double shift
  SR R6,R6 *Make ready for double shift
  SLDL R6,4 *00000000x xxxxxxx0
  STM R6,R7,DORD *Store for Unpack
  UNPK WTOSAFRC,DORD *Unpack
  NC WTOSAFRC,=4X'0F' *Remove F's
  TR WTOSAFRC,TRHEX *Translate to EBCDIC
  L R7,RACLIST *RACF RC
  SR R6,R6 *Make ready for double shift
  SLDL R6,4 *00000000x xxxxxxx0
H.0 Appendix H. Sample MVS Concurrent Server

This appendix contains a description of a sample socket application that was developed during the creation of this book.

The application is called TCP/IP Programming Interfaces (TPI) and consists of the following components:

1. A concurrent server implemented in an MVS address space and based on the Sockets Extended assembler macro interface
2. A REXX client using REXX sockets
3. A CICS client written in COBOL using Sockets Extended call interface
4. An IMS client written in COBOL using Sockets Extended call interface
5. A REXX client using REXX socket used to load the database
The purpose of the application was to illustrate as many of the new IBM TCP/IP Version 3 Release 1 for MVS programming interfaces as possible.

The server maintains a DB2 table called `tpidata` with administrative information related to TCP/IP hosts in the ITSO-Raleigh environment. The REXX client uses ISPF panels as user interface, and connects to the server address space for add, query, update or delete requests of DB2 data.

The IMS and CICS clients are query only clients.

H.1 TPI Concurrent MVS Server
H.2 TPI REXX Client Application
H.3 TPI DB2 Table Definition
H.4 Sample Log from TPI Server Execution

**H.1 TPI Concurrent MVS Server**

The concurrent TPI server is implemented in an MVS address space (started task or batch job). It is based on the Sockets Extended assembler macro programming interface.
SELECT (asynchronous)
Wait on ECB-list
If select was posted
    SYNC
Do until no more SDs
    If read selected on listener SD
        ACCEPT
        If free server task available
            GIVESOCKET to server subtask
            Post server subtask
        else
            Prepare error message
    If exception selected
        CLOSE
    If write selected
        SEND pending error message
    end-do
If modify ECB was posted
    Signal closedown
If server subtask terminated
    if reinstate count < limit
        reinstate server subtask
    else
        Signal closedown
If logwriter task terminated
    Signal closedown
end-do
Post server subtasks with RC=4 to stop
Request logwriter DST to stop
exit

Figure 63. TPI Server Address Space Logic

H.1.1 TPIMAIN Concurrent Server Main Process
H.1.2 TPILOGWT Logwriter Data Services Task
H.1.3 TPISERV Concurrent Server Subtask
H.1.4 TPISERVD Concurrent Server DB2 Access
H.1.5 TPISEND Send Data Over a Stream Socket
H.1.6 TPIRECV Receive Data Over a Stream Socket
H.1.7 TPIMCB Macro Main Task Control Block
H.1.8 TPISCB Macro Subtask Control Block
H.1.9 TPILOG Macro Issue Logwriter Request
H.1.10 TPITRC Macro Issue Trace Request
H.1.11 TPIMASK Macro Set and Test Bits in Select Mask
H.1.12 TPIREC Macro DB2 Row Layout
H.1.13 TPIMSO Macro Socket Descriptor Table

H.1.1 TPIMAIN Concurrent Server Main Process

*******************************************
* Name: TPIMAIN *
* Function: TCP/IP Programming Interfaces sample ITSO application main module. *
* Interface: - none - *

A Beginner's Guide to MVS TCP/IP Socket Programming 271
Logic: This module is the main task module in the TPI Server application. It is started either as a normal batch job or as a started task. The main logic is as follows:

1. Initialize Main task Control Block (TPIMCB)
2. Establish EZA Global Work Area addressability
3. Get main storage for Server task Control Blocks (TPISCB)
4. Attach Log Writer Task
5. Initialize socket environment via the INITAPI function and fetch our own TCP/IP Client id
6. Get main storage for Main task Socket descriptor table (TPIMSO) and initialize entries.
7. Attach server subtasks and initialize TIPSCBs
8. Set up, so operator can issue a /P command to stop server address space
9. Get a socket to be used for listen
10. Bind the socket to the TPI Server application port number
11. Issue a Listen on the socket
12. Here starts Main task loop:
   - Build bit masks for select command and issue an asynchronous select (with an ECB keyword)
13. Wait on an ECBlist including:
   - select ECB
   - operator modify ECB
   - log writer subtask termination ECB
   - server subtask termination ECBS
14. When wait comes through, analyze event:
   - Select: a. Issue a socket SYNC call to synchronize with socket interface.
   - Analyze all returned bits in the select bitmasks.
   - b. If read selected: Issue an accept, a givesocket, and post a free server subtask.
   - c. If exception selected: Issue a close socket (Server subtask has taken socket with a takesocket call).
   - d. If write selected: Write out any pending error message and close socket.
   - Modify: Request all subtasks to terminate and close down.
   - Subtask termination: If it is log writer task, treat it as a shutdown request. If it is a server subtask, reinstate the subtask (keeping track of number of reinstates in order to avoid abend loops).
15. Continue with item no. 12 above.

Abends: U1000: Could not find posted ECB after Wait on ECBLIST
Returncode: - none -
Written: May 28'th 1994 at ITSO Raleigh
Modified:

***********************************************************************
PRINT NOGEN
A Beginner's Guide to MVS TCP/IP Socket Programming

COMMAREA DSECT *For EXTRACT macro
IEZCOM
CIBAREA DSECT *For QEDIT
IEZCIB
IHAASCBC *ASCB layout
PRINT GEN
TPISCB TPISCB *Server task Control Block dsect
TPIMSO TPIMSO *Main task socket descriptor table
TPIMAIN INIT 'TPI Main Task',RENT=NO,MODE=24,BASE=(12,11,10)

*---------------------------------------------------------------------*
* Initialize default runtime parameters *
*---------------------------------------------------------------------*
MVC TPIMDB2,=CL4'DSNI' *DB2 subsystem name
MVC TPIMTCP1,=CL8'T18ATCP' *TCP/IP AS name
MVC TPIMPORT,=AL2(9999) *Port number
MVC TPIMNOST,=AL2(2) *Start 2 server tasks
MVC TPIMMAXS,=AL2(50) *Max 50 sockets
MVC TPIMMAXD,=AL4(50) *Max 50 socket descriptors
L R3,X'10' *-> CVT
L R3,0(R3) *-> TCB Words
L R15,12(R3) *-> Current ASCB (My ASCB)
L R3,4(R3) *-> Current TCB (My TCB)
SR R2,R2 *Make ready for double shift
SLDL R2,4 *0000000x xxxxxxx0
STM R2,R3,DORD *Store for Unpack
UNPK TPIMTCBE,DORD *Unpack
NC TPIMTCBE,=8X'0F' *Remove F's
TR TPIMTCBE,TRHEX *Translate to EBCDIC
 USING ASCB,R15
ICM R14,15,ASCBJBNI *-> Jobname if initiated
BNZ INITJBN *If not zero, pointer is OK
ICM R14,15,ASCBJBNS *-> Jobname if start/logon
BNZ INITJBN *If not zero, pointer is OK
MVC IDENTJOB,=CL8' ' *We did not find a jobname
B INITJOBS *Job name is initialized

INITJBN EQU *
MVC IDENTJOB,0(R14) *Move in job name
DROP R15

INITJOBS EQU *
LA R15,MAINGLOB *-> EZA Global work area
ST R15,TPIMGLOB *Subtasks will need it
TPITRC TYPE=INIT, *Enable trace points
 C
TRACE=YES, *
 C
MOD=TPIMAIN *Tracing module is TPIMAIN

*---------------------------------------------------------------------*
* Getmain storage for server task control blocks (TPISCB) *
*---------------------------------------------------------------------*
LH R2,TPIMNOST *Number of server tasks
MH R2,=AL2(TPISCBLN) *Multiply by TPISCB length
STORAGE OBTAIN, *Getmain for pool of
 C
LENGTH=(R2), * Server task Control
 C
LOC=BELOW * Blocks
ST R1,TPIMSCBB *-> First TPISCB
MVC TPIMSCBB+4(4),=A(TPISCBLN) *L'TPISCB Entry
LH R2,TPIMNOST *Number of TPISCB entries
BCTR R2,0 *The last number
MH R2,=AL2(TPISCBLN) *Times length per entry
AR R2,R1 *-> Last TPISCB entry
ST R2,TPIMSCBB+8 *Ready for BXLE

*---------------------------------------------------------------------*
* *
* Start up our Log writer task. *
* It will wait for work on ECB: TPIMLECB *
* *
*---------------------------------------------------------------------*

XC TPIMLDON,TPIMLDON *Clear for wait
ATTACH EP=TPILOGWT, *Module name: TPILOGWT C
PARAM=(TPIMCB), *Pass Main task Control Block C
ECB=ECBTLOGW *Termination ECB
ST R1,TPIMLTCB *-> Log writer TCB
WAIT ECB=TPIMLDON *LOGWT will post, when init done.
XC TPIMLDON,TPIMLDON *Clean up

*---------------------------------------------------------------------*

* Initialize socket API *
* *
* Get our client id and log it to the log file *
* *
*---------------------------------------------------------------------*

MVC IDENTTCP,TPIMTCPI *TCP/IP Address space name
MVC TRCMLFUN,=CL8'INITAPI'
EZASMI TYPE=INITAPI, *Initialize socket interface C
MAXSOC=TPIMMAXS, *So many concurrent sockets C
SUBTASK=TPIMTCBE, *My TCB address in EBCDIC C
IDENT=IDENTSTR, *TCP/IP AS name and my AS name C
MAXSNO=TPIMMAXD, *Max. no of socket descriptors C
ERRNO=ERRNO, C
RETCODE=RETCODE, C
ERROR=EZAERROR
ICM R15,15,RETCODE *Initapi OK
BM EZAERROR *- No.
MVC TRCMLFUN,=CL8'GETCLNID'
EZASMI TYPE=GETCLIENTID, *Get our own client id C
CLIENT=TPIMCLNI, *Store it in Main task Control bl. C
ERRNO=ERRNO, C
RETCODE=RETCODE, C
ERROR=EZAERROR
ICM R15,15,RETCODE *Was it OK
BM EZAERROR *- No, stop now.
L R15,TPIMCDOM *Addressing Family
CVD R15,DORD *From binary to decimal
OI DORD+7,'X'0F' *A nice sign.
UNPK CLNLOGAF,DORD *Into logging line
MVC CLNLOGAS,TPIMCNAM *Address Space Name
MVC CLNLOGST,TPIMCTSK *Subtask Name
TPILOG TEXT=CLNLOGLN, *Log client id C
MSGNO=1, *Text is prebuilt C
MOD=TPIMAIN *Main is logging the message

*---------------------------------------------------------------------*

* Getmain storage for Main task socket descriptor table. *
* Initialize socket descriptor table. *
* *
*---------------------------------------------------------------------*

L R2,TPIMMAXD *Max. number of socket descriptors
MH R2,=AL2(TPIMSOLN) *Multiply by TPIMSO length
STORAGE OBTAIN, *Getmain for pool of socket C
LENGTH=(R2), *- descriptor control C
LOC=BELOW *- blocks
ST R1,TPIMSOTB *-> First TPIMSO
MVC TPIMSOTB+4(4),=A(TPIMSOLN) *TPIMO Entry
LH R2,TPIMMAXS *Number of TPIMO entries
BCTR R2,0 *The last number
MH R2,=AL2(TPIMSOLN) *Times length per entry
AR R2,R1 *-> Last TPIMO entry
ST R2,TPIMSOTB+8 *Ready for BXLE
LM R1,R3,TPIMSOTB *TPIMO BXLE addresses
SR R4,R4 *Socket counter

INITMSO EQU *
XC TPIMO(TPIMSOLN),TPIMO *Clear TPIMO entry
MVC TPIMSEYE,=CL8'TPIMO' *Move in eyecatcher
STH R4,TPIMSNO *Socket number
LA R4,1(R4) *Advance
BXLE R1,R2,INITMSO *Do them all
DROP R1

*---------------------------------------------------------------------*
* Start Server subtasks and initialize TPISCB entries *
*---------------------------------------------------------------------*
LA R6,ECBPSTS *First Subtask Term. ECB pointer
LM R3,R5,TPIMSCBB *Loop addresses for TPISCBs

INITSCB EQU *
XC TPISCB(TPISCBLN),TPISCB *Clear storage
MVC TPISSEYE,=CL8'TPISCB' *Move in eyecatcher
MVC TPISMCB,=A(TPIMCB) *-> TPIMCB
LA R8,TPISTECB *-> Term. ECB
ATTACH EP=TPISERV, *Server subtask main module C
PARAM=((R3)), *Pass TPISCB as only parameter C
ECB=(R8) *Termination ECB
ST R1,TPISTCB *-> TCB of subtask
WAIT ECB=TPISIECB *Wait for subtask initialization
LA R1,TPISTECB *-> Subtask termination ECB
ST R1,0(R6) *Put it into ECB List
LA R6,4(R6) *Next one goes here
BXLE R3,R4,INITSCB *Start them all
S R6,=A(4) *This was last ECB Pointer
OI 0(R6),BIT0 *Close ECB List
DROP R3

*---------------------------------------------------------------------*
* Set up, so we can receive Modify commands from MVS operator *
* We will actually never analyze input, but terminate as soon as *
* the Modify ECB is posted *
*---------------------------------------------------------------------*
LA R2,COMMADDR *-> Communications Area pointer
EXTRACT (R2),FIELDS=COMM
L R2,COMMADDR *-> Communications Area
USING COMMAREA,R2
QEDIT ORIGIN=COMCIBPT,CIBCTR=1 *Only one Modify accepted
L R3,COMECBPBT *-> Modify ECB
ST R3,ECBPMODI *Add to our Wait ECB-list
DROP R2

* We are now ready to get a SOCKET, BIND it to our port and issue *
* a LISTEN command. *
A Beginner's Guide to MVS TCP/IP Socket Programming

*---------------------------------------------------------------------*
MVC TRCMLFUN,=CL8'SOCKET'
EZASMI TYPE=SOCKET, *Get a socket
    AF='INET', *In the INET addressing family
    SOCKET='STREAM', *Of type stream
    ERRNO=ERRNO, C
    RETCODE=RETCODE, C
    ERROR=EZAERROR
ICM R2,15,RETCODE *If Retcode < zero it is
BM EZAERROR *- an error - else it is socket descr
TPITRC 'Socket descriptor from SOCKET Call',
REG=R2 *Trace new socket descriptor
LM R3,R5,TPIMSO TB *Our socket table
USING TPIMSO,R3
SOCKLLOP EQU *
CH R2,TPIMSN0 *This socket descriptor?
BE SOCKLLOK *- Yes this is our listener socket.
BXLE R3,R4,SOCKLLOP *Loop through all
TPILOG MOD=TPIMAIN, *If error here, nothing will work C
MSGNO=10
BXLE R3,R4,CLOSEDWN *Fatal error
SOCKLLOP EQU *
OI TPIMSBIT,TPIMSACT+TPIMSREA+TPIMSLIS
ST R2,TPIMSOCK *Just so we have it.
LH R1,TPIMPORT *Our port number
STH R1,SSTRPORT *Into socket structure for bind
MVC TRCMLFUN,=CL8'BIND'
TPITRC 'Issuing BIND with socket descriptor',
    WORD=TPIMSOCK *Trace entry to Bind
EZASMI TYPE=BIND, *Bind socket to our port
    S=TPIMSOCK, *Our listener socket descriptor
    NAME=SOCSTRUC, *Port and INADDR_ANY IP address
    ERRNO=ERRNO, C
    RETCODE=RETCODE, C
    ERROR=EZAERROR
ICM R2,15,RETCODE *If Retcode < zero it is
BM EZAERROR *- an error
MVC TPIMSSOC,SOCSTRUC *This is listener socket struct.
DROP R3
MVC TRCMLFUN,=CL8'LISTEN'
TPITRC 'Issuing LISTEN with socket descriptor',
    WORD=TPIMSOCK *Trace entry to Listen
EZASMI TYPE=LISTEN, *Issue listen call
    S=TPIMSOCK, *On our listener socket
    BACKLOG=10, *Max 10 in the backlog queue
    ERRNO=ERRNO, C
    RETCODE=RETCODE, C
    ERROR=EZAERROR
ICM R2,15,RETCODE *If Retcode < zero it is
BM EZAERROR *- an error

*---------------------------------------------------------------------*

Here our main loop starts.

* Based on our socket descriptor table, build the three bit strings to*
  * be used in a SELECT call, and issue the SELECT call. *

*---------------------------------------------------------------------*
DOSELECT EQU *
LM R3,R5,TPIMSO TB *BXLE addresses for socket table
USING TPIMSO,R3
XC SELMasks(SELMASKL),SELMasks *Clear them all

A Beginner's Guide to MVS TCP/IP Socket Programming 276
A Beginner's Guide to MVS TCP/IP Socket Programming

SELMSOLP EQU *
TM TPIMSBIT, TPIMRACT *Do we work with it?
BZ SELMSNORX *- No, try next one
TPIMASK SET, *Set Exception bit for all C
MASK=ESNDMASK, *- our active C
SD=TPIMSN0 *- socket descriptors
TM TPIMSBIT, TPIMSRAT *Set read bit?
BZ SELMSNORX *- No, no test for read
TPIMASK SET, *Set read bit for C
MASK=RSNDMASK, *- for our C
SD=TPIMSN0 *- listener socket descriptor

SELMSONR EQU *
TM TPIMSBIT, TPIMSRAT *Set write bit?
BZ SELMSNORX *- No, no test for write
TPIMASK SET, *Set write bit for socket C
MASK=WSNDMASK, *- for our C
SD=TPIMSN0 *- pending error message for it.

