TCP/IP Sockets Conversion Cookbook

February 25, 1998

Communications Server for OS/390 Group

contact:
commserv390@vnet.ibm.com
RTP

Contents

Introduction

This document describes how you can convert between socket APIs. There are several reasons why a you should or must convert your applications either to OS/390 UNIX system services (OpenEdition or OE) sockets or to TCP/IP Sockets Extended APIs.

- The IUCV and VMCF TCP/IP APIs no longer exist in IP services in Communications Server for OS/390, Version 2 Release 5. However, applications using these APIs can continue to run on previous TCP/IP stacks, such as TCP/IP for MVS Version 3 Release 2.
- The OE C libraries are POSIX and XPG4 compliant, but the TCP/IP C libraries are not.
- Future TCP/IP enhancements will be implemented on the OE APIs but not necessarily on the TCP/IP APIs.

The purpose of this cookbook is to identify areas affecting existing applications that have been coded to the IBM TCP/IP APIs. In order to give you early insight to the methods of converting their applications, this cookbook currently has sections that are not complete. As you read sections that are incomplete or observe sections that are missing, please contact us so that we can add to or update this document. Our goal is to make your transition as easy as possible. Your input is a valuable part of the process.

This is a cookbook, not an IBM manual. As such, it will have sections from several people, each with their own unique style; giving you access to the information is a higher priority that conforming to standardized rules of documentation.

Essential Information

To understand what may have to be changed in applications, let's establish a groundwork of information that will help you make the correct decisions. The first step is to clarify the terminology. The next step involves describing the evolution of the TCP/IP APIs with respect to the levels of MVS and OS/390 releases. The final step is to describe all the modifications necessary to convert from one API to another.

Terminology

These terms will be used throughout this document. For those terms that require more than a few sentences to explain, we provide a separate section to address the topic. For brevity, we will use the BPX, OE, and SE acronyms throughout this document.

- IUCV API: a TCP/IP sockets API that uses IUCV, originally a VM system facility.
- VMCF API: a TCP/IP sockets API that uses VMCF, originally a VM virtual machine system facility.
- VMCF/IUCV address space: for AF_IUCV domain sockets and for pre-TCP/IP V3R2, the VMCF/IUCV address space was used for data tranfers and communications with applications through TCP/IP.
- AF_IUCV domain: a domain for TCP/IP C sockets used for InterProcess Communications (IPC) between apps on the same host. See the C sockets section for replacing AF_IUCV with AF_UNIX.
- AF_UNIX domain: a domain for OE sockets used for InterProcess Communications (IPC) between apps on the same host.
- RTL-RunTime Library. See the Compiler section for more information
- C/C++- OS/390 compiler. See the Compiler section for more information.
- OE: OpenEdition, aka Unix System Services (USS).
- BPX: the OE Assembler Callable Services. All functions start with "BPX."
- OS/390: Operating System. Formerly MVS, plus much more.
- SE API: the TCP/IP Sockets Extended macro and callable APIs.

APIs and Operating Systems

Table 1 (Page 1 of 2). APIs and OpSys

(op sys -->)	**MVS 3.1**	**MVS 4.3**	**MVS 5.2.2**	**OS/390R3**	**OS/390R4**	**OS/390R4**	**OS/390R5**
(tcp/ip -->)	V3R1	V3R2	V3R2	V3R2	V3R2	V3R3	V3R4
TCP/IP V3R1 C	Yes	Yes	Yes	Yes	Yes	-	-
TCP/IP V3R1 SE	Yes	Yes	Yes	Yes	Yes	-	Yes
TCP/IP V3R2 C	-	Yes	Yes	Yes	Yes	-	Yes
TCP/IP V3R2 SE	-	Yes	Yes	Yes	Yes	-	Yes

Table 1 (Page 2 of 2). APIs and OpSys							
(op sys -->)	**MVS 3.1**	**MVS 4.3**	**MVS 5.2.2**	**OS/390R3**	**OS/390R4**	**OS/390R4**	**OS/390R5**
TCP/IP OS/390R5	-	-	-	-	-	-	Yes
TCP/IP OS/390R5	-	-	-	-	-	-	Yes
OE C *	-	-	Yes	Yes	Yes	Yes	Yes
OE BPX *	-	-	Yes	Yes	Yes	Yes	Yes

Notes:

1. Some MVS and OS/390 systems can run multiple, concurrent different releases of TCP/IP
2. The second row (tcp/ip-->) indicates the TCP/IP running on the first row (op sys-->) operating system
3. The first column indicates apps that are built with the specified TCP/IP libraries
4. The OE APIs have evolved over several releases of OS/390. Check OE documents for other limitations per OS/390 release.

For example, look down the column marked "OS/390 R5, V3R4". On an OS/390 R5 system running TCP/IP V3R4, you cannot run an app that was linked with TCP/IP V3R1 C socket library, but you can run an app that was linked with the TCP/IP V3R1 SE library.