SELMSONR EQU *
BXLE R3, R4, SELMSOLP *Loop through all sock descr.
DROP R3
MVC TRCMLFUN, =CL8'SELECT', C
TPITRC 'Issuing SELECT with MAXSOC', C
WORD=TPIMMAXD *Trace entry to select
XC ECBSELE, ECBSELE *Clean up ECB
EZASMI TYPE=SELECT, *Select call C
MAXSOC=TPIMMAXD, *Max. this many descr. to test C
TIMEOUT=SELTIMEO, *One hour timeout value C
RSNDMSK=RSNDMASK, *Read mask C
RRETMASK=RRETMASK, *Returned read mask C
WSNDMSK=WSNDMASK, *Write mask C
WRETMASK=WRETMASK, *Returned write mask C
ESNDMSK=ESNDMASK, *Exception mask C
ERETMASK=ERETMASK, *Returned exception mask C
ECB=ECBSELE, *Post this ECB when activity occurs C
ERNO=ERN0, *- ECB points to an ECB plus 100 C
RETCODE=RETCODE, *- bytes of workarea for socket C
ERROR=EZAERROR *- interface to use.
ICM R2, 15, RETCODE *If Retcode < zero it is
BM EZAERROR *- an error

*---------------------------------------------------------------------*
* *
* Wait for something to happen, which can be one of the following *
* events: *
* 1. SELECT was posted *
* 2. Modify was issued from MVS operator: close down *
* 3. Log Writer Task terminated unexpected: close down *
* 4. A subtask ended prematurely *
* *
*---------------------------------------------------------------------*

DOMWAIT EQU *
WAIT 1, ECBLIST=ECBLIST *Wait for something
L R14, ECBMODI *--> Modify ECB
TM 0(R14), BIT1 *Was modify used?
BZ POSTNMOD *- No, it was not modify
TPILLOG MOD=TPIMAIN, *Message from TPIMAIN C
MSGNO=11 *We are modified to Stop
WTO 'TPIMAIN Modified to STOP - Closing Down'
B CLOSEDWN *Close down the server address space

POSTNMOD EQU *
TM ECBSELE, BIT1 *Was Select posted?
BO SELPOSTE *- Yes, process select
TM ECBTLOGW, BIT1 *Did Logtask terminate?
BZ NOTLOGT *- No, test for server subtasks

TPILog MOD=TPIMAIN, *Message from TPIMAIN
 MSGNO=12 *Log writer task terminated
B CLOSEDWN *Treat as closedown

*---------------------------------------------------------------------*
* *
* Test for terminated server subtask. If a server subtask ended *
* print out the task termination code on the log file. *
* We allow up to 2 times number of subtasks reinstates. *
* If reinstate counter is not exceeded, we attach the server subtasks *
* again, and continue processing. *
* *---------------------------------------------------------------------*

NOTLOGT EQU *
LM R3,R5,TPIMSCBB *BXLE addresses for TPISCBS
USING TPISCB,R3

TSERTERL EQU *
TM TPISTECB,BIT1 *Did Server subtask terminate?
BZ TSERTERN *- No, test next task
L R1,TPISTECB *Termination ECB Contents 00xxxxxx
SLL R1,8 *Remove wait and xxxxxxx00
SRL R1,4 *- post bits 0xxxxxxx0
XC DORD,DORD *Clear work area
ST R1,DORD+4 *Store as lower half of doubleword
UNPK TERMCODE,DORD *Unpack it
NC TERMCODE,=6X'0F' *Remove zones
TR TERMCODE,TRHEX *Translate to text
LA R2,TERMTEXT *-> Termination message
TPILog MOD=TPIMAIN, *Message from TPIMAIN
 TEXT=(R2), *R2 points to 80 character message
 MSGNO=1 *Msgno=1 means prebuilt text
L R1,TPIMREIN *So many reinstates until now
LH R2,TPIMNOST *So many server subtasks
SLL R2,1 *Multiply by two
CR R1,R2 *We allow 2*n'subtask resinstates
BL TSERTREI *- We are under, so do reinstate
TPILog MOD=TPIMAIN, *Message from TPIMAIN
 MSGNO=13 *Reinstate limit is exceeded
B CLOSEDWN *Do a close down

TSERTREI EQU *
LA R1,1(R1) *Increment reinstate count
ST R1,TPIMREIN *Keep track of it..
XC TPISTECB,TPISTECB *Clear ECB
LA R8,TPISTECB *-> Server task Term. ECB
ATTACH EP=TPISERV, *Server subtask main module
 PARAM=((R3)), *Pass TPISCBS as only parameter
 ECB=((R8)) *Termination ECB
ST R1,TPISTCB *-> TCB of subtask
SR R8,R8 *Make ready for double shift
LR R9,R1 *TCB address
SLLD R8,4 *0000000x xxxxxxx0
STM R8,R9,DORD *Store for Unpack
UNPK TPISTCB,E,DORD *Unpack
NC TPISTCBE,=8X'0F' *Remove F's
TR TPISTCB,E,TRHEX *Translate to EBCDIC
WAIT ECB=TPISIECB *Wait for subtask initialization
TPILog MOD=TPIMAIN, *Message from TPIMAIN
 MSGNO=15 *We have reinstated a server task
B DOMWAIT *Go into a new wait on ECBLIST

TSERTERN EQU *
BXLE R3,R4,TSERTERL *Look for server task termination
TPITRC 'Wait completed - no ECBs posted',
A Beginner's Guide to MVS TCP/IP Socket Programming

W=ECBLIST, MOD=TPIMAIN
ABEND 1000, DUMP *
This is a SNO error (!)

* *--------------------------------------------------------------------- *
* Select was posted. *
* *
* First thing we must do is to synchronize our module with the *
* socket interface via the SYNC socket call. Return info from select *
* will be placed in our parameters when we issue the SYNC call (this *
* includes return codes and return masks from select). *
* *
* If SYNC is successful, the RETCODE holds the number of selected *
* socket descriptors. We must remember to process ALL selected *
* socket descriptors; a socket descriptor will only be marked as *
* selected one time for a given activity. *
* *
* *--------------------------------------------------------------------- *

SELPOTSE EQU *
EZASMI TYPE=SYNC, *Synchronize function C
ECB=ECBSEL, *Select ECB plus 100 bytes workarea C
ERRNO=ERRNO, C
RETCODE=RETCODE, C
ERROR=EZAERROR
ICM R15, 15, RETCODE *Was everything OK
BM EZAERROR *- No, some error
ST R15, NOSELCD *Number of sd's selected
TPITRC 'SYNC completed - number of SDs returned', C
W=NOSELCD *Trace number of selected sd's
USING TPIMSO, R3

SPMSOLP EQU *
TPIMASK TEST, *Test a bit C
MASK=RRETMASK, *- in the returned read mask C
SD=TPIMSNO *- for this socket descriptor
BNE SPMSNORD *No read pending on this one
L R15, NOSELCD *Decrement number of
BCTR R15, 0 *- selected socket descriptors
ST R15, NOSELCD *- by one.
TM TPIMSBIT, TPIMSLIS *Is it our listener socket?
BO SPDACC *- Yes, do an accept
TPITRC 'Unexpected read returned', *We only expect read C
H=TPIMSNO *- on our listener socket
B SPECLOSE *Just close it

SPMSNORD EQU *
TPIMASK TEST, *Test a bit in the C
MASK=ERETMASK, *- returned exception mask C
SD=TPIMSNO *- for this socket descriptor
BNE SPMSNEX *No exception pending in this one
L R15, NOSELCD *Decrement number of
BCTR R15, 0 *- selected socket descriptors
ST R15, NOSELCD *- by one.
TM TPIMSBIT, TPIMEXP *Did we expect it?
BO SPECLOSE *- Yes, server has taken socket.
TPITRC 'Unexpected exception returned', *We only expect C
H=TPIMSNO *- exception, when takesocket is OK
B SPECLOSE *For everything else, just close it

SPMSNEX EQU *
TPIMASK TEST, *Test a bit in the C
MASK=WRETMASK, *- returned write mask C
SD=TPIMSNO *- for this socket descriptor
BNE SPMSNXT *No write pending
L R15, NOSELCD *Decrement pending number of
BCTR R15,0 *- selected socket descriptors
ST R15,NOSELCD *- by one.
TM TPIMSBIT,TPIMSWRT *Did we expect it?
BO SPWRITE *- Yes, write out message
TPITRC 'Unexpected Write returned', *We only expect write 
H=TPIMSN0 *- for pending error message
B SPECLOSE *- Close it

SPMSONXT EQU *
BXLE R3,R4,SPMSOLP *Someone must be ready
CLC NOSELCD,=A(0) *Should be zero now
BE DOSELECT *- It is, do new select
TPITRC MSGNO=14, *Not all selected sd's found. This 
MOD=TPIMMAIN *- is an SNO error, but we will 
B DOSELECT *- continue with new select.

*---------------------------------------------------------------------*
* Write selected. *
* Write out pending message to client *
* * 
*---------------------------------------------------------------------*

SPWRITE EQU *
MVC TRCMLFUN,=CL8'WRITE'
TPITRC 'Write no-server message', *Trace write call 
H=TPIMSN0 *- on this socket descriptor
LA R2,TPIMSN0 *Socket descriptor
MVC REQLEN,MSGNOLEN *We want to send full message
XC ACTLEN,ACTLEN *Clean before call
MVC SENDFLAG,=A(SENDDATA) *We want to send the data
MVC TRCMFUN,=CL8'SEND' *For EZAERROR routine
CALL TPISEND,(MAINGLOB, *EZA Global workarea 
MAINTASK, *EZA Task work area 
(R2), *Socket descriptor 
MSGNOSRV, *Output buffer 
REQLEN, *Requested length 
ACTLEN, *Returned actual length 
SENDFLAG, *SEND flag = Send data 
RETCODE, *EZA Retcode 
ERNNO),VL *EZA Error number
LTR R15,R15 *Was send successfull ?
BZ SENDOK *- Yes, buffer has been sent
CH R15,=AL2(4) *Did peer close socket?
BE SPECLOSE *- Yes, we close as well
B EZAERROR *Others means EZA error code

SENDOK EQU *
TPITRC 'Sent so many bytes', *Trace the send call 
W=ACTLEN
B SPECLOSE *And close the socket

*---------------------------------------------------------------------*
* * 
* Close a socket and free socket descriptor entry *
* * 
*---------------------------------------------------------------------*

SPECLOSE EQU *
TPITRC 'Closing down socket', *Trace close call 
H=TPIMSN0 *For this socket descriptor
MVC TRCMFUN,=CL8'CLOSE'
EZASMI TYPE=CLOSE, *Close socket 
S=TPIMSN0, *This socket descriptor 
ERNNO=ERNNO, 
RETCODE=RETCODE, 
ERROR=EZAERROR
ICM R15,15,RETCODE *OK?
A Beginner's Guide to MVS TCP/IP Socket Programming

```
BM  EZAERROR   *- No..
MVI  TPIMSBIT,0  *Clear out in-use flag
B  SPMSONXT  *Test next sd after select post
DROP R3

*---------------------------------------------------------------------*
* *
* Read selected on Listener socket; issue an ACCEPT *
* R3 points to listener socket descriptor entry *
* * *
* * *
* * *
*---------------------------------------------------------------------*

SPDOACC  EQU *
USING TPIMSO,R3
MVC  TRCMLFUN,=CL8'ACCEPT'
TPITRC 'Issuing ACCEPT', *Trace accept call C
H=TPIMSN
EZASMI  TYPE=ACCEPT, *Accept new connection C
S=TPIMSN, *On listener socket descriptor C
NAME=SOCSTRUC, *Returned client socket structure C
ERRNO=ERRNO, C
RETCODE=RETCODE, C
ERROR=EZAERROR
ICM  R2,15,RETCODE  *OK?
BM  EZAERROR  *- No, error indicated
TPITRC 'ACCEPT returned new socket descriptor', *Trace C
REG=R2  *- new socket descriptor
LR  R15,R2  *We need it later
MH  R2,=AL2(TPIMSOLN) *Offset into socket table
A  R2,TPIMSO
DROP R3  *Drop listener socket descr. base
USING TPIMSO,R2  *Base for the new descriptor no.
XC  TPIMSO(TPIMSOLN),TPIMSO *Clear entry
STH  R15,TPIMSN  *Descriptor number
OI  TPIMSBIT,TPIMSWRT  *Socket descriptor temp. active
MVC  TPIMSSOC,SOCSTRUC  *Socket structure

*---------------------------------------------------------------------*
* *
* Find an available server subtask, issue a GIVESOCKET *
* and POST server task. *
* *
* *
* *
* If no server subtask is available, we mark the new socket *
* descriptor for write pending and includes it in a new select. *
* When write is selected, we write out an error message about no *
* available server. *
* *
* * *
* * *
*---------------------------------------------------------------------*

LM  R7,R9,TPIMSCBB  *Subtask BXLE addresses
USING TPISCB,R7
ACCSUBLP  EQU *
TM  TPISECB,BIT0  *Is this one waiting for work?
BO  ACCFREST  *- Yes, we found a free server task
BXLE  R7,R8,ACCSUBLP  *Look through them all
MVC  TPIMSEN0,=AL2(1)  *Indicate no server
OI  TPIMSBIT,TPIMSWRT  *We want to write
B  SPMSONXT  *Test next sd after select

ACCFREST  EQU *
MVC  CLNNAME,TPIMCNAM  *Our Client ID Address Space Name
MVC  CLNTASK,TPISTCBE  *To this subtask
L  R15,CLNFAM  *Addressing Family
CVD  R15,DORD  *From binary to decimal
OI  DORD+7,X'0F'  *A nice sign.
UNPK  GIVLOGAF,DORD  *Into logging line
LH  R15,TPIMSN0  *Socket descr. to give

A Beginner's Guide to MVS TCP/IP Socket Programming 281
```
From binary to decimal

A nice sign.

Into logging line

*Address Space Name

*Subtask Name

*Log client id to give sd to C

*Text is prebuilt C

Main is logging the message

Givesocket C

Give this socket descriptor C

* to an available server task C

C

ERROR=CZAERROR

OK ?

-- No, tell about it.

Main task sockdesr. for takesocket

Wake up server task

We expect an except. aft. takesock.

Drop work base register

- to an available server task C

Back to listener socket descriptor

Test next socket after select

Here we come if non-successfull EZASMI macro call

Write out message to log file, and terminate.

--------------------------------------------------------------------------------

EZAERROR EQU *

TPILog MOD=TPIMAIN, *TPIMAIN is logging message C

FUNC=TRCMLFUN, *This socket function C

ERRNO=ERRNO, *Socket error number C

RETCODE=RETCODE, *Socket return code C

MSGNO=0 *Construct socket error message

*--------------------------------------------------------------------------------

Closedown

Try to post server subtask for orderly shutdown - followed

by detach.

Msgno=999 instructs the log writer task to close its logfile

DCB and to terminate. Allow both server subtasks and log writer

task time to do proper termination.

--------------------------------------------------------------------------------

CLOSEDWN EQU *

IM R3,R5,TPIMSCBB *SCB bxle

USING TPISCB,R3

CLSLOOP EQU *

CLC TPISTCB,=4X'00' *Is the task attached ?

BE CLNSNODET *-- No, so do not detach it.

TM TPISECB,BIT0 *Is it waiting for work?

BO CLSSWAIT *-- Yes, ask it to terminate.

STIMER WAIT, *Wait 500 msec for C

BINTVL=MSEC500 *-- it to finish work.

B CLSDET *-- and then just detach it.

CLSSWAIT EQU *

POST TPISECB,4 *Post with RC=4 for terminate

STIMER WAIT, *Wait 500 msec for C

BINTVL=MSEC500 *-- it to terminate
A Beginner's Guide to MVS TCP/IP Socket Programming

CLSDET EQU *        /* and then just detach it.
  LA  R2, TPISTCB    /*-> Server TCB address
  DETACH (R2)       /*Just go away now...

CLSNODET EQU *
  BXLE R3, R4, CLSLOOP *Kill them all
  DROP R3
  TPILOG MOD=TPIMAIN,
     MSGNO=999 /*- log file and terminate
    STIMER WAIT,
       BINTVL=MSEC500 /*- logfile and terminate
  LA  R2, TPIMLTCB  /*- Log writer TCB address
  DETACH (R2)      /*Kill Log Writer task
  EZASMI TYPE=TERMAPI /*Terminate socket API
  TERM RC=0          /*And out we go
  LTORG

*---------------------------------------------------------------------*
* Select masks used by the socket select call and select control *
* variables
*---------------------------------------------------------------------*

DS 0F
  DC CL16 'SELECT MASKS' *Eyecatcher

SELMASKS DS 0F
  RSNDMASK DC XL8 '00000000' *Read mask
  RRETMASK DC XL8 '00000000' *Returned read mask
  WSNDMASC DC XL8 '00000000' *Write mask
  WRETMASK DC XL8 '00000000' *Returned write mask
  ESNDMASC DC XL8 '00000000' *Exception mask
  ERETMASK DC XL8 '00000000' *Returned exception mask

SELMASKL EQU *-SELMASKS

NOSELCD DC A(0)     /*Keep track of selected sd's
SELTIMEO DC A(3600, 0) /*One hour timeout
ECBSELE DC A(0)      /*Select ECB
  DC 100X '00'        /*Required by EZASMI !!

*---------------------------------------------------------------------*
* No available server error message
*---------------------------------------------------------------------*

MSGNOSRV DC C'B', C'TPI', C'0007'
  DC CL80 'No server is currently available - try again later'

MSGNOLEN DC A(*-MSGNOSRV) /*L'message

*---------------------------------------------------------------------*
* Socket interface variables and structures
*---------------------------------------------------------------------*

TRCMLFUN DC CL8 ' '    /*Current socket function
  ERRNO DC A(0)         /*Errorno from EZASMI
  RETCODE DC A(0)       /*Returncode from EZASMI

IDENTSTR DS 0F         /*INITAPI: Ident structure
  IDENTTCP DC CL8 ' '   /*TCP/IP Address space name
  IDENTJOB DC CL8 ' '   /*My Address space name

SOCSTRUC DS 0F         /*BIND and ACCEPT: Socket structure
  SSTRFAM DC AL2(2)    /*TCP/IP Addressing family
  SSTRPORT DC AL2(0)   /*Port number
  SSTRADDR DC AL4(0)   /*IP Address (=X'00' is INADDR_ANY)
  SSTRRESV DC 8X '00'  /*Reserved
* CLNSTRUC DS OF *GIVESOCKET: Client structure
CLNFAM DC A(2) *TCP/IP Addressing family
CLNNAME DC CL8' ' *Address space name of target
CLNTASK DC CL8' ' *Subtask id of target
CLNRESV DC XL20'00' *Reserved
*---------------------------------------------------------------------*
* TPIRECV and TPISEND Communication fields *
*---------------------------------------------------------------------*
REQLEN DC A(0) *Requested receive/send length
ACTLEN DC A(0) *Actually received/sent length
SENDFLAG DC A(0) *SEND flags
SENDDATA EQU 0 *Send data
*---------------------------------------------------------------------*
* Main task Control Block *
*---------------------------------------------------------------------*
TPIMCB TPIMCB TYPE=CSECT *Main Control Block
*---------------------------------------------------------------------*
* ECBlist for wait *
*---------------------------------------------------------------------*
ECBLIST DS OF
ECBPMODI DC A(0) *--> Modify ECB
DC A(ECBSELE) *--> Select ECB
DC A(ECBTLOGW) *--> Logwriter termination ECB
ECBPSTS DC 10A(0) *Max 10 subtasks term. ECB's
*---------------------------------------------------------------------*
* Subtask termination log message *
*---------------------------------------------------------------------*
TERMTEXT DC 0CL80' '
TERMCODE DC CL6' ',CL1' ' *Completion code
DC CL(80-(**-TERMTEXT))'Subtask prematurely terminated'
*---------------------------------------------------------------------*
* Logging line for client id *
*---------------------------------------------------------------------*
CLNLOGLN DC 0CL80' '
DC C'TPIMAIN Client ID ' 
DC C'Family=' 
CLNLOGAF DC CL4' ',CL1' ' *Addressing Family 
DC C'Address Space=' 
CLNLOGAS DC CL8' ',CL1' ' *Address Space Name 
DC C'Subtask=' 
CLNLOGST DC CL8' ' *Subtask Name 
DC CL(80-(**-CLNLOGLN))' ' 
*---------------------------------------------------------------------*
* Logging line client id on givesocket *
*---------------------------------------------------------------------*
GIVLOGLN DC 0CL80' '
DC C'Givesocket SD=' 
GIVLOGSD DC CL4' ',CL1' ' 
DC C'Family='
**H.1.2 TPILOGWT Logwriter Data Services Task**

* Name: TPILOGWT

* Function: Log writer Data Services Task in the TPI server application.

* Interface: R1 -> parameter list with one pointer:
  * +0 -> TPI Main task Control Block, which holds the parameters required by TPILOGWT to write out a message to the log file on DD stmt TPILOG.

* Logic: This program executes as an independent subtask attached by TPIMAIN as part of initialization. During startup, it will open a DCB for the TPILOG dataset. TPIMAIN waits for TPILOGWT to initialize on TPIMLDON ECB in the Main task Control Block, which TPILOGWT posts when initialization is done. Requests to print a message are initiated from other tasks via the TPILOG macro. Combined processing of TPILOG Macro and TPILOGWT is:

1. TPILOG macro enqueues on TPI, TPILOGWT to serialize use of the Log Writer interface fields in the Main task Control Block.
2. When TPILOG macro gets its enqueue, it builds TPILOGWT parameters in the Main task Control Block.
3. TPILOG macro posts TPIMLECB, which TPILOGWT is waiting on for work.
4. TPILOG macro then issues a wait on TPIMLDON.
5. TPILOGWT wakes up and processes the log request writing a message to log file.
6. When TPILOGWT has finished processing this request it posts TPIMLDON, which TPILOG macro is waiting on. TPILOGWT then issues a new wait on TPIMLECB - waiting for a new log request.
7. The TPILOG macro dequeues from TPI, TPILOGWT and exits from the macro expansion code.

If TPILOGWT receives a message number of 999, it will close the log file DCB and terminate.

Abends: - none -
Returncode: - none -
Written: May 28'th 1994 at ITSO Raleigh

Instream macro for formatting numbers

MACRO
FORMNUM &FROM,&TO
L R15,&FROM
CVD R15,DWORD
OI DWORD+7,X'0F'
UNPK &TO.,DWORD
MEND

Instream macroe for generating message table entry

MACRO
MSG &NO,&TEXT
DC A(&NO.),CL80&TEXT.
MEND

TPIMCB  TPIMCB TYPE=DSECT *Main task Control Block
TPILOGWT INIT 'Log data set writer task',MODE=24
* Initialize - open DCB and put out greeting message on log file.

L R9,0(R1)  *-> Main task Control Block (TPIMCB)
USING TPIMCB,R9  *Address it
OPEN (TPILOG,(OUTPUT))  *Open our log writer DCB
PUT TPILOG,HD1  *Print our header line

* For each request, post back when done - as requester is waiting for us to complete work on TPIMLDON in the main task Control
* Then issue a wait on TPIMLECB also in the Main task Control Block * 
* for a new work request. * 
* *

```
WAITWORK EQU *
POST TPIMLDON,0 *Tell requester, we are done.
XC TPIMLECB,TPIMLECB *Clear our ECB
WAIT ECB=TPIMLECB *Wait for next work request
CLC TPIMLMNO,=A(999) *Means close down
BE GETOUT *- So just close DCB and exit.
TIME DEC, *Let us see the current time C
TIMENOW, *- when we were woken up C
LINKAGE=SYSTEM
```

* Build fixed part of each trace line: timestamp, module and message number *

```
MVI LIN,C' ' *Initialize output line
MVC LIN+1(L'LIN-1),LIN *- with spaces
MVC EDWORK,EDMASK *Time edit mask
LM R2,R3,TIMENOW *hhmmssth xxxx0000
SRDL R2,28 *Shift out so 00000h hhmssthx
STM R2,R3,DWORD *Treat as decimal
OI DWORD+7,X'0F' *Put in a sign
ED EDWORK,DWORD+3 *Edit time as hh:mm:ss.th
MVC TIMESTMP,EDWORK+1 *Time to output line
MVC MODULE,TPIMLMOD *Name of calling module
FORMNUM TPIMLMNO,MSGNO *Format message number to line
```

* Do message code specific processing: *
* Msgno = 0 means we must format Socket interface return info *
* Msgno = 1 means that a prebuilt text string has been passed *
* as message text, and we just put it out *
* *
* For all other message numbers passed, a corresponding text is *
* found in the message table, which is part of this module. *

```
L R2,TPIMLMNO *Passed message number
LTR R2,R2 *MSGNO=0 means EZASMI returninfo
BZ EZAINFO *Go and format socket return info
C R2,=A(1) *MSGNO=1 means passed text string
BE TEXTOUT *Just put out the passed string
LM R3,R5,MSGTABBX *Search message table
```

```
MSGLOOP EQU *
CLC TPIMLMNO,0(R3) *This message ?
BE FOUNDMSG *- Yes, print it
BXLE R3,R4,MSGLOOP *Look through them all
LA R3,DUMMYMSG *If not found, use a dummy text
```

```
FOUNDMSG EQU *
MVC TEXT,4(R3) *Print table message text
PUT TPILOG,LIN *Print it
B WAITWORK *Wait for next request
```

```
TEXTOUT EQU *
MVC TEXT,TPIMLTEXT *Print passed text string
PUT TPILOG,LIN *Print it
B WAITWORK *Wait for next request
```
EZAINF O EQU *
MVC EZAFUN,C'EZASMI Function='
MVC EZAERR,C'ErrNo='
MVC EZARET,C'RetCode='
MVC EZAFUNCD,TPIMLFUN *Socket function
FORMNUM TPIMLERR,EZAERRNO *Socket error number
FORMNUM TPIMLRET,EZARETCD *Socket return code
PUT TPILOG,LIN *Print it
B WAITWORK *Wait for next request

*---------------------------------------------------------------------*
* When we receive msgno=999, we close the log file DCB and *
* terminate *
*---------------------------------------------------------------------*
GETOUT EQU *
CLOSE (TPILOG) *Close log file DCB
POST TPIMLDON,0 *Tell requester, we are done.
TERM RC=0 *No reason for anything else.
LTORG

*---------------------------------------------------------------------*
* Formatting work fields *
*---------------------------------------------------------------------*
EDWORK DC CL12'
EDMASK DC XL12'2120207A20207A20204B2020'
DWORD DC D'0'
TIMENOW DC XL16'00'

*---------------------------------------------------------------------*
* Header line and detail line layout *
*---------------------------------------------------------------------*
HD1 DC CL130'ITPI Log Writer Task has started'
LIN DS 0CL130'
DC C'
TIMESTAMP DC CL11',,CL1'
MODULE DC CL8',,CL1'
MSGNO DC CL3',,CL1'
EZAFUN DC C'EZASMI Function='
EZAFUNCD DC CL8',,CL1'
EZAERR DC C'ErrNo='
EZAERRNO DC CL5',,CL1'
EZARET DC C'RetCode='
EZARETCD DC CL4',,CL1'
ORG EZAFUN
TEXT DC CL80'
DC CL(130-(*-LIN))'

*---------------------------------------------------------------------*
* Log file DCB *
*---------------------------------------------------------------------*
TPILOG DCB DDNAME=TPILOG,MACRF=(PM),RECFM=FBA,LRECL=130,
C DSORG=PS,BLKSIZE=1300

* Message text table - key is message number *

A Beginner's Guide to MVS TCP/IP Socket Programming
H.1.3 TPISERV Concurrent Server Subtask

*******************************************************************************
*                                                                          *
* Name:          TPISERV                                                   *
*                                                                          *
* Function:      This module is the main module in each TPI server subtask  *
*                                                                          *
* Interface:     R1 -> parameter list with one pointer:                    *
*                +0 -> TPISCB TPI Server task Control Block                 *
*                Pointers to the TPI Main task Control Block and           *
*                to the socket global workarea are picked up from          *
*                the Server task Control Block.                             *
*                                                                          *
* Logic:         This module receives control as the main module when      *
*                the main task issue an Attach to start a new server subtask.*
*                1. Pointers to the Main task Control Block and to the     *
*                EZA Global Workarea are established.                       *
*                2. Pointer to the task level EZA Work Area is set up.       *
*                3. When the task has finished initialization, it posts    *
*                TPISIECB in the Server task Control Block, which          *
*                the main task is waiting on.                               *
*                4. The module then enters a loop, where it waits for      *
*                work on TPISIECB in the Server task Control Block, which    *
*                will be posted by the main task, when a new connection      *
*                arrives from the network.                                  *
*                If the main task posts with an RC=0 it means work.         *
*                If the main task posts with an RC=4 it means shutdown.      *
*                5. When work arrives, a Takesocket is issued to take      *
*                the socket given by the main task. The main task passes    *
*                the socket descriptor in the Server task Control Block and *
*                the main task client id in the Main task Control Block.      *
*                6. DB2 connection is established and a DB2 plan is opened.   *
*                7. Data is received over the socket interface into a       *
*                buffer.                                                    *
*                8. The buffer is passed to TPISERVD, which will do the      *
*                required processing on it.                                  *
*                9. When control returns from TPISERVD, the input buffer     *
*                has been replaced by an output buffer, which is sent over   *
*                the socket interface.                                      *
*                10. The socket is closed.                                    *
*                11. The connection with DB2 is closed.                        *
*******************************************************************************
12. Processing continues at item no. 4 above.

Abends: - none -

Returncode: - none -

Written: May 28'th 1994 at ITSO Raleigh

Modified:

***********************************************************************
PRINT GEN
TPIMCB TPIMCB TYPE=DSECT *Main task Control block dsect
TPISCB TPISCB TYPE=DSECT *Server task Control Block dsect
TPIREC TPIREC TYPE=DSECT *TPI input and output record dsect
PRINT NOGEN
EZAGLOB EZASMI TYPE=GLOBAL, *EZA Global work Area dsect C
STORAGE=DSECT

TPISERV INIT 'TPI Server task',RENT=NO,MODE=24,BASE=(12,11)

* Pointer for Server task Control Block (TPISCB) is passed from *
* the main task on the attach call. *
* Addressability to the Main task Control Block (TPIMCB) and the *
* main task EZA Global Work Area is established. *
* Addressability to task level EZA Work area is established. *
* *
* Subtask client ID is built, and a socket INITAPI call *
* is issued. *
* *
* Main task waits for server subtask to initialize on TPISIECB. *
* *
*----------------------------------------------------------------------
L R10,0(R1) *-> Server task Control Block
USING TPISCB,R10 *Server task Control Block
L R9,TPISMCB *-> Main task Control Block
USING TPIMCB,R9 *Main task Control Block
TPITRC TYPE=INIT, *Enable trace points C
MOD=TPISERV, *Tracing module is TPISERV C
TRACE=YES
TPITRC 'TPISERV entered', C
REG=R10 *Address of TPISCB
L R8,TPIMGLOB *-> EZA Global work area
USING EZAGLOB,R8 *EZA Global work area
LA R1,EZATASK *-> EZA task work area
ST R1,TPISTASK *Just so we have it.
L R3,X'10' *-> CVT
L R3,0(R3) *-> TCB Words
L R3,4(R3) *-> Current TCB (My TCB)
SR R2,R2 *Make ready for double shift
SLDL R2,4 *0000000x x0000000
STM R2,R3,DORD *Store for Unpack
UNPK TPISTCBE,DORD *Unpack
NC TPISTCBE,=8X'0F' *Remove F's
TR TPISTCBE,TRHEX *Translate to EBCDIC
MVC TRCLFUN,=CL8'INITAPI'
MVC IAPITCP,TPIMTCPI *TCP/IP address space name
MVC IAPIAS,TPIMCNAM *Our address space name
EZASMI TYPE=INITAPI, *Initialize socket API C
MAXSOC=IAPISOC, *This many sockets C
SUBTASK=TPISTCBE, *My TCB address in EBCDIC C
IDENT=IAPIIDEN, *TCP/IP AS name and my AS name C
MAXSNO=IAPISNO, *This many socket descriptors C
ERNO=ERNO, C
RETCODE=RETCODE

ICM R15,15,RETCODE *Did we do well ?
BM EZAERROR *- No, deal with it.
MVC TRCMLFUN,=CL8 'GETCLNID'
EZASMI TYPE=GETCLIENTID, *Get our own client id C
CLIENT=TPISCLNI, *Store it in Server task Control B. C
ERNO=ERNO, C
RETCODE=RETCODE, C
ERROR=EZAERROR

ICM R15,15,RETCODE *Was it OK
BM EZAERROR *- No, stop now.
L R15,TPISCDOM *Addressing Family
CVD R15,DORD *From binary to decimal
OI DORD+7,X'0F' *A nice sign.
UNPK CLNLOGAF,DORD *Into logging line
MVC CLNLOGAS,TPISCNAM *Address Space Name
MVC CLNLOGST,TPISCTSK *Subtask Name
TPILOG TEXT=CLNLOGLN, *Log client id C
MSGNO=1, *Text is prebuilt C
MOD=TPISERV *Main is logging the message
POST TPISIECB,0 *OK, we have initialized

*---------------------------------------------------------------------*
* *
* Wait-for-Work loop starts here. Main task will post TPISECB, *
* when there is work to be done. *
* *
* RC=0 means work to do.
* RC=4 means shutdown.
* *
*---------------------------------------------------------------------*

WAITLOOP EQU *

TPITRC 'TPISERV Going to sleep.', C
REG=R10 *Address of TPISCB
XC TPISECB,TPISECB *Clean up
WAIT ECB=TPISECB *Wait for work
TPITRC 'TPISERV Woke up', C
W=TPISECB *ECB in trace
L R2,TPISECB *Let us see the RC
SLL R2,8 *Get rid of
SRL R2,8 *- post and wait bits
LTR R2,R2 *RC=0 means work
BNZ GETOUT *Anything else means shutdown

*---------------------------------------------------------------------*
* *
* Main task socket descriptor is passed in TPISCB as TPISSOD. *
* Main task client id is in TPIMCB as TPIMCLNI. *
* On Takesocket, we must point to the socket descriptor and the *
* client id from the task that issued Givesocket. *
* *
* Takesocket returns a new socket descriptor, which will be used *
* in this subtask for further communication. *
* *
*---------------------------------------------------------------------*

MVC TRCMLFUN,=CL8 'TAKESOCK'
TPITRC 'Takesocket With old descriptor', C
H=TPISSOD *Trace old socket descr.
EZASMI TYPE=TAKESOCKET, *Takesocket C
CLIENT=TPIMCLNI, *Main task client id structure C
SOCRECV=TPISSOD, *Main task socket descriptor C
ERRNO=ERRNO,  
RETCODE=RETCODE,  
ERROR=EZAERROR  
ICH R15,15,RETCODE *Did we do well ?  
BM EZAERROR -*- No, deal with it.  
STH R15,TPISNSOD *Server task socket descr.no  
TPITRC 'Takesocket returned new descriptor',  
REG=R15 *Trace new socket descriptor

*---------------------------------------------------------------------*
* * Issue a Getpeername call to obtain socket address structure of  *
* client. Format and print it on log file.  *
* *---------------------------------------------------------------------*

MVC TRCMFUN,=CL8'GETPEERN'  
EZASMI TYPE=GETPEERNAME, *Getpeername  
S=TPISNSOD, *Of connected client  
NAME=PEERNAME, *Return socket address struc here  
ERRNO=ERRNO,  
RETCODE=RETCODE,  
ERROR=EZAERROR  
ICH R15,15,RETCODE *Did we do well ?  
BM EZAERROR -*- No, deal with it.  
LH R2,PEERFAM *Addressing family of peer  
CVD R2,DORD *To decimal  
OI DORD+7,X'0F' *Put in sign  
UNPK PERLOGAF,DORD *Into logging line  
LH R2,PEERPORT *Port number of peer  
CVD R2,DORD *To decimal  
OI DORD+7,X'0F' *Put in sign  
UNPK PERLOGPO,DORD *Into logging line  
CALL TPIINTOA,(PEERIP, *Convert from 4 bytes network order  
PERLOGIP),VL *- to 15 char text.  
TPILOG TEXT=PERLOGLN, *Log peers socket address  
MSGNO=1, *Text is prebuilt  
MOD=TPISERV *From TPISERV

*---------------------------------------------------------------------*
* * Open our DB2 plan, and let CAF issue an implicit DB2 connection.  *
* The DB2 subsystem id is picked up from the Main task Control Block.  *
* *---------------------------------------------------------------------*

MVC CAFFUNC,=CL12'OPEN' *Open a PLAN  
MVC CAFSSNM,TPIMDB2 *DB2 subsystem name  
MVC CAFPLAN,=CL8'TPISERV' *DB2 plan name  
CALL DSNALI,(CAFFUNC, *Function=OPEN  
CAFSSNM, *Subsystem name  
CAFPLAN, *Plan name=TPISERV  
CAFRC, *CAF Return code  
CAFREAS),VL *CAF Reason code  
CLC CAFRC,=A(0) *Was it OK?  
BE CAFOPNOK -*- Yes, we have a connection  
TPITRC 'CAF Open Return Code', *Trace the bad  
W=CAFRC -*- CAF Return code  
TPITRC 'CAF Open Reason Code', *Trace the bad  
W=CAFREAS -*- CAF Reason code  
B CLOSEROK *And give up this socket.

*---------------------------------------------------------------------*
* * Use TPIRECV for the actual socket RECV call.  *
* * Start with a peek at the first 5 bytes.  *
The first byte in the received data is a record code we use to decide how many bytes we must read to have a full record. The next 4 bytes is used to decide whether the received data is ASCII or EBCDIC. The fixed text in our application is 4 bytes with the value 'TPI'.

Based on the decision about the number of bytes and bytes read, we call TPIRECV again for an actual read of the number of bytes we now know should be there.

CAFOPNOK EQU *
LA R6,BUFFER *Begin to read into
LA R5,TPISNSOD *Socket descriptor
MVC REQLEN,=A(5) *We want to see first 5 bytes
MVC RECVFLAG,=A(RECVPEEK) *We just want to peek.
MVC TRCMLFUN,=CL8'RECV' *For EZAERROR routine
CALL TPIRECV,((R8), *EZA Global workarea C
(EZATASK, *EZA Task work area C
(R5), *Socket descriptor C
(R6), *Input buffer C
REQLEN, *Requested length C
ACTLEN, *Returned actual length C
RECVFLAG, *RECV flag = Peek at data C
RETCODE, *EZA Retcode C
ERNNO),VL *EZA Error number
LTR R15,R15 *Successfull ?
BZ PEEKOK *- Yes, buffer has first 5 bytes
CH R15,=AL2(4) *Did peer close socket?
BE CLOSESOK *- Yes, we close as well
B EZAERROR *Others means EZA error code

PEEKOK EQU *
TPITRC 'Peek returned so many bytes', C
W=ACTLEN
LM R1,R3,RECIDBXL *BXLE for record IDs

READFID EQU *
CLC 0(1,R1),BUFFER *First byte is record ID
BE GOTANID *This is it
BXLE R1,R2,READFID *If not found - error message back:
LA R6,BUFFER *- Input buffer
USING TPIREC,R6 *Let us see if it is ascii/ebcdic
MVI TPISCTYP,TPISEBCD *Default client is EBCDIC
CLC IIDENT,=CL4'TPI' *Is client in EBCDIC?
BE RESP3EBC *- Yes, flag is correct: EBCDIC
MVI TPISCTYP,TPISASCII *- No, set client flag: ASCII

RESP3EBC EQU *
MVI IRECID,IRESP *Build error response with
MVC ICODE,=CL4'0003' *- errorcode = 0003 (invalid recid)
B PREPSEND *Go and send it.
DROP R6 *Was only temporary for error 0003.

GOTANID EQU *
LH R2,1(R1) *Length of this record type
ST R2,REQLEN *So long is pending message
XC ACTLEN,ACTLEN *Just clean it before call
LA R6,BUFFER *- Input data
LA R5,TPISNSOD *Socket descriptor
MVC RECVFLAG,=A(RECVREAD) *We want to read the data now
MVC TRCMLFUN,=CL8'RECV' *For EZAERROR routine
CALL TPIRECV,((R8), *EZA Global workarea C
(EZATASK, *EZA Task work area C
(R5), *Socket descriptor C
(R6),   *Input buffer
REQLEN,  *Requested length  C
ACTLEN,  *Returned actual length  C
RECVFLAG,  *RECV flag = Peek at data  C
RETCODE,  *EZA Retcode  C
ERRNO),VL  *EZA Error number
LTR R15,R15  *Successful ?
BZ READOK  -* Yes, buffer has full message
CH R15,=AL2(4)  *Did peer close socket?
BE CLOSESOK  -* Yes, we close as well
B EZAERROR  *Others mean EZA error code

*---------------------------------------------------------------------*
*                                                                   *
* If input data is in ASCII, we translate the whole string into     *
* EBCDIC, and set a switch so we remember to translate output       *
* data from EBCDIC to ASCII before we send it.                      *
*                                                                   *
* The buffer is then passed to TPISERVD, which will do whatever      *
* processing is needed and build output data in the buffer area.     *
*                                                                   *
*---------------------------------------------------------------------*

READOK EQU *
USING TPIREC,R6  *Let us work on it.
TPITRC 'Receive returned so many bytes',  C
W=ACTLEN
MVI TPISCTYP,TPISEBCD  *Default client is EBCDIC
CLC IIDENT,=CL4'TPI'  *Do we need ASCII translate
BE RECINEBC  -* No, it is in EBCDIC
CALL EZACIC05,((R6),  *Translate from ASCII to C
ACTLEN),VL  *EBCDIC
MVI TPISCTYP,TPISASCI  *Client is ASCII

RECINEBC EQU *
CALL TPISERVD,((R10),  *-> Server task Control Block C
BUFFER,  *Input buffer  C
RECLEN),VL  *L'input buffer

*---------------------------------------------------------------------*
*                                                                   *
* On return from TPISERVD, the buffer holds a partly completed      *
* output record.                                                   *
* If the request to TPISERVD was to fetch an existing DB2 record,   *
* the buffer is complete and we need find out just how many bytes   *
* to send to the client.                                           *
*                                                                   *
* If output is a message indicating successful or unsuccessful     *
* processing, we find a suitable message text to pass back.        *
*                                                                   *
*---------------------------------------------------------------------*

PREPSEND EQU *
LM R1,R3,RECIDBXL  *BXLE for record IDs
SENDID EQU *
CLC IRECID,0(R1)  *First byte is record ID
BE SENDID  *This is it
BXLE R1,R2,SENDID  *Look for it
MVC ICODE,=CL4'0003'  *Invalid record id (err in TPISERVD)
MVI IRECID,IRESP  *Response
B PREPSEND  *Redrive BXLE

SENDID EQU *
LH R2,1(R1)  *Length of this record type
ST R2,RECLLEN  *So long is current record
CLI IRECID,IRQRESP  *Is it a query response record?
BE SENDNRS  -* Yes, buffer is complete.
CLC ICODE,=CL4'0006'  *SQL Error message is complete.
BE SENDNRSP *Do not modify DSNTIAR text
LM R1,R3,MSGBXLE *BXLE addresses for msgtext