When To Convert

Let's start with some generalizations.

- New apps should use OE C and BPX socket APIs whenever possible
- OE C and BPX socket APIs are available on OS/390 systems
- OE C socket API is POSIX and XPG4 compliant
- A TCP/IP V3R1 SocketsExtended app will run on the broadest range of MVS and OS/390 systems
- OE BPX socket API on OS/390R5 is a superset of the TCP/IP SocketsExtended Macro API

There are several questions that must be answered to determine if/when a conversion of APIs is necessary.

- Which/how many MVS levels does my app need to run on?
- Which/how many TCP/IP levels does my app need to run on?
- What features does my app require?
  - POSIX compliant
  - Async
  - Multithreaded
  - Programming language
  - Portability without recompile/relink
  - BSD level
- What socket calls are needed?

Armed with this information, you can use a combination of the above generalizations and the "APIs and Operating Systems" section.

Terminology Explanations

These sections give additional information about the terms mentioned in the "Terminology" section above.

IUCV Sockets API

TCP/IP releases after TCP/IP V3R2 will not have a communications path to TCP/IP that uses the IUCV address space or the IUCV API. When the IUCV API was introduced in TCP/IP Version 2, it satisfied a customer need for a multithreaded API. IUCV API did not provide all the BSD socket functions, but a sufficient subset. Its down-side: it is based upon a VM concept that was ported to MVS. The SE API was introduced in TCP/IP V3R1 to provide the functionality of the IUCV API, but make the interface more generic and less VM-like. Under the covers, IUCV was used as the transport. In this way, as internal transports evolved, the external API could remain relatively constant.

There are some VM concepts that exist in the IUCV API that are unrelated to sockets: the system mask and control reg0 manipulation. Depending upon your use of these features for disabling IUCV interrupts, the complexity when converting from the IUCV API to either SE or BPX API will vary.

TCP/IP non-OE C socket API

There have been several releases of TCP/IP that run in an MVS or OS/390 environment. The socket APIs have evolved in each of these releases. This section describes the non-OE C socket API.

The non-OE C sockets API is BSD 4.3-based; the standard Berkeley socket functions are present and some new functions have been added in the MVS environment. These are carried over into the OS/390 environment, plus they have been added to the OE C sockets API for compatibility. Therefore, these functions will not be a problem when converting from non-OE C sockets to OE C sockets.

- getibmopt()
- getibmsockopt()
- givesocket()
- ibmsflush()
- maxdesc()
- setibmopt()
- setibmsockopt()
- sock_debug()
- sock_debug_bulk_perf0()
- sock_do_bulkmode()
- sock_do_teststor()
- tcperror()
- takesocket()

For explanations of these functions, see the TCP/IP API manual (SC31-7187).

The TCP/IP C sockets API is statically bound with an application during linkedit time. In TCP/IP V3R2, this is also true, but some internal modules are dynamically loaded. This allows an application built with

TCP/IP V3R2 C socket library to run on higher releases without recompiling or relinking. There are some exceptions to this rule. Here are a few:

- Raw mode sockets require AC(1) authorization. A relink may be necessary to do this.
- Some combinations of compilers and RTL specifications will cause an app to fail in an OS/390 environment. If your app was compiled with the AD/Cycle 1.2 compiler or higher, you should be fine. The RTL that comes with OS/390 R5 is a form of LE RTL. Prior RTLs (like the C/370 RTL) may not work.

See the Migration Guide for TCP/IP OS/390 R5 for all the specifics.

The TCP/IP C sockets API has remained the same since TCP/IP V3R1 with the exception of two new functions, getibmopt() and setibmopt(). Since the OE C sockets has the same functions as the TCP/IP C sockets, converting to the OE C sockets is a relatively minor task. See "Converting to OE C sockets" section for details.

TCP/IP non-OE SocketsExtended API

There have been several releases of TCP/IP that run in an MVS or OS/390 environment. The socket APIs have evolved in each of these releases. This section describes the non-OE SocketsExtended APIs.

The SE APIs come in two flavors - macro and callable. The callable API can be considered similar to the TCP/IP C socket API in that it is synchronous. The macro API is more robust since the functions can be issued in an asynchronous manner. It also has a concept called "apitype=3", which allows multiple concurrent outstanding requests on the same socket. One would consider this API as thread-enabled.

The SE API was introduced in TCP/IP V3R1 and enhanced in TCP/IP V3R2. Reference SC31-7187 for TCP/IP V3R2 for the specific details, but here is a brief summary of the V3R2 SE API enhancements:

- User exits added
- Several functions added (e.g., readv, writev)
- Default initapi if initapi was not issued prior to a TCP/IP communication socket call
- Modified format

Another interesting fact is that the SE API dynamically loads some lower function modules rather than having the SE library statically bound to the application at linkedit time. Because of this, an application built with the TCP/IP V3R2 SE libraries does not have to be relinked to run on higher TCP/IP releases. As a matter of fact, the same is true for a application built with a TCP/IP V3R1 SE library, although there is some internal conversion done to adjust the format of the parameters to match the TCP/IP V3R2 SE format.