SENDMLOP EQU *
CLC ICODE,0(R1) *This message code?
BE SENDMFND *- Yes, message found
BXLE R1,R2,SENDMLOP *Look through them all
MVC IMESSAGE,=CL80'No message text found'
B SENDNRSP *Default has been set

SENDMFND EQU *
MVC IMESSAGE,4(R1) *Return this message

*---------------------------------------------------------------------*
* If received data was ASCII, the client most likely wants the *
* response in ASCII again. *
*---------------------------------------------------------------------*

SENDNRSP EQU *
TM TPISCTYP,TPISASCI *Is Client ASCII ?
BZ SENDIT *- No, just send data
CALL EZACICO4,(BUFFER, CRECLEN),VL *- to ASCII

*---------------------------------------------------------------------*
* Send data to client, close socket, close DB2 connection and go *
* and wait for more work. *
*---------------------------------------------------------------------*

SENDIT EQU *
LA R5,TPISNSOD *Socket descriptor
MVC REQLEN,RECLEN *We want to send full message
XC ACTLEN,ACTLEN *Clean before call
MVC SENDFLAG,A(SENDDATA) *We want to send the data
MVC TRCMLFUN,=CL8'SEND' *For EZAERROR routine
CALL TPISSEND,((R8), C)EZATASK, *EZA Global workarea
R5,TPISNSOD *Socket descriptor
BUFFER, *Output buffer
REQLEN, *Requested length
ACTLEN, *Returned actual length
SENDFLAG, *SEND flag = Send data
RETCODE, *EZA Retcode
ERRNO),VL *EZA Error number

LTR R15,R15 *Was send successfull ?
BZ SENDOK *- Yes, buffer has been sent
CH R15,=AL2(4) *Did peer close socket?
BE CLOSEOK *- Yes, we close as well
B EZAERROR *Others means EZA error code

SENDOK EQU *
TPITRC 'Sent so many bytes', *Trace the send call
REG=ACTLEN

CLOSESOK EQU *
MVC TRCMLFUN,=CL8'CLOSE'
EZASMI TYPE=CLOSE, *Close the socket
S=TPISNSOD, *Subtask socket descriptor
ERRNO=ERRNO, C
RETCODE=RETCODE, C
ERROR=EZAERROR

ICM R15,15,RETCODE *Was close socket done ?
BM EZAERROR *- No, some error
TPITRC 'Close done', *Trace the close call
REG=R15

MVC CAFFUNC,=CL12'CLOSE'
CALL DSNALI, (CAFFUNC, *Function=CLOSE C
CAFTERMO, *Termination options: Commit C
CAFRC, *CAF Return code C
CAFREAS), VL *CAF Reason code C
CLC CAFRC, =A(0) *Was CAF Close OK ? C
BE WAITLOOP *- Yes, wait for more work C
TPITRC 'CAF Close Return Code', *Trace the bad C
W=CAFRC *- CAF Return code C
TPITRC 'CAF Close Reason Code', *Trace the bad C
W=CAFREAS *- CAF Reason code C
B WAITLOOP *Wait for work C
*---------------------------------------------------------------------*
* If we receive an unexpected error code from the socket interface, *
* we write out diagnostic info to the log data set and close the *
* socket before we go and wait for new work. *
*---------------------------------------------------------------------*
EZAERROR EQU *
EZAERROR EQU *
TPILOG MOD=TPISERV, *TPISERV module is logging C
FUNC=TRCMGLFUN, *This was the socket function C
ERNRE=ERNRE, *- that gave this error code C
RETCODE=RETCODE, *- with this retcode. C
MSNO=0 *No message passed - build it. C
EZASMI TYPE=CLOSE, *Close the socket C
S=TPISNSOD, *Subtask socket descriptor C
ERNRE=ERNRE, *We really do not care about C
RETCODE=RETCODE *- these, but for the sake of it. C
B WAITLOOP *Just wait for another client C
*---------------------------------------------------------------------*
* Terminate subtask. *
*---------------------------------------------------------------------*
GETOUT EQU *
EZASMI TYPE=TERMAPI *Terminate socket API C
TPITRC 'TPISERV is shutting down', C
W=TPISECB *Trace the ECB C
TERM RC=0 *No reason for anything else C
LTORG C
TRHEX DC C'0123456789ABCDEF' *Hex translate table C
*---------------------------------------------------------------------*
* CAF Call Attachment Facility interface parameters *
*---------------------------------------------------------------------*
CAFFUNC DC CL12' ' *CAF Function code C
CAFSSNM DC CL4' ' *DB2 subsystem name C
CAFPLAN DC CL8' ' *DB2 Plan name C
CAFRC DC A(0) *CAF Return code C
CAFREAS DC A(0) *CAF Reason code C
CAFTERMO DC CL4'SYNC' *CAF Termination option C
*---------------------------------------------------------------------*
* Initapi call parameters *
*---------------------------------------------------------------------*
IAPISOC DC AL2(10) *Max socc C
IAPIIEN DS 0C C
IAPITCP DC CL8' ' *TCP/IP Address space name C
IAPIAS DC CL8' ' *My address space name C
IAPISNO DC AL4(10) *Max sno
IAPITYPE DC AL2(2) *Api type

* Getpeername call parameters *

PEERNAME DS 0C *Returned socket address structure
PEERFAM DC AL2(0) *Addressing family
PEERPORT DC AL2(0) *Port number
PEERIP DC AL4(0) *IP address
   DC 8X'00' *Reserved

* TPIRECV and TPISEND Communication fields *

REQLEN DC A(0) *Requested receive/send length
ACTLEN DC A(0) *Actually received/sent length
RECVFLAG DC A(0) *RECV flags
RECVREAD EQU 0 *Read data
RECVPEEK EQU 2 *Peek at data
SENDFLAG DC A(0) *SEND flags
SENDDATA EQU 0 *Send data

* Socket call error status information *

TRCMLFUN DC CL8 ' ' *Socket function for errorlog
ERRNO DC A(0) *Socket error code
RETCODE DC A(0) *Socket return code
MSGCODE DC AL2(0) *Message code to be returned

* Table over valid record id's and length of records *

RECIDBXL DC A(RECIDST,3,RECIDSL-3)
RECIDST EQU * *Record id and length table
   DC 'C'1',AL2(230) *Add new record
   DC 'X'31',AL2(230) *Add new record ASCII 1
   DC 'C'2',AL2(230) *Update existing record
   DC 'X'32',AL2(230) *Update existing record ASCII 2
   DC 'C'3',AL2(24) *Query existing record
   DC 'X'33',AL2(24) *Query existing record ASCII 3
   DC 'C'4',AL2(24) *Delete existing record
   DC 'X'34',AL2(24) *Delete existing record ASCII 4
   DC 'C'A',AL2(230) *Query response with data
   DC 'C'a',AL2(230) *Query response with data
   DC 'X'41',AL2(230) *Query response with data ASCII A
   DC 'X'61',AL2(230) *Query response with data ASCII a
   DC 'C'B',AL2(89) *Response with text
   DC 'C''b',AL2(89) *Response with text
   DC 'X'42',AL2(89) *Response with text ASCII B
   DC 'X'62',AL2(89) *Response with text ASCII b

RECIDSL EQU *

* EZA Task level work area and buffer with control fields *

*
A Beginner's Guide to MVS TCP/IP Socket Programming

H.1.4 TPISERVD Concurrent Server DB2 Access
* Name: TPISERVD

* Function: This module processes the transactions received from a TPI client. The module is called from TPISERV when an input buffer has been received.

* Interface: R1 -> parameter list with three pointers:
  * +0 -> TPISCB TPI Server task Control Block.
  * +4 -> Buffer holding input record and in which output record will be built.
  * +8 -> Fullword with length of input record.

* Logic: The input record is analyzed for a function code, that identifies which processing is required by this module.

Four basic functions are supported:

A: Add a row to DB2. The passed buffer contains all data required to build the SQL variables to be inserted into DB2.

U: Update a row in DB2. The passed buffer contains all data of all columns in the row. All columns are updated.

D: Delete a row in DB2. The passed buffer contains the primary key: The IP address.

Q: Query a row in DB2. The passed buffer contains the primary key: The IP address. An output buffer will be constructed with data from all columns in the row.

The following response codes can be returned from this module in a response record to the client:

0000 Successfull processing. If it was a query the rest of the record holds the fetched data.

0001 Requested row does not exist

0002 Record not added - IP address already defined

0003 (Not returned by TPISERVD, but by TPISERV)

0004 Invalid function code in input record

0005 Invalid IP address - syntax error

0006 Undefined SQL error - SQLCode and first line of DSNTIAR message is returned

* Abends: none

* Returncode: none

* Written: May 28'th 1994 at ITSO Raleigh

* Modified:

***********************************************************************
PRINT GEN
TPISCB TPISCB TYPE=DSECT *Server task Control Block dsect
TPIMCB TPIMCB TYPE=DSECT *Main task Control Block dsect
PUSH PRINT
EXEC SQL INCLUDE TPIREC
POP PRINT
**TPREC**  **TPREC** TYPE=DSECT  *TPI input and output record dsect

**TPISERVD INIT**  'TPI Server database access module',MODE=24,RENT=NO,  
BASE=(12,11)

*---------------------------------------------------------------------*

* Establish addressability to both Server task and Main task  *
* Control Blocks.  *
* * Pick up pointer to input buffer. Data is in EBCDIC at this point  *
* in time.  *
* * Acquire storege for SQL work area.  *
* *---------------------------------------------------------------------*

L R10,0(R1)  *-> Server task Control Block
USING TPISCB,R10
L R9,TPISMCB  *-> Main task Control Block
USING TPIMCB,R9
L R8,4(R1)  *-> Buffer holding input record
USING TPIREC,R8
TPITRC TYPE=INIT,  *Enable trace points  
TRACE=YES,  
MOD=TPISERVD  *Tracing module is TPISERVD
L R7,SQLDSIZ  *Length of SQL work area
STORAGE OBTAIN,  *Getmain SQL  
LENGTH=(R7),  *- Work area
LOC=BELOW
LR R7,R1  *-> SQL work area
USING SQLDSECT,R7

*---------------------------------------------------------------------*

* IP address is presented to the user as max 15 character text  *
* string in dotted decimal notation. The key in DB2 is a  *
* fullword in network byte order.  *
* Module TPIIADDR will convert from dotted decimal format to  *
* network byte order format.  *
* *---------------------------------------------------------------------*

LA R2,IIPADDR  *-> 15 char ip address
CALL TPIIADDR,((R2),  *R2 point to dotted decimal value  
HIPADDR),VL  *Convert to fullword
LTR R15,R15  *Was IP address OK ?
BE RECIACK  *- Yes, it translated to fullword
MVC MSGCODE,=AL2(5)  *Invalid IP address format
B WRITEMSG  *Write back error message

*---------------------------------------------------------------------*

* Test for function code in received buffer and pass control to  *
* function specific parts of this module.  *
* *---------------------------------------------------------------------*

RECIACK EQU  *

CLI IRECID,IRECADD  *Is it Add ?
BE ADDRDEC  - Yes, do it.
CLI IRECID,IRECUPD  *Is it Update ?
BE UPDREC  - Yes, do it.
CLI IRECID,IREQUE  *Is it Query ?
BE QUEREC  - Yes, do it.
CLI IRECID,IRECDEL  *Is it Delete ?
BE DELREC  - Yes, do it.
MVC MSGCODE,=AL2(4)  *Invalid function requested.
ADDREC EQU *
MVC MSGCODE,=AL2(0) *Anticipate record was added.
CLC SQLCODE,=A(0) *Was insert succesfull?
BE WRITEMSG *- Yes, OK response is set
B BADSQL *Else send back SQL message

QUEREC EQU *
EXEC SQL SELECT * INTO :
:HIPADDR,
:IHOSTNM,
:IADDNM,
:IROOM,
:IONWER,
:IONNMPH,
:IEQUIP,
:IOPERSYS,
:ITEX
FROM TPIDATA WHERE IPADDR = :HIPADDR
CLC SQLCODE,=A(0) *Was Query succesfull?
BNE BADSQL *- No, send back SQL message
LA R2,IIPADDR *Character IP address
CALL TPIINTOA,(HIPADDR, :HIPADDR, FULLWORD FORMAT (R2),VL) *Fullword format to dotted decimal format by
MVC ICODE,=CL4'0000' *OK Message number
MVI IQRESP *Query response
B RETURN *Everything OK for return

UPDATE REC EQU *
MVC MSGCODE,=AL2(0) *Anticipate record was updated.
CLC SQLCODE,=A(0) *Was Update succesfull?
BNE BADSQL *- No, send back SQL message
LA R2,IIPADDR *Character IP address
CALL TPIINTOA,(HIPADDR, :HIPADDR, FULLWORD FORMAT (R2),VL) *Fullword format to dotted decimal format by
MVC ICODE,=CL4'0000' *OK Message number
MVI IQRESP *Query response
B RETURN *Everything OK for return
* contents of the row - make the required modifications and return *
* the full record in an update request. *
* *---------------------------------------------------------------------*
UPDREC EQU *
EXEC SQL UPDATE TPIDATA C
SET C
IPADDR = :HIPADDR, C
HOSTNM = :IHOSTNM, C
ADDNM = :IADDNM, C
ROOM = :IROOM, C
OWNER = :IOWNER, C
OWNERPH = :IOWNERPH, C
EQUIP = :IEQUIP, C
OPERSYS = :IOPERSYS, C
TEXT = :ITEXT C
WHERE IPADDR = :HIPADDR
CLC SQLCODE,A(0) *Was update succesfull?*
BNE BADSQL *- No, send SQL message
MVC MSGCODE,AL2(0) *OK, Record was updated
B WRITEMSG *Write back OK message
* *---------------------------------------------------------------------*
* Delete a row in DB2. *
* *---------------------------------------------------------------------*
DELREC EQU *
EXEC SQL DELETE FROM TPIDATA C
WHERE IPADDR = :HIPADDR
CLC SQLCODE,A(0) *Was delete succesfull?*
BNE BADSQL *- No, send SQL message
MVC MSGCODE,AL2(0) *OK Record was deleted
B WRITEMSG *Send OK message back
* *---------------------------------------------------------------------*
* Build response header with response record id and code *
* TPISERV will find a message based in the code and put that into *
* output record, before it is sent. *
* *---------------------------------------------------------------------*
WRITEMSG EQU *
LH R2,MSGCODE *This message code to return
CVD R2,DORD *We like character data..
OI DORD+7,X'0F' *Reads nice and clear
UNPK ICODE,DORD *Number into buffer
MVI IRECID,IRESP *This is response message record id
B RETURN *Return to TPISERV
* *---------------------------------------------------------------------*
* If SQLcode <> 0, we come here. *
* *---------------------------------------------------------------------*
* SQLCode = 100 is OK and means: Row not found - we return a *
* response code 0001 to the client. *
* SQLCode = -803 is also OK and means: You tried to insert an IP *
* address, that already existed - we return a *
* response code 0002 to the client. *
* *---------------------------------------------------------------------*
* Other SQLCodes are handed over to DSNTIAR for translation into *
* some text. The full DSNTIAR Message is logged on the log file, *
* the SQL Code and the first 75 bytes of the DSNTIAR message buffer *
* are returned to the client as message text. *
**A Beginner's Guide to MVS TCP/IP Socket Programming**

```
*---------------------------------------------------------------------*
| BADSQL EQU *                                                    |
| CLC SQLCODE,=F'100'  *Record not found?  |
| BE SQLNDFND  *- Yes, we take care of this one |
| CLC SQLCODE,=F'-803'  *Duplicate record ID?  |
| BNE SQLERR  *- No. |
| MVC MSGCODE,=AL2(2)  *This one we handle |
| B WRITEMSG  *Treat as normal response |
| SQLNDFND EQU *                                |
| MVC MSGCODE,=AL2(1)  *We handle this one |
| B WRITEMSG  *Treat as normal response |
| SQLERR EQU *                                        |
| TPITRC 'Bad SQL Return Code', W=SQLCODE  *Trace the SQLCode |
| CALL DSNTIAR, (SQLCA, C | SQL Communications Area |
|          , C | DSNTIAR Return area |
|          , VL  *L'DSNTIAR output lines |
| LA R3,DSNTIARA+2  *First line |
| L R4,DSNTIARP  *L'each line |
| LR R5,R3  |
| AH R5,DSNTIARA  *First byte after area |
| S R5,DSNTIARP  *--> Last possible line |
| BADSQLLP EQU *                                         |
| CLC 0(80,R3),=CL80' '  *Empty line=>no more |
| BE BADSQLNM  *We are done |
| TPILOG MOD=TPISERV, MSGNO=1, TEXT=(R3)  *Log the full |
| BXLE R3,R4,BADSQLLP  *Put them out all |
| BADSQLNM EQU *                                         |
| MVC IMESSAGE,=CL80' '  *Clear message area |
| MVC ICODE,=CL4'0006'  *SQL Error |
| MVI IRECID,IRESP  *Response record |
| L R2,SQLCODE  *This was the SQLCode |
| LTR R2,R2  *Was it negative? |
| BP BADSQLPO  *- No, it is positive |
| LPR R2,R2  *Ensure it is positive |
| MVI IMESSAGE,C'-'  *Show it was negative |
| BADSQLPO EQU *                                         |
| CVD R2,DORD  |
| OI DORD+7,X'0F'  |
| UNPK IMESSAGE+1(3),DORD  *Put it into message |
| MVC IMESSAGE+5(75),DSNTIARA+2  *First 75 bytes from DSNTIAR |

*---------------------------------------------------------------------*

* Return to TPISERV, Output buffer has been built. *

*---------------------------------------------------------------------*

RETURN EQU *
TERM RC=0
```