In general, the TCP/IP V3R2 SE API can run on any MVS 4.3 or higher operating system, which makes it an attractive API for customers requiring to run an application on several MVS levels. The TCP/IP V3R1 SE API has an even broader range (starting at MVS 3.1); the tradeoff is fewer functions and a small additional internal conversion.

Compiler/RTL Levels

There has been an evolution of MVS compilers:

- C370 (used for TCP/IP V3R1)
- AD/Cycle (used for TCP/IP V3R2 and TCP/IP V3R3)
- C/C++ 3.1

- C/C++ 3.2
- C/C++ for OS/390 R4 (used for TCP/IP on OS/390 R5)

There has also been an evolution of RunTimeLibraries:

- C370
- LE 1.3 - 1.8
- RTL for OS/390 Rx

Fortunately, as we move to the OS/390 environment, we are converging to common names and matching components. Hence, the next TCP/IP release is officially called "TCP/IP for OS/390 R5", not "TCP/IP V3R4".

Another point of confusion is the fact that the C/C++ compiler has two distinct parts to it - the C part and the C++ part. Our TCP/IP product strictly uses the C part. We do not currently ship any C++ applications as part of our base product (but we do have some C++ additional applications).

OE BPX API

In OS/390 R4, the OE Assembler Callable Services (aka BPX) was enhanced to provide async support with exits driven in SRB mode. The next release will extend that support to provide exits in TCB mode similar to TCP/IP SocketsExtended macro API.

Conversions

Based upon the information described above, there are several combinations of conversions. For apps that are using the IUCV or VMCF APIs, the target API choices are:

- SE
- OE BPX
- OE C

Apps that are using TCP/IP C or SE sockets APIs might also consider converting. For these, the target API choices are:

- OE C
- OE BPX

There are also some apps that will not have to convert. The "When to Convert" section identifies the conditions that influence these decisions.

Different IBM customers have different needs. For example, if a customer has one mainframe, then an app that can run on several previous TCP/IP-MVS releases may not be a requirement. On the other hand, an ISV that supports customers across the spectrum of TCP/IPs and MVSs is looking for an API that span all these combinations. So, identify your requirements and then decide which APIs are appropriate for you.

Our goal is to provide APIs that satisfy the customer for both function and performance in a way that is least disruptive (time, dollars, ...). The ultimate OS/390 environment will see a convergence of the APIs into an assembler and a C version; namely, the OE APIs. In the meantime, we need to provide the means of getting you there in the least painful way.

IUCV API to BPX API

Introduction

This documentation is created to give user applications running IUCV native sockets a basic approach on how to convert to OpenEdition MVS (OE) sockets.

Generally, all application users follow the same IUCV native sockets syntax rules and protocols. However, the methodology of implementing IUCV native sockets could differ from one user application to another application because all user applications are not internally structured the same. For example, the application's tasking structure, task dispatchability, synchronous or asynchronous, and event completion notification (I/O completion) through an ECB or exit, should be taken into consideration when migrating to OE sockets.

Therefore, taking all applicable issues into consideration, it should be understood that the approach given by this documentation is very high level in scope and may not fully address all user application problems when converting to OE sockets.

If necessary , users are advised to restructure their application tasking or process systems to use the OE sockets interface efficiently.

There are two parts in this documentation:

- What users should do when planning to migrate from an IUCV based environment to an OE based environment.

- What users should do when converting a specific IUCV socket call to a corresponding OE socket call.

PART I

Background: native IUCV interface

Currently MVS IUCV provides a cross memory message passing facility which manages communication between User application API socket requests and the TCP/IP stack protocol. This cross memory message passing facility is the VMCF/IUCV subsystem layer. The VMCF and the IUCV together make up a communication management layer subsystem that runs in a completely different address space from the User's address space and the TCP/IP address space.

All socket requests and data flow through the VMCF/IUCV intermediate address space, which causes sever performance degradation between the user application and the TCP/IP stack.

The current high level IUCV interface is best illustrated by the table below. There are three operations that User applications must do to implement the native IUCV socket API interface.