---

```
*---------------------------------------------------------------------*
| HIPADDR DC F'0'  *Hexadecimal IP Address |
| HIPADDRT DC CL15' '  *Character IP address |
| MSGCODE DC H'0'  *Return message code |
| DORD DC D'0'  |

*---------------------------------------------------------------------*

---

A Beginner's Guide to MVS TCP/IP Socket Programming 303
EXEC SQL INCLUDE SQLCA
DSNTIARP DC A(80)
DSNTIARA DS 0C
DSNTIARL DC AL2(8*80)
DC 8CL80' ' END

H.1.5 TPISEND Send Data Over a Stream Socket

*******************************************************************************
**
* Name: TPISEND
*
* Function: Issue SEND socket calls to send a specified number of bytes.
*
* Interface: R1 -> parameter list with the following pointers:
* +0 -> EZA Global work area (In)
* +4 -> EZA Task work area (In)
* +8 -> Halfword with socket descriptor (In)
* +C -> Buffer (In)
* +10 -> Fullword with requested length (In)
* +14 -> Fullword for actual length (Out)
* +18 -> Fullword with SEND flags (In)
* +20 -> Fullword for SEND Retcode (Out)
*
* Logic: This module is to send data from a buffer to a socket.
* The routine will repeat the send operation until either the requested length has been sent or send returns a length of zero (peer closed socket)
*
* Abends: - none -
*
* Returncode: RC = 0 Everything OK
* RC = 4 Peer closed the socket
* RC = 8 Examine Retcode and Errorcode for details
*
* Written: June 18' th 1994 at ITSO Raleigh
*
* Modified:
*
*******************************************************************************
PARMS DSECT
PEZAGLOB DC A(0) *-> EZA Global workarea
PEZATASK DC A(0) *-> EZA Task workarea
PSD DC A(0) *-> Socket descriptor
P BUFFER DC A(0) *-> Send buffer
PREQLEN DC A(0) *-> Word with requested length
PACTLEN DC A(0) *-> Word to return actual length
PSENDFLG DC A(0) *-> Word with SEND flags
PRETCODE DC A(0) *-> Retcode to return
PERRNO DC A(0) *-> Error no to return
*
PRINT NOGEN
EZAGLOB EZA S M I TYPE=GLOBAL, *EZA Global workarea
STORAGE=DSECT
EZATASK EZASMI TYPE=Task, C *EZA Task workarea
STORAGE=DSECT

TPISEND INIT 'TPI Send data over a socket', RENT=NO, C
BASE=(12), MODE=24

LR R10, R1 *So we want destroy it
USING PARMS, R10 *Here we have them all
L R2, PSD *--> Socket descriptor
MVC SD, 0(R2) *Now we have it
L R2, PREQLEN *--> Requested length
MVC REQLEN, 0(R2) *Now we have it
MVC REMLEN, 0(R2) *Remaining length := requested len.
L R2, PBUFFER *--> Buffer
ST R2, BUFNEXT *Here to fetch first byte
L R2, PSENDFLAGS *--> Send flags
MVC SENDFLAGS, 0(R2) *Copy to us self
L R8, PEZAGLOB *--> EZA Global workarea
USING EZAGLOB, R8 *Addressability
L R9, PEZATASK *--> EZA Task workarea
USING EZATASK, R9 *Addressability
XC RC, RC *RC = 0
XC RETCODE, RETCODE *RETCODE = 0
XC ERRNO, ERRNO *ERRNO = 0
XC ACTLEN, ACTLEN *ACTLEN = 0

DOSEND EQU *
L R2, BUFNEXT *--> Here to fetch data
EZASMI TYPE=Send, C *Send call
S=SD, *From this socket descriptor
NBYTE=REMLEN, *Request remaining length
BUF=(R2), *--> Read data into buffer
FLAGS=SENDFLAGS, *Send flags
ERRNO=ERRNO, *Put error here
RETCODE=RETCODE *Retcode/length here
ICM R15, 15, RETCODE *Let us have a look
BM EZAERROR * < 0 Something seriously wrong
BZ SDCLOSED * = 0 Means peer closed socket
A R15, ACTLEN *Add to actual until now
ST R15, ACTLEN *Update it
L R15, REMLEN *Original remaining length
S R15, RETCODE *Minus what we got now
ST R15, REMLEN *New remaining length
L R2, BUFNEXT *Here we started to fetch
A R2, RETCODE *If more, fetch from here
ICM R15, 15, REMLEN *Is there more to send ?
BNZ DOSEND *-- Yes, do a new send
B RETURN *-- No, we have sent all

SDCLOSED EQU *
MVC RC, =A(4) *Set RC=4 for socket closed
B RETURN *And return current status

EZAERROR EQU *
MVC RC, =A(8) *Set RC=8 for EZA error codes
L R2, PRETCODE *--> Callers RETCODE
MVC 0(L'RETCODE, R2), RETCODE *Return RETCODE to Caller
L R2, PERRNO *--> Callers ERRNO
MVC 0(L'EERRNO, R2), ERRNO *Return ERRNO to Caller

RETURN EQU *
L R2, PACTLEN *--> Callers ACTLEN
MVC 0(L'ACTLEN, R2), ACTLEN *Return actual length
L R15, RC *Return code
TERM RC=R15
LTORG
*
SD DC AL2(0) *Socket descriptor
REQLEN DC A(0) *Requested length
ACTLEN DC A(0) *Sent so far
REMLEN DC A(0) *Remaining length
BUFNEXT DC A(0) *-> Where to fetch next byte
SENDFLAG DC A(0) *Send flags
RETCODE DC A(0) *EZASMI Returncode
ERRNO DC A(0) *EZASMI Error code
RC DC A(0) *TPISEND Return code
*
END

H.1.6 TPIRECV Receive Data Over a Stream Socket

*********************************************************************************
*                             *                                           *
* Name: TPIRECV                *                                           *
*                             *                                           *
* Function: Issue RECV socket calls to receive a specified                *
* number of bytes.                   *                                           *
*                             *                                           *
* Interface: R1 -> parameter list with the following pointers:          *
* +0  -> EZA Global work area (In)                                       *
* +4  -> EZA Task work area (In)                                        *
* +8  -> Halfword with socket descriptor (In)                           *
* +C   -> Buffer (Out)                                                   *
* +10  -> Fullword with requested length (In)                           *
* +14  -> Fullword for actual length (Out)                              *
* +18  -> Fullword with RECV flags (In)                                 *
* +1C   -> Fullword for RECV Retcode (Out)                               *
* +20  -> Fullword for RECV Error code (Out)                            *
*                             *                                           *
* Logic:  This module is to read data from a socket into a              *
*         program buffer.                                                *
*         The routine will repeat the RECV operation until               *
*         either the requested length has been read or RECV              *
*         returns a length of zero (peer closed socket)                  *
*                             *                                           *
* Abends:   - none -                                                    *
*                             *                                           *
* Returncode: RC = 0 Everything OK                                       *
* RC = 4 Peer closed the socket                                         *
* RC = 8 Examine Retcode and Errorcode for details                      *
*                             *                                           *
* Written:  June 18'th 1994 at ITSO Raleigh                              *
*                             *                                           *
* Modified:    *                                                         *
*                              *                                           *
*********************************************************************************
PARMS DSECT
PEZAGLOB DC A(0) *---> EZA Global work area
PEZATASK DC A(0) *---> EZA Task work area
PSD DC A(0) *---> Socket descriptor
PBUFFER DC A(0) *---> Read buffer
PREQLEN DC A(0) *---> Word with requested length
PACTLEN DC A(0) *---> Word to return actual length
PRECVFLG DC A(0) *---> Word with RECV flags
PRETCODE DC A(0) *---> Retcode to return
PERRNO DC A(0) "-> Error no to return

PRINT NOGEN

EZAGLOB EZASMI TYPE=GLOBAL, STORAGE=DSECT

EZATASK EZASMI TYPE=TASK, STORAGE=DSECT

TPIRECV INIT 'TPI Receive data over a socket', RENT=NO, BASE=(12), MODE=24

LR R10, R1 "So we wont destroy it
USING PARMS, R10 "Here we have them all
L R2, PSD "-> Socket descriptor
MVC SD, 0(R2) "Now we have it
L R2, PREQLEN "-> Requested length
MVC REQLEN, 0(R2) "Now we have it
MVC REMLEN, 0(R2) "Remaining length := requested len.
L R2, PBUFFER "-> Buffer
ST R2, BUFNEXT "Here to store first byte
L R2, PRECVFLG "-> Receive flags
MVC RECVFLAG, 0(R2) "Copy to us self
L R8, PEZAGLOB "-> EZA Global workarea
USING EZAGLOB, R8 "Addressability
L R9, PEZATASK "-> EZA Task workarea
USING EZATASK, R9 "Addressability
XC RC, RC "RC = 0
XC RETCODE, RETCODE "RETCODE = 0
XC ERRNO, ERRNO "ERRNO = 0
XC ACTLEN, ACTLEN "ACTLEN = 0

DORECV EQU *
L R2, BUFNEXT "-> Here to store data
EZASMI TYPE=RECV, "Receive call
S=SD, "From this socket descriptor
NBYTE=REMLEN, "Request remaining length
BUF=(R2), "-> Read data into buffer
FLAGS=RECVFLAG, "Receive flags
ERRNO=ERRNO, "Put error here
RETCODE=RETCODE "Retcode/length here
ICM R15, 15, RETCODE "Let us have a look
BM EZAERROR "< 0 Something seriously wrong
BZ SDCLOSED "= 0 Means peer closed socket
A R15, ACTLEN "Add to actual until now
ST R15, ACTLEN "Update it
L R15, REMLEN "Original remaining length
S R15, RETCODE "Minus what we got now
ST R15, REMLEN "New remaining length
L R2, BUFNEXT "Here we started to store
A R2, RETCODE "If more, it goes here
ICM R15, 15, REMLEN "Is there more to receive?
BNZ DORECV "- Yes, do a new receive
B RETURN "- No, we have it.

SDCLOSED EQU *
MVC RC, =A(4) "Set RC=4 for socket closed
B RETURN "And return current status

EZAERROR EQU *
MVC RC, =A(8) "Set RC=8 for EZA error codes
L R2, PRETCODE "-> Callers RETCODE
MVC 0(L‘RETCODE, R2), RETCODE "Return RETCODE to Caller
L R2, PERRNO "-> Callers ERRNO
MVC 0(L'ERRNO,R2),ERRNO *Return ERRNO to Caller
RETURN EQU *
L R2,PACTLEN *-> Callers ACTLEN
MVC 0(L'ACTLEN,R2),ACTLEN *Return actual length
L R15,RC *Return code
TERM RC=R15
LTORG
*
SD DC AL2(0) *Socket descriptor
REQLEN DC A(0) *Requested length
ACTLEN DC A(0) *Read so far
REMLLEN DC A(0) *Remaining length
BUFNEXT DC A(0) *-> Where to store next byte
RCVFLAG DC A(0) *Receive flags
RETCODE DC A(0) *EZASMI Returncode
ERRNO DC A(0) *EZASMI Error code
RC DC A(0) *TPIRECV Return code
*
END

H.1.7 TPIMCB Macro Main Task Control Block

MACRO
&NAME TPIMCB &TYPE=DSECT
PUSH PRINT
PRINT GEN
AIF ('&TYPE' EQ 'DSECT').DSEC
&NAME DS 0F
AGO .HDOK
&NAME DSECT
.HDOK ANOP
&NAME DSECT
.HDOK ANOP
**********************************************************************
* *
* TPI Main Control Block (TPIMCB). *
* *
**********************************************************************
TPIMEYE DC CL8'TPIMCB' *Eyecatcher
TPIMGLOB DC A(0) *-> EZA Global workarea
TPIMDB2 DC CL4' ' *DB2 Subsystem name to use
TPIMTCPI DC CL8' ' *TCPIP Address space name
TPIMPORT DC AL2(0) *Listen port number
TPIMNOST DC AL2(0) *Number of server subtasks
TPIMAXS DC AL2(0) *Maximum number of sockets
TPIMAXD DC AL4(0) *Maximum descriptor number
TPIMCBE DC CL8' ' *TCB Address in EBCDIC
TPIMCBB DC 3A(0) *TPISCB Table BXLE addresses
TPIMSCB DC 3A(0) *TPIMSO Table BXLE addresses
TPIMSOCK DC A(0) *Listen socket number
TPIMREIN DC A(0) *Times server reinstated
TPIMECPB DC A(0) *-> Main Wait ECBList
TPIMFECB DC A(0) *Modify ECB
TPIMECBS DC A(0) *Select ECB
TPIMCLNI DS 0C *Main task client id
TPIMCDOM DC A(0) *Domain: AF-INET
TPIMCNAM DC CL8' ' *Our address space name
TPIMCTSK DC CL8' ' *Our task id
DC 20X'00' *Reserved (part of clientid)
TPIMLQNM DC CL8'TPI' *TPI Log writer Qname
TPIMLRNM DC CL8'TPILOGWT' *TPI Log writer Rname
H.1.8 TPISCB Macro Subtask Control Block

MACRO
&NAME TPISCB &TYPE=DSECT
  AIF ('&TYPE' EQ 'DSECT').DSEC
  &NAME DS 0F
  AGO .HDOK
  .DSEC ANOP
&NAME DSECT
  .HDOK ANOP
*******************************************************************************
  *   *
  * TPI Server Control Block (TPISCB).   *
  *   *
*******************************************************************************
TPISEYE DC CL8’TPISCB’   *Eyecatcher
TPISTASK DC A(0)   *-> EZA Task workarea
TPISMCB DC A(0)   *-> TPIMCB
TPISTCB DC A(0)   *-> Subtask TCB
TPISTCBE DC CL8’ ’   *Subtask TCB address in EBCDIC
TPISECB DC A(0)   *Subtask wait-for-work ECB
TPISTEBCB DC A(0)   *Subtask termination ECB
TPISIECB DC A(0)   *Subtask initialization ECB
TPISSOD DC AL2(0)   *Parent socket descr. no.
TPISNSOD DC AL2(0)   *Subtask socket descr. no.
TPISCINI DS 0C   *Server task client id
TPISCDOM DC A(0)   *Addressing Family
TPISCNAM DC CL8’ ’   *Address space name
TPISCTSK DC CL8’ ’   *Subtask name
  DC 20X'00'   *Reserver - part of clientid
TPISCTYP DC X'00'   *Current Client option
TPISASCII EQU BIT0   *- Client is ASCII based
TPISEBCD EQU BIT1   *- Client is EBCDIC based
  DC XL3'00'   *Reserved
* 
TPISCBLN EQU *-&NAME
MEND

H.1.9 TPILOG Macro Issue Logwriter Request

MACRO
TPILOG &MOD=TPIMAIN,        C
   &FUNC=,        C
   &ERRNO=,        C
   &RETCODE=,        C

A Beginner's Guide to MVS TCP/IP Socket Programming
A Beginner’s Guide to MVS TCP/IP Socket Programming

&MSGNO=0,
&TEXT='No text'

GBLB &TPLOGSW
AIF (&TPLOGSW).NOTFRST
B TPIA&SYSNDX.

MSEC200 DC F'20'
MSEC500 DC F'50'
TPLOGR2 DC A(0)

TPLOGEQ EQU *
   LA R14,TPIMLQN
   LA R15,TPIMLRNM
   ENQ ((R14),(R15),E,8,STEP)
   BR R2

TPLOGDQ EQU *
   LA R14,TPIMLQN
   LA R15,TPIMLRNM
   DEQ ((R14),(R15),8,STEP)
   BR R2

&TLOGSW SETB 1
TPIA&SYSNDX. EQU *
   .NOTFRST ANOP
      ST R2,TPLOGR2  *Save work register
      BAL R2,TPLOGEQ  *Do enqueue
      TM TPIMLECB,BIT0  *Is he waiting
      BO TPILOG&SYSNDX.
      STIMER WAIT,BINTVL=MSEC500
      TM TPIMLECB,BIT0  *Is he waiting
      BO TPIO&SYSNDX.  *Drop it..

TPILOG&SYSNDX. EQU *
   MVC TPIMLMOD,=CL8'&MOD.'
   AIF (T'&FUNC EQ 'O').NOFUNC
   AIF ('&FUNC'(1,1) EQ '''').FUNSTR
   MVC TPIMLFUN,&FUNC.
   AGO .NOFUNC

.FUNSTR ANOP
   MVC TPIMLFUN,=CL8&FUNC.

.NOFUNC ANOP
   XC TPIMLRERR,TPIMLRERR
   AIF (T'&ERRNO EQ 'O').NOERRNO
   MVC TPIMLRERR,&ERRNO

.NOERRNO ANOP
   XC TPIMLRET,TPIMLRET
   AIF (T'&RETCODE EQ 'O').NORETC
   MVC TPIMLRET,&RETCODE

.NORETC ANOP
   LA R15,&MSGNO
   ST R15,TPIMLMNO
   AIF (T'&TEXT EQ 'O').NOTEXT
   MVI TPIMLTXT,C'
   MVC TPIMLTXT+1(L'TPIMLTXT-1),TPIMLTXT
   AIF ('&TEXT'(1,1) EQ '''').TEXTSTR
   AIF ('&TEXT'(1,1) EQ '(').REGADR
   MVC TPIMLTXT(L'&TEXT),&TEXT.
   AGO .NOTEXT

.REGADR ANOP
   MVC TPIMLTXT,0&TEXT.
   AGO .NOTEXT

.TEXTSTR ANOP
   LCLA &NBYTES
&NBYTES SETA K'&TEXT
&NBYTES SETA &NBYTES-2
   MVC TPIMLTXT(&NBYTES.),=C&TEXT.
A Beginner's Guide to MVS TCP/IP Socket Programming

H.1.10 TPITRC Macro Issue Trace Request

MACRO
TPITRC &TXT, &REG=, &WORD=, &H=, &W=, &MOD=, C
&TYPE=TRACE, &TRACE=YES
GBLB &TRCSW
GBLB &TPITRC
GBLC &TPITRCM
AIF (&TRCSW).NOTFRST
B TRCA&SYSNDX.
TPITRCTX DS 0CL80
TPITRCHX DC CL8' ',CL1' '
TPITRCT DC CL71' '
TRCHEX DC C'0123456789ABCDEF'
TRCDWORD DC D'0'
DS 0F
TRCA&SYSNDX. EQU *
&TRCSW SETB 1
&TPITRC SETB 1
.NOTFRST ANOP
AIF (&TPITRC).TRON
MEXIT
.TRON ANOP
AIF ('&TYPE' EQ 'TRACE').DOTRCE
AIF ('&TRACE' EQ 'YES').TRYES
&TPITRC SETB 0
MEXIT
.TRYES ANOP
&TPITRCM SETC '&MOD.'
MEXIT
.DOTRCE ANOP
AIF (T'&REG EQ 'O').NOREG
LR R15, &REG
AGO .COMM
.NOREG ANOP
AIF (T'&WORD EQ 'O').NOWORD
L R15, &WORD
AGO .COMM
.NOWORD ANOP
AIF (T'&W EQ 'O').NOW
L R15, &W
AGO .COMM
.NOW ANOP
AIF (T'&H EQ 'O').N0H
LH R15, &H
AGO .COMM
.NOH ANOP
.COMM ANOP
SR R14, R14
SLDL R14, 4
STM R14, R15, TRCDWORD
UNPK TPITRCHX, TRCDWORD

*Wake him up
*Wait for him to do it
*Do dequeue
*Restore work register
H.1.11 TPIMASK Macro Set and Test Bits in Select Mask

MACRO
TPIMASK &TYPE,&MASK=,&SD=
SR R14,R14 *Nullify
AIF ('&SD'(1,1) EQ '(').SDREG
LH R15,&SD *Socket descriptor
AGO .SDOK
.SDREG ANOP
LR R15,&SD *Socket descriptor
.SDOK ANOP
D R14,=A(32) *Divide by 32
SLL R15,2 *Multiply offset with word length
AIF ('&MASK'(1,1) EQ '(').MASKREG
LA R1,&MASK *Here mask starts
AGO .MASKOK
.MASKREG ANOP
LR R1,&MASK *Here mask starts
.MASKOK ANOP
AR R15,R1 *Here our word starts
LA R1,1 *Rightmost bit on
SLL R1,0(R14) *Shift left rest from division
O R1,0(R15) *Or bits from mask
AIF ('&TYPE' EQ 'SET').DOSET
C R1,0(R15) *If equal, bit was on
MEXIT
.DOSET ANOP
ST R1,0(R15) *New mask
MEND

H.1.12 TPIREC Macro DB2 Row Layout

MACRO
&NAME TPIREC &TYPE=DSECT
PUSH PRINT
PRINT GEN
AIF ('&TYPE' EQ 'DSECT').DSEC
&NAME DS 0F
AGO .HDOK
.DSEC ANOP
&NAME DSECT
.HDOK ANOP
**********************************************************************
H.1.13 TPIMSO Macro Socket Descriptor Table

MACRO
&NAME TPIMSO &TYPE=DSECT
AIF ('&TYPE' EQ 'DSECT').DSEC
&NAME DS 0F
AGO .HDOK
&DSEC ANOP
&NAME DSECT
.HDOK ANOP
**********************************************************************
* *
* TPI Main task Socket Descriptor Table Entry (TPIMSO) *
* *
**********************************************************************

TPIMSOEYE DC CL8'TPIMSO'  *Eye catcher
TPIMSN DC AL2(0)  *Main task socket number
TPIMSBIT DC X'00'  *Status bits
TPIMSACT EQU BIT0  *This socket is in use
TPIMSLIS EQU BIT1  *This is main listen socket
TPIMSREA EQU BIT2  *Read OK (only Listen socket)
TPIMSWRT EQU BIT3  *Write OK if TPIMSENO<>0
TPIMSEX EQU BIT4  *Exception expected (Takesocket)
  DC X'00'  *Reserved
TPIMSENO DC AL2(0)  *Pending Error number
TPIMSSOC DS 0C  *Socket structure
TPIMSFAM DC AL2(0)  *Addressing family
TPIMSPOR DC AL2(0)  *Peer port number
TPIMSADR DC AL4(0)  *Peer IP address
  DC XL8'00'  *Reserved
*
H.2 TPI REXX Client Application

The TPI REXX client consists of a REXX program, an ISPF panel and an ISPF message definition member.

H.2.1 TPI REXX Client

H.2.2 TPI REXX Client ISPF Panel Definition

H.2.3 TPI REXX Client ISPF Message Definitions

H.2.1 TPI REXX Client

/* REXX */
/*------------------------------------------------------------------*/
/* */
/* Name: TPIREXXC - TPI Demo application REXX Client */
/* */
/* Function: Controls user interface for update and query of */
/* TPI data. User interface is ISPF panel. Communication* /
/* with TPI server is via REXX Socket interface. */
/* */
/* Interface: - none - */
/* */
/* Logic: This REXX controls a user dialog, where the user */
/* uses ISPF panels to interface to the TPI server. The */
/* REXX pgm. builds a transaction, which is sent to the */
/* TPI server over a socket connection, receives the */
/* response and displays it to the user. */
/* */
/* Returncode: RC = 0, processing OK */
/* Everything else is non-successful returncode from */
/* socket interface. */
/* */
/* Written: April 13, 1995 at ITSO Raleigh */
/* */
/* Modified: */
/* */
/*------------------------------------------------------------------*/

dotrace = 0 /*Controls tracing */
* /
* dotrace = 1 for trace */
tpiport = '9999' /*Server port number */
tpiserver = 'mvs18' /*Server host name */
subtaskid = 'tpirexxc' /*Subtask id */
/*----------------------------------------------------------------------*/
/* */
/* All socket calls are performed by subroutine DoSocket */
/* */
/*------------------------------------------------------------------*/
sockval = DoSocket('Terminate') /*Ensure clean interface*/
/*----------------------------------------------------------------------*/
/* */
/* Initialize REXX socket interface */
/* and get our own TCP/IP Client id. */
/* */
/*----------------------------------------------------------------------*/
Address TSO "ALLOC FI(SYS1.TCPPARMS(TCPPDATA)) SHR"
sockval = DoSocket('Initialize', subtaskid)
if sockrc <> 0 then do
  say 'Initialize failed, rc='sockrc
  exit(sockrc)
end
sockval = DoSocket('Getclientid')
if sockrc <> 0 then do
  say 'Getclientid failed, rc='sockrc
  exit(sockrc)
end
servipaddr = DoSocket('Gethostbyname', tpiserver)
if sockrc <> 0 then do
  say 'Gethostbyname failed, rc='sockrc
  x=Doclean
  exit(sockrc)
end
numips = words(servipaddr)
parse value servipaddr with s1 s2 s3 s4 s5 s6 s7 s8 s9
do i = 1 to numips
  sipaddr.i = word(servipaddr, i)
  if dotrace then say 'sipaddr.'i' = 'sipaddr.i
end
sipaddr.0 = numips
if dotrace then say 'Number of IP addresses = 'sipaddr.0
/*------------------------------------------------------------------*/
/* */
/* Initialize REXX and ISPF variables */
/* */
/*------------------------------------------------------------------*/
ispfloop = 0
tpiact = 'Q'
tpiip = ''
tpihost = ''
tpiaddnm = ''
tpiroom = ''
tpiowner = ''
tpiiphone = ''
tpiequip = ''
ptios = ''
tpitext = ''
tpimsg = ''
/*------------------------------------------------------------------*/
/* */
/* Display dataentry panel, process input until user presses PF3 */
/* */
/*------------------------------------------------------------------*/
Do until ispfloop
  address ispexec "Display panel(tpi)"
  if rc > 0 then do
    ispfloop = 1
    iterate
  end
  if tpiact = 'A' | tpiact = 'U' then do
    /*------------------------------*/
    /* */
    /* If user wants to add or update TPI information, build a */
    /* TPI ADD or UPDATE transaction string. */
    /* */
    /*------------------------------*/
    recident = 'TPI'
    recode = '0000'
    recipaddr = substr(tpiip,1,15)
    rechostnm = substr(tpihost,1,18)
recaddnm = substr(tpiaddnm,1,18)
recroom = substr(tpiroom,1,10)
recowner = substr(tpiowner,1,32)
recownerph = substr(tpiphone,1,16)
recequip = substr(tpiequip,1,16)
recopersys = substr(tpios,1,16)
rectext = substr(tpitext,1,80)
if tpiact = 'A' then recid = '1'
if tpiact = 'U' then recid = '2'
record = recid||recident||reccode||recipaddr||rechostnm||recaddnm
record = record||recroom||recowner||recownerph||recequip||recopersys
record = record||rectext
end
else do
/*--------------------------------------------------------------*/
/* */
/* If user wants to delete or query TPI information, build a */
/* TPI DELETE or QUERY transaction string. */
/* */
/*--------------------------------------------------------------*/
recident = 'TPI '
reccode = '0000'
recipaddr = substr(tpiip,1,15)
if tpiact = 'Q' then recid = '3'
if tpiact = 'D' then recid = '4'
record = recid||recident||reccode||recipaddr
end
/*--------------------------------------------------------------*/
/* */
/* Get a socket and try to connect to the server */
/* */
/*--------------------------------------------------------------*/
i = 1
connected = 0
do until (i > sipaddr.0 | connected)
sockdescr = DoSocket('Socket')
if sockrc <> 0 then do
 say 'Socket failed, rc='sockrc
 x=Doclean
 exit(sockrc)
end
name = 'AF_INET '||tpiport||' '||sipaddr.i
sockval = DoSocket('Connect', sockdescr, name)
if sockrc = 0 then do
 connected = 1
end
else do
 parse value respdata with resplen response
 sockval = DoSocket('Close', sockdescr)
 if sockrc <> 0 then do
 say 'Close failed, rc='sockrc
 x=Doclean
 exit(sockrc)
end
i = i + 1
end
if ,connected then do
say 'Connect failed, rc='sockrc
say sockval
x=Doclean
exit(sockrc)
end
/*----------------------------------------------------------------*/
/* */
/* Send the TPI transaction to the TPI server */
/* */
/* */
/*----------------------------------------------------------------*/
sockval = DoSocket('Write', sockdescr, record)
if dotrace then say 'Write returned: 'sockval
if sockrc <> 0 then do
  say 'Write failed, rc='sockrc
  x=Doclean
  exit(sockrc)
end
/*----------------------------------------------------------------*/
/* */
/* Read the response from the TPI Server */
/* */
/* */
/*----------------------------------------------------------------*/
respdata = DoSocket('Read', sockdescr)
if sockrc <> 0 then do
  say 'Read failed, rc='sockrc
  x=Doclean
  exit(sockrc)
end
/*----------------------------------------------------------------*/
/* */
/* Close the socket */
/* */
/* */
/*----------------------------------------------------------------*/
parse value respdata with resplen response
sockval = DoSocket('Close', sockdescr)
if sockrc <> 0 then do
  say 'Socket Close failed, rc='sockrc
  x=Doclean
  exit(sockrc)
end
if substr(response,1,1) = 'A' then do
  /*----------------------------------------------------------------*/
  /* */
  /* If it is a query response, the returned string is a */
  /* */
  /* complete TPI record, which will be unpacked into the */
  /* */
  /* corresponding REXX variables. */
  /* */
  /*----------------------------------------------------------------*/
  tpiip = substr(response,10,15)
  tpihost = substr(response,25,18)
  tpiaddnm = substr(response,43,18)
  tpiroom = substr(response,61,10)
  tpiowner = substr(response,71,32)
  tpiphone = substr(response,103,16)
  tpiequip = substr(response,119,16)
  tpios = substr(response,135,16)
  tpitext = substr(response,151,80)
end
else do
  /*----------------------------------------------------------------*/
  /* */
  /* If it is non-query response, the returned string holds */
  /*----------------------------------------------------------------*/

/* a TPI response code and optionally a return message. */
/* */
/* Response code = 0000 means successful completion. */
/* */
/* */
/*--------------------------------------------------------------*/
if substr(response,6,4) = '0000' then do
    tpimsg = ' '
    address ispexec "setmsg msg(tpi004)"
end
else do
    /*------------------------------------------------------------*/
    /* */
    /* Response code = 0003 means no data found on a query */
    /* request. */
    /* */
    /* */
    /* Clear out all variables. */
    /* */
    /* */
    /*------------------------------------------------------------*/
    tpimsg = substr(response,10,80)
    tpirespc = substr(response,6,4)
    address ispexec "setmsg msg(tpi003)"
    if tpirespc = '0001' then do
        address ispexec "setmsg msg(tpi005)"
        tpimsg = ' '
        tpihost = '
        tpiaddnm = '
        tpiroom = '
        tpiowner = '
        tphone = '
        tpequip = '
        tpios = '
        tpitext = '
    end
    /*------------------------------------------------------------*/
    /* */
    /* Response code = 0002 means data was not added - TPI */
    /* data for specified IP address already existed. */
    /* */
    /* */
    /*------------------------------------------------------------*/
    if tpirespc = '0002' then do
        address ispexec "setmsg msg(tpi006)"
        tpimsg = ' '
    end
end
end
/*--------------------------------------------------------------*/
/* */
/* Terminate socket interface */
/* */
/*--------------------------------------------------------------*/
sockval = DoSocket('Terminate')
if sockrc <> 0 then do
    say 'Socket Close failed, rc='sockrc
    exit(sockrc)
end
Exit(0)
/*--------------------------------------------------------------*/
/* */
/* DoClean Procedure. */
/* */
/* If a socket call failed and we are about to exit this 
/* Rexx application, close the socket and terminate the 
/* socket interface. 
/* */
/*------------------------------------------------------------------*/
Doclean:
  if dotrace then do
    say 'Cleaning up socket descriptor = 'sockdescr
  end
  sockval = DoSocket('Close', sockdescr)
  sockval = DoSocket('Terminate')
return sockres

/*-------------------------------------------------------------*/
/* */
/* DoSocket procedure. */
/* */
/* Do the actual socket call, and parse the return code. */
/* */
/* Return rest of string returned from socket call. */
/* */
/*-------------------------------------------------------------*/
DoSocket:
  numargs = ARG() /*Number of passed args */
  argstring = '' /*Init arg string */
  if dotrace then do /*Tracepoint */
    say 'DoSocket subroutine' /*Trace entry to routine*/
    say ' - Number of args = 'numargs /*Trace number of args */
  end /* */
  do subix=1 to numargs /*Build argument string */
    if dotrace then do /*Tracepoint */
      say ' - arg('subix') = 'arg(subix) /*Trace each argument */
    end /* */
    argstring = argstring||'arg('subix')' /*for the socket call */
    if subix<numargs then do /*If not last argument */
      argstring = argstring||',' /*add a comma */
    end /* */
  end /* */
  msgstat = msg() /*Save message status */
  z = msg("OFF") /*Turn messages off */
  interpret 'Parse value Socket('||argstring||') with sockrc sockres'
  z = msg(msgstat) /*Restore message status*/
  if dotrace then do /*Tracepoint */
    say ' - return code = 'sockrc /*Trace returncode */
    say ' - return string = 'sockres /*Trace return string */
  end /* */
return sockres /*Return socket result */

H.2.2 TPI REXX Client ISPF Panel Definition

)ATTR
@ type(text) intens(high) color(turq) /*hilite(reverse)*/
$ type(text) intens(high) color(green) /*hilite(reverse)*/
% type(text) intens(high) color(red) /*hilite(reverse)*/
+ type(text) intens(high) color(white) /*hilite(reverse)*/
! type(text) intens(high) color(turq) /*hilite(reverse)*/
? type(text) intens(high) color(turq) /*hilite(reverse)*/
_ type(input) intens(high) color(turq) caps(on)
? type(input) intens(high) color(turq) caps(off)
)body
!------------------ TCP/IP MVS Programming Interfaces ------------------------
$OPTION ===>_ZCMD
H.2.3 TPI REXX Client ISPF Message Definitions

TPI001 'Invalid action code'.TYPE=ACTION
'TPI001E: Action code &tpiact. is invalid. You may choose +
'between A for Add, U for Update, D for Delete or Q for Query.' +
'Use a Query before you do an Update.'
TPI002 'IP address is required'.TYPE=ACTION
'TPI002E: You must type in an IP address for all request types.'
TPI003 'Error response received'.TYPE=WARNING
'TPI003E: Server returned a negative response with code=&tpirespc.'
TPI004 'Processing successfully'.TYPE=WARNING
'TPI004I: Request processed successfully.'
TPI005 'No record found'.TYPE=ACTION
'TPI005E: No DB2 record exists for specified IP address.'
TPI006 'Duplicate IP address'.TYPE=ACTION
'TPI006E: Specified IP Address already exists in DB2.'

H.3 TPI DB2 Table Definition

create table tpidata (ipaddr int not null,
    hostnm char(18),
    addnm char(18),
    room char(10),
    owner char(32),
    ownerph char(16),
    equip char(16),
    opersys char(16),
    text char(80),
    primary key(ipaddr));
create unique index tpiindex on tpidata (ipaddr);
commit;
### H.4 Sample Log from TPI Server Execution

The following is an example of logwriter output from an execution of the TPI concurrent server. Two server subtasks are started, and one client connection is processed before the server is modified to close down.

<table>
<thead>
<tr>
<th>Time</th>
<th>Module</th>
<th>Function</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>13:11:11.16</td>
<td>TPIMAIN</td>
<td>TPI Log Writer Task has started</td>
<td></td>
</tr>
<tr>
<td>13:11:11.90</td>
<td>TPISERV</td>
<td>TPIMAIN 001 TPIMAIN Client ID Family=0002 Address Space=T18ATPI Subtask=008FDA28</td>
<td></td>
</tr>
<tr>
<td>13:11:12.20</td>
<td>TPISERV</td>
<td>TPISERV Client ID Family=0002 Address Space=T18ATPI Subtask=008FOBF8</td>
<td></td>
</tr>
<tr>
<td>13:11:12.52</td>
<td>TPISERV</td>
<td>TPISERV Going to sleep.</td>
<td></td>
</tr>
<tr>
<td>13:11:12.54</td>
<td>TPIMAIN</td>
<td>TPIMAIN 001 00000000 Socket descriptor from SOCKET Call</td>
<td></td>
</tr>
<tr>
<td>13:12:28.49</td>
<td>TPIMAIN</td>
<td>Peer socket address - Family=0002 Port number=01024 IP address=9.67.56.81</td>
<td></td>
</tr>
<tr>
<td>13:12:28.50</td>
<td>TPIMAIN</td>
<td>Issuing ACCEPT 2</td>
<td></td>
</tr>
<tr>
<td>13:12:28.51</td>
<td>TPISERV</td>
<td>Takesocket With old descriptor 5</td>
<td></td>
</tr>
<tr>
<td>13:12:28.52</td>
<td>TPISERV</td>
<td>Takesocket returned new descriptor 5</td>
<td></td>
</tr>
<tr>
<td>13:12:28.53</td>
<td>TPIMAIN</td>
<td>Peer socket address - Family=0002 Port number=01024 IP address=9.67.56.81</td>
<td></td>
</tr>
<tr>
<td>13:12:28.54</td>
<td>TPIMAIN</td>
<td>Issuing SELECT with MAXSOC 4</td>
<td></td>
</tr>
<tr>
<td>13:12:28.55</td>
<td>TPIMAIN</td>
<td>Issuing SELECT with MAXSOC</td>
<td></td>
</tr>
<tr>
<td>13:12:28.56</td>
<td>TPIMAIN</td>
<td>Issuing SELECT with MAXSOC 8</td>
<td></td>
</tr>
<tr>
<td>13:12:28.57</td>
<td>TPISERV</td>
<td>Peek returned so many bytes 9</td>
<td></td>
</tr>
<tr>
<td>13:12:28.58</td>
<td>TPISERV</td>
<td>Receive returned so many bytes</td>
<td></td>
</tr>
<tr>
<td>13:12:29.90</td>
<td>TPISERV</td>
<td>Peer socket address - Family=0002 Port number=01024 IP address=9.67.56.81</td>
<td></td>
</tr>
<tr>
<td>13:12:29.93</td>
<td>TPISERV</td>
<td>Peer socket address - Family=0002 Port number=01024 IP address=9.67.56.81</td>
<td></td>
</tr>
<tr>
<td>13:12:29.96</td>
<td>TPISERV</td>
<td>Peer socket address - Family=0002 Port number=01024 IP address=9.67.56.81</td>
<td></td>
</tr>
<tr>
<td>13:12:31.68</td>
<td>TPISERV</td>
<td>Issuing SELECT with MAXSOC 4</td>
<td></td>
</tr>
<tr>
<td>13:12:31.69</td>
<td>TPISERV</td>
<td>Issuing SELECT with MAXSOC 8</td>
<td></td>
</tr>
<tr>
<td>13:12:31.72</td>
<td>TPISERV</td>
<td>Takesocket With old descriptor 5</td>
<td></td>
</tr>
<tr>
<td>13:15:25.29</td>
<td>TPIMAIN</td>
<td>TPIMAIN Modified to STOP - we close down.</td>
<td></td>
</tr>
<tr>
<td>13:15:25.35</td>
<td>TPISERV</td>
<td>TPISERV Woke up</td>
<td></td>
</tr>
<tr>
<td>13:15:25.36</td>
<td>TPISERV</td>
<td>TPISERV is shutting down</td>
<td></td>
</tr>
<tr>
<td>13:15:25.85</td>
<td>TPISERV</td>
<td>TPISERV Going to sleep.</td>
<td></td>
</tr>
</tbody>
</table>

The first column is a timestamp column. The second column is the name of the module that requested the line printed on the log. Third column is an internal message number. The remaining part of a log line is free format.

Note the sequence of events around the client connection:

1. Main task is posted in its `select`
2. Main task issues `accept`
3. Main task issues `givesocket` and posts subtask to start processing.
4. Main task enters a new `select`
5. Subtask wakes up and issues a `takesocket`
6. Main task is again posted in its `select`
7. Main task closes the socket it gave to the subtask.
8. Main task issues a new `select`
9. Subtask goes on processing the client request.

### I.0 Appendix I. Sample Compile and Link JCL Procedures
This appendix contains the compilation and link procedures that were used to compile and link the sample programs in this book.

The procedures use an ITSO utility program called JCLTEST. This program compares two comma-separated strings that are passed in the PARM field. If the strings are equal, it returns a return code of zero. We used this program to set return codes to be used by the conditional JCL statements. Using this technique, we avoided maintaining four different procedures per language; but we were able to package all combinations of SQL, CICS and code without SQL or CICS into one procedure per language.

I.1 Assemble JCL Procedure
I.2 COBOL Compile JCL Procedure
I.3 C/370 Compile JCL Procedure
I.4 Link/Edit JCL Procedure

I.1 Assemble JCL Procedure

```c
//TCPASM PROC MEMBER=TEMPNAME,
// USER=TCPIP,
// MLQ=ITSC,
// SUFFIX=1$,
// WSPC=500,
// DB2=
// TCPMLQ=V3R1M0,
// OUTC='*',
// WORK=SYSDA,
// ASMPARM='OBJECT,NODECK,NOXREF'

//******************************************************************
//* *
//* TCP/IP MVS V3R1 - ITSO, Raleigh *
//* *
//* Assemble an Assembler module *
//* ---------------------------- *
//* *
//* Input: user.mlq.ASM(member) *
//* Macros: user.mlq.ASM *
//* Object deck: user.mlq.OBJ(member) *
//* *
//* MEMBER Module member name *
//* USER HLQ for source, object and list datasets *
//* MLQ MLQ for source, object and list datasets *
//* DB2 Specify DB2=YES if source includes SQL stmt's *
//* CICS Specify CICS=YES if source includes CICS stmt's *
//* TCPMLQ TCP/IP dataset MLQ *
//* SUFFIX CICS translator module suffix *
//* ASMPARM Assembler parameters *
//* OUTC Output class for SYSOUT *
//* WORK Work UNIT *
//* *
//******************************************************************

//DB2TEST EXEC PGM=JCLTEST,PARM='YES,&DB2.'
//STEPLIB DD DSN=TCPIP.ITSC.LOAD,DISP=SHR

// IF (DB2TEST.RC=0) THEN *** Include DB2 Precompile ***
//
//DB2PRE EXEC PGM=DSNHPC,PARM='HOST(ASM),TWOPASS',REGION=4096K
//DBRMLIB DD DSN=USER..&MLQ..DBRMLIB.DATA(&MEMBER),DISP=SHR
//STEPLIB DD DSN=SYS1.DSN230.DSNEXIT,DISP=SHR
// DD DSN=SYS1.DSN230.DSNLOAD,DISP=SHR
```
A Beginner's Guide to MVS TCP/IP Socket Programming
A Beginner's Guide to MVS TCP/IP Socket Programming

I.2 COBOL Compile JCL Procedure

//TCPCOB  PROC SUFFIX=1$,
//   DB2=,
//   CICS=,
//   MEMBER=,
//   USER=TCPIP,
//   MLQ=ITSC,
//   WSPC=500,
//   OUTC='*','
//   WORK=SYSDA

I.2.1 A Beginner's Guide to MVS TCP/IP Socket Programming

//DB2TEST EXEC PGM=JCLTEST,PARM='YES,&DB2.'
//STEPLIB DD DSN=TCPIP.ITSC.LOAD,DISP=SHR
/*
   IF (DB2TEST.RC=0) THEN *** DB2 Precompile ***
*/
//DB2PRE EXEC PGM=DSNHPC,PARM='HOST(COB2),APOST',REGION=4096K
//DBRMLIB DD DSN=&USER..&MLQ..DRBMLIB.DATA(&MEMBER.),
//   DISP=SHR
//STEPLIB DD DSN=SYSLIB.DSN230.DSNEXIT,DISP=SHR
// DD DSN=SYSLIB.DSN230.DSNLOAD,DISP=SHR
//SYSCIN DD DSN=&DSNHOUT,DISP=(NEW,PASS),UNIT=&WORK.,
// SPACE=(800,(&WSPC,&WSPC))
//SYSLIB DD DSN=&USER..&MLQ..SRCLIB.DATA,DISP=SHR
//SYSPRINT DD SYSDA
//SYSTERM DD SYSDA
//SYSUDUMP DD SYSDA
A Beginner's Guide to MVS TCP/IP Socket Programming

//SYSIN DD DSN=&USER..&MLQ..COBOL(&MEMBER.),DISP=SHR
/**
// ELSE
*** No DB2 Precompile, copy input ***
/**
//DB2COPY EXEC PGM=IEBGENER
//SYSUT1 DD DSN=&USER..&MLQ..COBOL(&MEMBER.),DISP=SHR
//SYSUT2 DD DSN=&DSNHOUT,DISP=(NEW,PASS),UNIT=&WORK.,
// SPACE=(800,(&WSPC,&WSPC))
//SYSIN DD DUMMY
//SYSPRINT DD DUMMY
/**
// ENDF
*** End of DB2 Precompile section ***
/**
//CICSTEST EXEC PGM=JCLTEST,PARM='YES,&CICS.'
//STEPLIB DD DSN=TCPIP.ITSC.LOAD,DISP=SHR
/**
// IF (CICSTEST.RC = 0) THEN *** CICS Translation ***
/**
//CICSTRN EXEC PGM=DFHECP&SUFFIX,
// PARM='COBOL2',
// REGION=4096K
//STEPLIB DD DSN=CICS.SDFHLOAD,DISP=SHR
//SYSPRINT DD SYSOUT=&OUTC
//SYSPUNCH DD DSN=&SYSCIN,
// DISP=(NEW,PASS),UNIT=&WORK,
// DCB=BLKSIZE=400,
// SPACE=(400,(400,100))
//SYSIN DD DSN=&DSNHOUT,DISP=(OLD,DELETE)
/**
// ELSE
*** No CICS Translation, copy input ***
/**
//CICSCOPY EXEC PGM=IEBGENER
//SYSUT1 DD DSN=&DSNHOUT,DISP=(OLD,DELETE)
//SYSUT2 DD DSN=&SYSCIN,
// DISP=(NEW,PASS),UNIT=&WORK,
// DCB=BLKSIZE=400,
// SPACE=(400,(400,100))
//SYSIN DD DUMMY
//SYSPRINT DD DUMMY
/**
// ENDF
*** End of CICS Translation section ***
/**
//* COBOL2 Compile source module into an object deck.
//*
//COBII EXEC PGM=IGYCRCTL,REGION=4096K,
// PARM='NODYNAM,LIB,OBJECT,RENT,RES,APOST'
//STEPLIB DD DSN=COB2.COB2COMP,DISP=SHR
//SYSLIB DD DSN=CICS.SDFHC OB,DISP=SHR
// DD DSN=CICS.USER.SDFH LOAD,DISP=SHR
//SYSPRINT DD SYSOUT=&OUTC.
//SYSTEMD DD SYSOUT=&OUTC.
//SYSIN DD DSN=&SYSCIN,DISP=(OLD,DELETE)
//SYSLIN DD DSN=&USER..&MLQ..OBJ(&MEMBER.),DISP=SHR
//SYSUT1 DD UNIT=&WORK,SPACE=(460,(350,100))
//SYSUT2 DD UNIT=&WORK,SPACE=(460,(350,100))
//SYSUT3 DD UNIT=&WORK,SPACE=(460,(350,100))
//SYSUT4 DD UNIT=&WORK,SPACE=(460,(350,100))
//SYSUT5 DD UNIT=&WORK,SPACE=(460,(350,100))
//SYSUT6 DD UNIT=&WORK,SPACE=(460,(350,100))
//SYSUT7 DD UNIT=&WORK,SPACE=(460,(350,100))
//SYSUT8 DD UNIT=&WORK,SPACE=(460,(350,100))
/**
A Beginner's Guide to MVS TCP/IP Socket Programming

1.3 C/370 Compile JCL Procedure

/* MVS Binder step without resolving external references */

//LKEDOBJ EXEC PGM=IEWL,REGION=4096K,
// PARM='NCAL,LET'
//SYSMOD DD DSN=&USER..&MLQ..LOAD(&MEMBER.),DISP=SHR
//SYSUT1 DD UNIT=&WORK,DCB=BLKSIZE=1024,
// SPACE=(1024,(200,20))
//SYSPRINT DD SYSOUT=&OUTC.
//SYSLIN DD DSN=&USER..&MLQ..OBJ(&MEMBER.),DISP=SHR
//PEND

I.3 C/370 Compile JCL Procedure

//TCPC370 PROC MEMBER=,
// USER=TCPIP,
// MLQ=ITSC,
// SUFFIX=I$,
// DB2=,
// CICS=,
// WORK=SYSDA,
// C370MLQ=V2R1M0,
// PLIMLQ=V2R3M0,
// TCPMLQ=V3R1M0,
// CPARM='DEF(MVS),SOURCE',
// WSPC=500,
// OUTC='*
//******************************************************************
//* ******** TCP/IP MVS V3R1 - ITSO, Raleigh ********
//* Compile a C source module
//* ------------------------------------------
//* Input: user.mlq.C(member)
//* Header files: user.mlq.H
//* Object deck: user.mlq.OBJ(member)
//* MEM
//* USER 
//* MLQ
//* SUFFIX
//* DB2
//* CICS
//* SUFFIX
//* OUTC
//* WORK
//*******************************************************************
//*
/* DB2TEST EXEC PGM=JCLTEST,PARM='YES,&DB2.'
//STEPLIB DD DSN=TCPIP.ITSC.LOAD,DISP=SHR
//*
/* IF (DB2TEST.RC=0) THEN *** DB2 Precompile ***
/*
//DB2PRE EXEC PGM=DSNHPC,PARM='HOST(C)',REGION=4096K
//DBRMLIB DD DSN=&USER..&MLQ..DBRMLIB.DATA(&MEMBER.),DISP=SHR
//STEPLIB DD DSN=SYS1.DSN230.DSNEXIT,DISP=SHR
A Beginner's Guide to MVS TCP/IP Socket Programming

/ DD DSN=SYS1.DSN230.DSNLOAD,DISP=SHR
/ SYSCIN DD DSN=&&DSNHOUT,DISP=(NEW,PASS),UNIT=&WORK.,
/ SPACE=(800,(&WSPC,&WSPC))
/ SYSLIB DD DSN=USER..&MLQ..SRCLIB.DATA,DISP=SHR
/ SYSPRINT DD SYSOUT=&OUTC.
/ SYSTEM DD SYSOUT=&OUTC.
/ SYSUDUMP DD SYSOUT=&OUTC.
/ SYSUT1 DD SPACE=(800,(&WSPC,&WSPC),,,ROUND),UNIT=&WORK.
/ SYSUT2 DD SPACE=(800,(&WSPC,&WSPC),,,ROUND),UNIT=&WORK.
/ SYSSIN DD DSN=USER..&MLQ..C(&MEMBER.),DISP=SHR

/*
** ELSE
*** No DB2 Precompile, copy input ***
*/
/ DB2COPY EXEC PGM=IEBGENER
/ SYSUT1 DD DSN=USER..&MLQ..C(&MEMBER.),DISP=SHR
/ SYSUT2 DD DSN=&&DSNHOUT,DISP=(NEW,PASS),UNIT=&WORK.,
/ SPACE=(800,(&WSPC,&WSPC))
/ SYSSIN DD DUMMY
/ SYSPRINT DD DUMMY
/*
** ENDIF *** End of DB2 Precompile section ***
*/
/ CICSTEST EXEC PGM=JCLTEST,PARM='YES,&CICS.'
/ STEPLIB DD DSN=TCPIP.ITSC.LOAD,DISP=SHR
/*
** IF (CICSTEST.RC = 0) THEN *** CICS Translation ***
*/
/ CICSTRN EXEC PGM=DFHEDP& SUFFIX,
/ REGION=4096K
/ STEPLIB DD DSN=CICS.SDFHLOAD,DISP=SHR
/ SYSPRINT DD SYSOUT=&OUTC
/ SYSPUNCH DD DSN=&&SYSCIN,
/ DISP=(NEW,PASS),UNIT=&WORK,
/ DCB=BLKSIZE=400,
/ SPACE=(400,(400,100))
/ SYSSIN DD DSN=&&DSNHOUT,DISP=(OLD,DELETE)
/*
** ELSE
*** No CICS Translation, copy input ***
*/
/ CICSCOPY EXEC PGM=IEBGENER
/ SYSUT1 DD DSN=&&DSNHOUT,DISP=(OLD,DELETE)
/ SYSUT2 DD DSN=&&SYSCIN,
/ DISP=(NEW,PASS),UNIT=&WORK,
/ DCB=BLKSIZE=400,
/ SPACE=(400,(400,100))
/ SYSSIN DD DUMMY
/ SYSPRINT DD DUMMY
/*
** ENDIF *** End of CICS Translation section ***
*/
/*
** COMPILC EXEC PGM=EDCCOMP,
** PARM=('&CPARM'),
** REGION=4096K
/ STEPLIB DD DSN=C370.&C370MLQ..SEDCLINK,DISP=SHR
/ DD DSN=PLI..&PLIMLQ..SIBMLINK,DISP=SHR
/ DD DSN=C370.&C370MLQ..SEDCCOMP,DISP=SHR
/ SYSLIB DD DSN=TCPIP..TCPMLQ..SEZACMAC,DISP=SHR
/ DD DSN=C370.&C370MLQ..SEDCHDRS,DISP=SHR,DCB=(BLKSIZE=3120)
/ USERLIB DD DSN=USER..&MLQ..H,DISP=SHR
/ SYSSIN DD DSN=&&SYSCIN,DISP=SHR
I.4 Link/Edit JCL Procedure

//TCPLINK PROC MEMBER=,
//  USER=TCP/IP,
//  MLQ=ITSC,
//  DB2=,
//  CICS=,
//  IMS=,
//  COBOL=,
//  ASM=,
//  C370=,
//  TCPMLQ=V3R1M0,
//  PLIMLQ=V2R3M0,
//  C370MLQ=V2R1M0,
//  OUTC='*','
//  LNKPARM='XREF,LIST',
//  WORK=SYSDA
//**************************************************************************
// *
/* TCP/IP MVS V3R1 - ITSO, Raleigh */
/* */
/* Bind (Link/Edit) a program */
/* */
/* -------------------------- */
/* */
/* Syslib: user.mlq.OBJ */
/* Load module: user.mlq.LOAD(member) */
/* */
/* MEMBER Module member name */
/* USER HLQ for source, object and list datasets */
/* MLQ MLQ for source, object and list datasets */
/* DB2 Specify DB2=YES if program uses SQL */
/* CICS Specify CICS=YES if program runs in CICS */
/* IMS Specify IMS=YES if program runs in IMS */
/* COBOL Specify COBOL=YESL if program includes COBOL */
/* ASM Specify ASM=YES if program includes ASM */
/* C370 Specify C370=YES if program includes C/370 */
/* TCPMLQ TCPIP dataset MLQ */
/* LNKPARM Linkage editor parameters */
/* OUTC Output class for SYSOUT */
/* WORK Work UNIT */
/* */
/* If you need to pass SYSLIN input to the Binder, override */
/* */
/* SYSIN with stepname LKEDCICS for CICS and LKEDBAT for */
/* non-CICS link jobs */
/* */
/* //LKEDCICS.SYSIN DD * - for CICS links */
/* or */
/* //LKEDBAT.SYSIN DD * - for non-CICS links */
/* */
/* *********************************************************** */
/* */
/* ALLOCW EXEC PGM=IEBGENER */
/* SYSUT1 DD DSN=NULLFILE,DCB=(RECFM=FB,LRECL=80,BLKSIZE=800) */
/* SYSUT2 DD DSN=&&WORKDS,DISP=(,PASS),UNIT=&WORK, */
/* SPACE=(TRK,(2,1)), */
/* DBC=(RECFM=FB,LRECL=80) */
/* SYSIN DD DUMMY */
/* SYSPRINT DD SYSOUT=* */
/* */
/* Test for program conditions - set RC's for later logic */
/* */
/* TESTCICS EXEC PGM=JCLTEST,PARM='YES,&CICS.' */
/* STEPLIB DD DSN=TCP/IP.ITSC.LOAD,DISP=SHR */
/* */
/* TESTCOB EXEC PGM=JCLTEST,PARM='YES,&COBOL.' */
/* STEPLIB DD DSN=TCP/IP.ITSC.LOAD,DISP=SHR */
/* */
/* TESTC370 EXEC PGM=JCLTEST,PARM='YES,&C370.' */
/* STEPLIB DD DSN=TCP/IP.ITSC.LOAD,DISP=SHR */
/* */
/* TESTDB2 EXEC PGM=JCLTEST,PARM='YES,&DB2.' */
/* STEPLIB DD DSN=TCP/IP.ITSC.LOAD,DISP=SHR */
/* */
/* TESTASM EXEC PGM=JCLTEST,PARM='YES,&ASM.' */
/* STEPLIB DD DSN=TCP/IP.ITSC.LOAD,DISP=SHR */
/* */
/* TESTIMS EXEC PGM=JCLTEST,PARM='YES,&IMS.' */
/* STEPLIB DD DSN=TCP/IP.ITSC.LOAD,DISP=SHR */
/* */
/* If this is a CICS program, we need language dependent */
/* CICS stubs for both CICS and socket interface */

A Beginner's Guide to MVS TCP/IP Socket Programming

329
A Beginner's Guide to MVS TCP/IP Socket Programming

/*
 * IF (TESTCICS.RC = 0) THEN
 * IF (TESTC370.RC = 0) THEN
 */
CICSC370 EXEC PGM=IEBGENER
/SYSUT1 DD DSN=CICS.SDFHC370(DFHEILID),DISP=SHR
/DD DSN=TCP/IP.ITSC.CNTL(EZACIC07),DISP=SHR
/SYSUT2 DD DSN=&&WORKDS,DISP=(MOD,PASS)
/SYSIN DD DUMMY
/SYSPRINT DD DUMMY
/*
 * ENDIF
 */
IF (TESTCOB.RC = 0) THEN
/
CICSCOB EXEC PGM=IEBGENER
/SYSUT1 DD DSN=CICS.SDFHC370(DFHEILIC),DISP=SHR
/DD DSN=TCP/IP.ITSC.CNTL(EZACICAL),DISP=SHR
/SYSUT2 DD DSN=&&WORKDS,DISP=(MOD,PASS)
/SYSIN DD DUMMY
/SYSPRINT DD DUMMY
/*
 * ENDIF
 */
IF (TESTASM.RC = 0) THEN
/
CICSAASM EXEC PGM=IEBGENER
/SYSUT1 DD DSN=CICS.SDFHMACH(DFHEILIA),DISP=SHR
/DD DSN=TCP/IP.ITSC.CNTL(EZACICAL),DISP=SHR
/SYSUT2 DD DSN=&&WORKDS,DISP=(MOD,PASS)
/SYSIN DD DUMMY
/SYSPRINT DD DUMMY
/*
 * ENDIF
 */
IF (TESTDB2.RC = 0) THEN
/
IF (TESTC370.RC = 0) THEN
/
DB2C370 EXEC PGM=IEBGENER
/SYSUT1 DD DSN=TCP/IP.ITSC.CNTL(DSNDLI),DISP=SHR
/SYSUT2 DD DSN=&&WORKDS,DISP=(MOD,PASS)
/SYSIN DD DUMMY
/SYSPRINT DD DUMMY
/*
 * ENDIF
 */
IF (TESTCOB.RC = 0) THEN
/
DB2COB EXEC PGM=IEBGENER
/SYSUT1 DD DSN=TCP/IP.ITSC.CNTL(DSNCLI),DISP=SHR
/SYSUT2 DD DSN=&&WORKDS,DISP=(MOD,PASS)
/SYSIN DD DUMMY
/SYSPRINT DD DUMMY
/*
 * ENDIF
 */
IF (TESTASM.RC = 0) THEN
/
DB2ASM EXEC PGM=IEBGENER
/SYSUT1 DD DSN=TCP/IP.ITSC.CNTL(DSNALI),DISP=SHR
/SYSUT2 DD DSN=&&WORKDS,DISP=(MOD,PASS)
/SYSIN DD DUMMY
//SYSPRINT DD DUMMY
//*
// ENDIF
//*
//* If program includes DL/I calls, we need a language
//* independent stub
//*
//* IF (TESTIMS.RC = 0) THEN
//*
//COPYIMS EXEC PGM=IEBGENER
//SYSUT1 DD DSN=TCPPIP.ITSC.CNTL(DPSLI000),DISP=SHR
//SYSUT2 DD DSN=&WORKDS,DISP=(MOD,PASS)
//SYSIN DD DUMMY
//SYSPRINT DD DUMMY
//*
// ENDIF
//*
//* We need different SYSLIB setup for CICS and non-CICS
//* environments.
//*
//* IF (TESTCICS.RC = 0) THEN
//*
//LKEDCICS EXEC PGM=IEWL,REGION=4096K,
// PARM='&LNKPARM.'
//SYSLIB DD DSN=&USER..&MLQ..LOAD,DISP=SHR
// DD DSN=TCPPIP.&TCPMLQ..SEZACMTX,DISP=SHR
// DD DSN=TCPPIP.&TCPMLQ..SEZATCP,DISP=SHR
// DD DSN=SYS1.TCPIP.&TCPMLQ..SEZALINK,DISP=SHR
// DD DSN=C370.&C370MLQ..SEDCBASE,DISP=SHR
// DD DSN=PLI.&PLIMLQ..SIBMBASE,DISP=SHR
// DD DSN=CICS.SDFHLOAD,DISP=SHR
// DD DSN=COB2.COB2CICS,DISP=SHR
// DD DSN=COB2.COB2LIB,DISP=SHR
// DD DSN=SYS1.DSN230.DSNLOAD,DISP=SHR
// DD DSN=IMS.RESLIB,DISP=SHR
//SYSLMOD DD DSN=&USER..&MLQ..LOAD(&MEMBER.),DISP=SHR
//SYSUT1 DD UNIT=&WORK,DCB=BLKSIZE=1024,
// SPACE=(1024,(200,20))
//SYSPRINT DD SYSOUT=&OUTC.
//SYSLIN DD DSN=&WORKDS,DISP=(OLD,DELETE)
// DD DSN=&USER..&MLQ..OBJ(&MEMBER.),DISP=SHR
// DD DDNAME=SYSIN
//SYSIN DD DUMMY
//*
// ELSE
//*
//LKEDBAT EXEC PGM=IEWL,REGION=4096K,
// PARM='&LNKPARM.'
//SYSLIB DD DSN=C370.&C370MLQ..SEDCBASE,DISP=SHR
// DD DSN=PLI.&PLIMLQ..SIBMBASE,DISP=SHR
// DD DSN=TCPPIP.&TCPMLQ..SEZACMTX,DISP=SHR
// DD DSN=TCPPIP.&TCPMLQ..SEZATCP,DISP=SHR
// DD DSN=SYS1.TCPIP.&TCPMLQ..SEZALINK,DISP=SHR
// DD DSN=&USER..&MLQ..LOAD,DISP=SHR
// DD DSN=COB2.COB2LIB,DISP=SHR
// DD DSN=SYS1.DSN230.DSNLOAD,DISP=SHR
// DD DSN=IMS.RESLIB,DISP=SHR
//SYSLMOD DD DSN=&USER..&MLQ..LOAD(&MEMBER.),DISP=SHR
//SYSUT1 DD UNIT=&WORK,DCB=BLKSIZE=1024,
// SPACE=(1024,(200,20))
//SYSPRINT DD SYSOUT=&OUTC.
The link/edit procedure uses a number of SYSLIN files:

DFHEILID  C/370 CICS language stub

    INCLUDE SYSLIB(DFHELII)

EZACIC07  C-sockets library routines

    INCLUDE SYSLIB(EZACIC07)

DFHEILIC  COBOL CICS language stub

    INCLUDE SYSLIB(DFHECI)

EZACICAL  Sockets Extended CICS interface stub

    INCLUDE SYSLIB(EZACICAL)

DFHEILIA  Assembler CICS language stub

    INCLUDE SYSLIB(DFHEAI)

DSNDLI    C/370 SQL language stub

    INCLUDE SYSLIB(DSNDLI)

DSNCLI    COBOL SQL language stub

    INCLUDE SYSLIB(DSNCLI)

DSNALI    Assembler SQL language stub

    INCLUDE SYSLIB(DSNALI)

DFSLI000  DL/I Language stub

    INCLUDE SYSLIB(DFSLI000)

**ABBREVIATIONS** List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACEE</td>
<td>Accessor Environment Element</td>
</tr>
<tr>
<td>ACK</td>
<td>Acknowledgement Segment (TCP)</td>
</tr>
<tr>
<td>AF_INET</td>
<td>Addressing Family Internet</td>
</tr>
<tr>
<td>AF_IUCV</td>
<td>Addressing Family IUCV</td>
</tr>
<tr>
<td>AF_UNIX</td>
<td>Addressing Family UNIX</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>AIX</td>
<td>Advanced Interactive Executive</td>
</tr>
<tr>
<td>APF</td>
<td>Authorized Program Facility</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>APPC</td>
<td>Advanced Program to Program Communication</td>
</tr>
<tr>
<td>ARP</td>
<td>Address Resolution Protocol</td>
</tr>
<tr>
<td>AS</td>
<td>Address Space</td>
</tr>
<tr>
<td>BMP</td>
<td>Batch Message Program (IMS)</td>
</tr>
<tr>
<td>BMS</td>
<td>Basic Mapping Support (CICS)</td>
</tr>
<tr>
<td>BSD</td>
<td>Berkeley Software Distribution</td>
</tr>
<tr>
<td>CICS</td>
<td>Customer Information Control System</td>
</tr>
<tr>
<td>CPI-C</td>
<td>Common Programming Interface - Communications</td>
</tr>
<tr>
<td>CSM</td>
<td>Completed Status Message</td>
</tr>
<tr>
<td>DCE</td>
<td>Distributed Computing Environment</td>
</tr>
<tr>
<td>DDM</td>
<td>Distributed Data Management</td>
</tr>
<tr>
<td>DLI</td>
<td>Data Language One (IMS)</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name System</td>
</tr>
<tr>
<td>DPI</td>
<td>Distributed Programming Interface</td>
</tr>
<tr>
<td>DPL</td>
<td>Distributed Program Link (CICS)</td>
</tr>
<tr>
<td>DRDA</td>
<td>Distributed Relational Data Access</td>
</tr>
<tr>
<td>DST</td>
<td>Data Services Task</td>
</tr>
<tr>
<td>DTP</td>
<td>Distributed Transaction Processing (CICS)</td>
</tr>
<tr>
<td>ECB</td>
<td>Event Control Block</td>
</tr>
<tr>
<td>EIB</td>
<td>Execute Interface Block (CICS)</td>
</tr>
<tr>
<td>EOM</td>
<td>End Of Message</td>
</tr>
<tr>
<td>FIN</td>
<td>Finish Segment (TCP)</td>
</tr>
<tr>
<td>FTP</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>GLBD</td>
<td>Global Location Broker Daemon (server)</td>
</tr>
<tr>
<td>GTF</td>
<td>Generalized Trace Facility</td>
</tr>
<tr>
<td>IC</td>
<td>Interval Control (CICS)</td>
</tr>
<tr>
<td>ICMP</td>
<td>Internet Control Message Protocol</td>
</tr>
<tr>
<td>IDL</td>
<td>Interface Definition Language</td>
</tr>
</tbody>
</table>
A Beginner's Guide to MVS TCP/IP Socket Programming

**IMS**
Information Management System

**IP**
Internet Protocol

**IPCS**
Interactive Problem Control System

**ISN**
Initial Sequence Number

**ITSO**
International Technical Support Organisation

**IUCV**
Inter-User Communication Vehicle

**JES**
Job Entry Subsystem

**LAN**
Local Area Network

**LFS**
Logical File System (OpenEdition/MVS)

**LLBD**
Local Location Broker Daemon (server)

**LPD**
Line Printer Deamon (server)

**MFS**
Message Formatting Services (IMS)

**MIB**
Management Information Base

**MID**
Message Input Descriptor (IMS)

**MOD**
Message Output Descriptor (IMS)

**MPP**
Message Processing Program (IMS)

**MPR**
Message Processing Region (IMS)

**MQI**
Message Queueing Interface

**MQM**
Message Queue Manager

**MSS**
Maximum Segment Size

**MTU**
Maximum Transmission Unit

**NCK**
Network Computing Kernel

**NCS**
Network Computing System

**NDB**
Network Data Base

**NFS**
Network File System

**NIDL**
Network Interface Definition Language

**NRGLBD**
Non Replicated Global Location Broker Daemon (server)

**ONC**
Open Network Computing

**OSF**
Open Software Foundation

**PCB**
Program Communication Block (IMS)

**PCT**
Program Control Table (CICS)
Physical File System (OpenEdition/MVS)
Protocol Data Unit
Processing Program Table (CICS)
Program Specification Block (IMS)
Resource Access Control Facility
Reverse Address Resolution Protocol
Resource Control Table (CICS)
Record Descriptor Word
Request for Comments
Routing Information Protocol
Remote Procedure Call
Remote Shell Protocol
Request Status Message
Security Access Facility
Single Byte Character Set
Simple Mail Transfer Protocol
Simple Network Management Protocol
System Network Architecture
Structured Query Language
Synchronize Segment (TCP)
Task Control Block
Transmission Control Protocol
Transient Data (CICS)
Trivial File Transfer Protocol
Transaction Initiation Message
Transaction Request Message
Task Related User Exit (CICS)
User Data Protocol
Universal Unique Identifier
VM Communication Facility
Virtual Telecommunication Access Method
INDEX

Special Characters
*CSMOKY*  9.3.4
*REQSTS*  9.3.2  9.3.3
*TRNREQ*  9.3.2  9.3.3

A
abbreviations  ABBREVIATIONS
accept  3.4  5.7  6.6  11.1
accept, trace  11.4
ACEE  5.11.2
ACK segment, TCP  11.3
acknowledgement, TCP  11.3
acronyms  ABBREVIATIONS
active close  5.9
active socket  5.5
adapter (CICS)  10.2
address class  3.2.1
address space name  5.3
address space prefix (IMS)  9.3.3
addressing  3.2
addressing family  3.6
ADDRSPCFLX  9.3.3
AF_INET  3.6  3.6.1  5.4  5.5
AF_INET socket address  3.6
AF_IUCV  3.6
AF_IUCV socket address  3.6
AF_UNIX  3.6  3.6.1
AIBTDLI  9.2
AMDPRDMP  11.3
AnyNet/MVS  3.6.1
AnyNet/MVS_socket  3.6
apitype3  5.10
APPC  1.2.3  9.1  10.1
application model  1.2.1
application protocol  5.8  5.8.1
application split  1.2.1
ARP  3.1
ASCII  5.8.3  9.3.3  9.3.4
ASMADLI  9.2  9.3.4
assembler macro interface  4.4
assist module (IMS)  9.2  9.3.4
association  3.3  5.7
assortedparms  8.2
ASXBSENV  5.11.2
asynchronous  5.10
asynchronous call  4.4
asynchronous select  5.5
Athena widget set  2.4
attach server subtasks  6.4
authentication  5.11  9.3.1
authorization  5.11  5.11.2  9.3.1

B
backlog queue  5.6  11.4
basic mapping support  10.1
Berkeley Software Distribution 2.2
big endian 5.8.3
binary integers 5.8.3
bind 5.5 5.9 8.2
bind for clients 8.2
bind, trace 11.4
blocking 5.10
BMS 10.1
BSD 4.2
BSD sockets 2.2
buffer flushing 5.8.1
C
C-sockets 2.2 4.2
CADLI 9.2 9.3.4
call interface 4.3
CBLADLI 9.2 9.3.4
choosing an API 2.1
CICS listener 10.2
CICS OS/2 10.1
CICS sockets 10.1
CICS task number 5.3
CICS/6000 10.1
CICS/ESA 10.1
client 3.7
client ID 5.3 6.6.1 6.6.2 9.3.3
client ID in CICS 5.3
client ID of a C-socket program 5.3.1
client ID structure 5.3 5.3.2
client program 3.7.3 7.1
client use of bind 8.2
client/server design model 1.2.2
close 5.9 5.9.2 7.4
closing a connection 5.9
communications model 1.2.3
completed status message (IMS) 9.3.4
concurrent server 3.7.2 3.7.3 6.1
concurrent server child program 3.7.3 6.1
concurrent server main program 3.7.3 6.1
connect 3.4 7.3
connect loop 7.3
connect on datagram sockets 8.3
connected sockets 5.8.2
connection 3.3
connection-oriented 3.4
connectionless 3.4 8.1
conversation 1.2.3
conversational 1.2.3
cooperative applications 1.2
CPI-C 1.2.3 9.1 10.1
CSKD CICS transaction 10.2
CSKE CICS transaction 10.2
CSKL CICS transaction 10.2
CSM (IMS) 9.3.4
CSMOKY 9.3.4
D
data representation 5.8.3
data services task 6.2.2
dataglance 11.3
datagram socket 2.2 3.4
datagram sockets 8.1
datagram truncation 8.4
DCE/RPC 1.2.3 9.1 10.1
A Beginner's Guide to MVS TCP/IP Socket Programming

file descriptor 3.6.1
FIN segment 5.9
FIONBIO 5.10
fixed length messages 5.8.1
flags on recv and send calls 5.8.2
flow control 5.4
fork 6.1
fragmentation 3.5
full duplex 3.4
function shipping (CICS) 10.1

G
getclientid 5.3.2 7.2.1
getclientid, trace 11.4
gethostbyaddr 3.2.1
gethostbyaddress 7.3.1
gethostbyname 3.2.1 7.3
getservbyname 3.2.2
getsocket 5.4
givesocket 6.5.1
givesocket (IMS) 9.3.3
global storage area 4.4
GTF event identifier 11.3
GTF trace collection 11.3

H
half association 3.3 3.7
half close 5.9.1
high-level API 2.1
host entry structure 7.3.1
host name 3.2.1
host name list 3.2.1
host tables 3.2.1
htnl 5.8.3
htons 5.8.3

I
ICMP 3.1
identifying your socket program 5.3
implicit mode 9.2 9.3.2 9.3.4
implicit mode restrictions on client 9.3.4
IMS assist module 9.2
IMS listener 9.2 9.3
IMS sockets 9.1
IMS/AS 9.1
IMSERVER 9.3.3
IMSLSECX 9.3.1
INADDR_ANY 5.5 8.2
inet_addr 3.2.1
inet_ntoa 3.2.1
initapi 5.3 6.4 7.2
initapi (CICS) 10.3
initapi (IMS) 9.3.3
initapi, trace 11.4
initial sequence number, TCP 11.3
initialize 7.2
integrated socket 3.6.1
Inter-User Communication Vehicle 4.7
internet domain 5.4
internet domain socket 3.6
interval control (CICS) 10.3
ioctl 5.10
IP 3.1
IP addresses 3.2.1
IP datagram fragmentation 3.5
A Beginner’s Guide to MVS TCP/IP Socket Programming

IP header 3.5
IPCS 11.3
ISN 11.3
iterative server 3.7.1 3.7.3 5.2
IUCV socket API trace 11.4
IUCV sockets 4.7

J
jobname 5.3

K
kerberos 2.7 5.11.1

L
LFS 3.6.1
linger 5.9.2
linger time 5.9.2
linking CICS socket programs 10.4
listen 3.4 5.6
listen, trace 11.4
listener (CICS) 10.2
listener (IMS) 9.2 9.3
listener configuration data set (IMS) 9.3.2
listener security exit (IMS) 9.3.1 9.3.3
little endian 5.8.3
local socket 3.6
logical filesystem 3.6.1
low-level API 2.1

M
macro interface 4.4
manifest header file 4.2
maxdesc 5.3.1
maximum segment size, TCP 11.3
maxsno 5.3
maxsoc 5.3
message (IMS) 9.3
message design 5.8
message formatting services (IMS) 9.1 9.4
message input descriptor 9.4
message output descriptor 9.4
message queuing 1.2.3
message type identifier 5.8.1
messages in a stream 5.8
MFS 9.1 9.4
MFS formatting options 9.4
MID 9.4
MOD 9.4
MORETRACE SOCKET command 11.4
Motif 2.4
MQI 1.2.3 9.1 10.1
MQM 1.2.3 9.1 10.1
MQSeries 1.2.3
MsgHi 11.4
MsgLo 11.4
MSS 11.3
MTU 3.5
multihomed host 3.2.1 5.5
multitasking 4.4 6.1
multithreaded 4.4 6.1

N
name server 3.2.1
named socket 5.4
NCS/RPC 1.2.3 2.3
network analyzer 11.3
network byte order 5.8.3
network socket 3.6
NFS 1.2.1
non-blocking 5.10 11.1
non-connected sockets 5.8.2
NOTRACE command 11.3
NOTRACE SOCKET command 11.4
noudpqueuelimit 8.2
ntoi 5.8.3
nito 5.8.3
null-terminated string 3.2.1
O
OBEYFILE command 11.2 11.3 11.4
obtain a socket 5.4
ONC/RPC 1.2.3 2.3
OpenEdition/MVS 3.6 3.6.1
OpenEdition/MVS socket 3.6.1
OSF/Motif widget set 2.4
P
packet trace 11.3
paradigm 1.1
parallel processing 1.2.2
Pascal sockets 4.6
passive close 5.5
passive socket 5.6
password 5.11.1
pathname 3.6
peek flag 5.8.1
peeking into the buffer 5.8.1
peer-to-peer 1.2.2
pending activity 5.10 6.5
pending exception 6.5 6.6.2
pending read 6.5
pper 4.2
PFS 3.6.1
physical filesystem 3.6.1
PING 3.4
pkttrace 11.3
PL/I 4.3.1
PLIADLI 9.2 9.3.4
port number 3.2.2 3.3 5.5
PrmMsg 11.4
process 3.7.3
processing flags 5.8.2
processor pool 1.2.2
protocol 3.1
protocol layers 3.1
protocol stack 3.1
pseudo abend (IMS) 9.5
R
RACF 5.11.1
RACROUTE 5.11.1 5.11.2 6.2.4
RARP 3.1
raw socket 2.2 3.4
RDW's 5.8.1
read 5.8.2
reading data 5.8.2 8.4
receive, trace 11.4
receiving data 5.8
record descriptor words 5.8.1
records in a stream 5.8
recoverable resources (IMS) 9.5
recovery (IMS) 9.5
recv 5.8.1 5.8.2
recv loop 5.8.2
recvfrom 5.8.2 8.2
reentrant code 6.2.5
reliable protocol 3.4
remote procedure call design model 1.2.3
REQSTS 9.3.2 9.3.3
request status message (IMS) 9.3.2 9.3.3
reserving a port number 3.2.2
resolve host name 7.3
resolver 3.2.1
retcode 4.3 11.1
retcode on read and write calls 5.8.2
retcode zero 5.8.1
REXX sockets 2.2 4.5
RFC1006 2.5
RPC 2.3
RPC design model 1.2.3
RSM (IMS) 9.3.2 9.3.3
RSM reasoncode 9.3.2
RSM returncode 9.3.2
SAF 5.11.1
SCREEN command 11.4
screen scrapers 1.2.1
security 5.11 6.2.4
security exit (IMS) 9.3.1
segment 9.3
segment size 3.5
select 5.10 6.5 11.1
select mask 6.5
selectex 5.10
send 5.8.2
sending data 5.8 8.4
sendto 5.8.2 8.2
serializing access 6.2.1
server 3.7
setsockopt 5.9 5.9.2
shutdown 5.9.1
SNA LU6.2 1.2.3 9.1 10.1
sniffer 11.3
SNMP/DPI 2.6
SO_LINGER 5.9.2
SO_REUSEADDR 5.9
Socket 3.3 5.4 11.1
socket address 3.3 3.6 5.4
socket address structure 5.4 5.5 5.7 8.2
socket API trace 11.4
socket descriptor 3.3 3.6 3.6.1 5.4 5.7
socket descriptor number 5.4
socket library 3.6 3.6.1
socket number 3.3
socket programming interface 2.2
socket type 5.4
socket types 3.4
socket, trace 11.4
Sockets Extended 2.2 4.3 4.4
Sockets Extended assembler macro interface 4.4
Sockets Extended call interface 4.3
Sockets Extended functions 4.3
somaxconn 5.6
split 1.2.1
stack 2.1
starting server subtasks 6.4
stream boundaries 5.8
stream chopping techniques 5.8.1
stream concept 5.8
stream socket 2.2 3.4 5.8
subtask 5.3 7.2
subtask parameter in CICS 5.3
subtasking 6.1
SYN segment 11.3
SYN+ACK segments 11.3
SYNC 6.5
synchronize after asynchronous call 6.5
synchronizing updates (IMS) 2.5
SYSEDEBUG 11.4
T
takesocket 5.6.2 11.1
takesocket (CICS) 10.3
takesocket (IMS) 9.3.3
task management 6.2.3
task number in CICS 5.3
task related user exit (CICS) 10.2
task storage area 4.4
task-level security 5.2.4
TCBSENV 5.11.2
TCP 3.1 3.4
TCP buffer flushing 5.8.1
TCP connection sequence 11.3
TCP header 3.5
TCP segment 3.5
TCP window 11.3
TCP/IP protocol stack 3.1
TCPICICSERR error message 10.3
tcperrno header file 4.2
tcpperro header file 4.2
tcpname 5.3
termapi 7.5
test for pending activity 5.10
three-way TCP handshake 11.3
TIM (CICS) 10.3
TIM (IMS) 9.3.2 9.3.3
time-out logic 8.1
TIMEWAIT state 5.9
tn3270 9.1 10.1
TRACE PACKET command 11.3
TRACE SOCKET command 11.4
trace, application 11.2
tracing, API 11.4
transaction initiation message (CICS) 10.3
transaction initiation message (IMS) 9.3.2 9.3.3
transaction request message (CICS) 10.3
transaction request message (IMS) 9.3.2 9.3.3
transaction routing (CICS) 10.1
transient data (CICS) 10.3
TCRFMT 11.3
trgcls 11.4
TRM (CICS) 10.3
TRM (IMS) 9.3.2 9.3.3
TRNREQ 9.3.2 9.3.3
TRUE (CICS) 10.2
U
U4093 4.3.2
A Beginner's Guide to MVS TCP/IP Socket Programming

UDP  3.1  3.4
UDP header  3.5
UDP receive queue  8.2
UNIX domain socket  3.6
unnamed socket  5.4
unreliable  8.1
user abend 4093  4.3.2
user ID  5.11.1
V
variable length messages  5.8.1
VMCF  4.1
W
wait-for-input transactions (IMS)  9.3.4
well-known port  3.2.2
well-known service  3.2.2
WFI  9.3.4
window, TCP  11.3
workload management  6.2.3
write  5.8.2
writing data  5.8.2
X
x client  2.4
x server  2.4
X-Windows 1.2.1  2.4
X-Windows intrinsic functions  2.4
X-Windows toolkits  2.4
X-Windows widget set  2.4
X/Open Transport Interface  2.5
X11.4  2.4
XTI  2.5
xxxADLI  9.2  9.3.4

COMMENTS ITSO Technical Bulletin Evaluation RED000

A Beginner's Guide to MVS TCP/IP Socket Programming

Publication No. GG24-2561-00

Your feedback is very important to help us maintain the quality of ITSO Bulletins. Please print out this questionnaire, fill it out, and then 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
Copy this section to file and send it via VNET to: QUALITY @ WTSCPOK

Please rate on a scale of 1 to 5 the subjects below.
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction   
Organization of the book  
Accuracy of the information  
Relevance of the information  
Completeness of the information  
Value of illustrations  
Grammar/punctuation/spelling  
Ease of reading and understanding  
Ease of finding information  
Level of technical detail  
Print quality  

A Beginner's Guide to MVS TCP/IP Socket Programming 344
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!)

Address your comments to:

IBM International Technical Support Organization
Department 545, Building 657
P.O. BOX 12195
RESEARCH TRIANGLE PARK NC
USA 27709-2195

Name . . . . . . . . . _______________________________________________
Company or Organization _____________________________________________
Address . . . . . . . ________________________________________________
______________________________________________________________
______________________________________________________________
Phone No. . . . . . . . _______________________________________________