Table 2. Simulated IUCV commands..
IUCV commands to perform three types of operations

Operation	IUCV command	Description
1. Register with IUCV to get services	`IUCVMINI SET...`	This operation is done only once
2. Create IUCV path to TCP/IP	`IUCV CONNECT....` `IUCVMCOM CONNECT...`	These sequence of commands are issued to create an IUCV path for each subtask(process) that is capable of issuing socket API call.
3. Issue API socket call to TCP/IP	`IUCV SEND...` `IUCVMCOM SEND....`	Send socket requests (connect,bind,listen,...) or data to TCP/IP over a given path.
4. Sever IUCV path to TCP/IP	`IUCV SEVER...` `IUCVMINI CLR...`	Sever IUCV path and detach subtasks(process)

Under IUCV, an application must first create an IUCV path connection between itself and the TCP/IP stack before sending any data or API socket request. The path serves as a channel connecting the application address space and the TCP/IP address space. Each path can handle up to 255 socket connections. This means that if the application wishes to make more than 255 socket connections, then it should create a second path connection. This path creation process continues until the maximum number of socket API connections that the application allows.

In OpenEdition socket interface environment, path creation is not needed and therefore applications migrating to OpenEdition sockets can easily delete process that currently handle IUCV path creation and termination (sever).

So, if we look at table 1, with operation 1 2 and 4 being deleted, we are left with operation 3. And operation 3 is what applications should focus on when converting IUCV socket API sockets to OpenEdition sockets.

MVS OpenEdition sockets

OpenEdition MVS provides support for a widely accepted protocol for client-server communication known as **sockets.**

OpenEdition sockets are a set of C language functions that closely correspond to the sockets used by UNIX applications that use the Berkeley Software Distribution (BSD). OpenEdition sockets support UNIX domain sockets and Internet domain sockets. The OpenEdition sockets discussed in this documentation is the Internet domain sockets, the AF_INET protocol family.

Migrating to OpenEdition MVS sockets.

IUCV application users should address the following issues when planning to convert from an IUCV environment to an OpenEdition environment.

- Defining a transport provider to OpenMVS (OMVS)
- Connecting user application to OpenMVS (OMVS)

- Using assembler callable services.
- Using asynchronous I/O (operation in SRB mode)

Defining a transport provider to OpenMVS (OMVS)

An OE application uses the OMVS kernel functions to perform socket calls. The OMVS kernel address space called the AF_INET physical files (PFS), then communicates with a TCP/IP address space or an AnyNet MVS address space to transport the socket call through the network. So, the OE application can use TCP/IP and/or AnyNet MVS as its transport provider, and a connection between the transport provider and OMVS must first be established before the application uses OMVS kernel functions to do socket calls. This is currently known as converged sockets because an OE application can use both MVS TCP/IP MVS AnyNet as its transport providers and can concurrently listen to incoming connections.

The OE application should specify which transport provider to use and the transport provider (TCP/IP or AnyNet) should be defined to OMVS. The transport provider is defined to OMVS by including the user name or proc name of the transport provider in the BPXPRMxx parmlib.

The transport provider, say in this case is TCPV33, then they way it should be defined in the BPXPRM.. library looks as follows:

```
FILESYSTYPE TYPE(CINET) ENTRYPOINT(BPXTCINT)
 NETWORK DOMAINNAME(AF_INET)
         DOMAINNUMBER(2)
         MAXSOCKETS(2000)
         TYPE(CINET)
         INADDRANYPORT(2000)
         INADDRANYCOUNT(100)

SUBFILESYSTYPE NAME(TCPV33)TYPE(CINET) ENTRYPOINT(EZBPFINI)
```

For more information on how to connect OMVS and a transport provider stack such as MVS TCP/IP or AnyNet MVS, users are advised to consult the OpenEdtion planning book.

Connecting user application to OpenMVS (OMVS)

To connect an application to OMVS, which is the kernel for OpenEdition, the application's address space must be known to OMVS. By making the application address space known to OMVS, the address space is said to be dubbed and is considered to be an OMVS process. Once dubbed, an application can receive OMVS services.

There are two ways of dubbing a task (in MVS) or a process (in OMVS):

- When a task invokes an OE socket call.
- When a task invokes the BPX1SDD service.

```
CALL BPX1SDD
        (DUBTHREAD,
         RETURN_VALUE,
         RETURN_CODE,
         REASON_CODE);
```

Dubbing an application's address space can be tricky and users must pay close attention as to why and how they should dub their applications. An application address space can have multiple tasks that can be dubbed separately. In OMVS environment, a dubbed task is treated as a unique process and the socket descriptors assigned to the process will also be all unique. This means that socket descriptors are all unique within a separately dubbed process. As was stated above, if an application has multiple tasks created and each task is separately dubbed then the socket descriptors assigned to the application cannot all be unique. There will be socket descriptors duplicated across tasks within the application address space and this will create difficulty in diagnosing and analyzing problems.

Unless it is necessary to let a task get dubbed when it invokes a socket call to OMVS, it is highly advised that users should invoke the BPX1SDD service with default dub setting to DUBTHREAD when the application opens it address space to MVS. What this OMVS service does is that it will dub the first (mother task) of the application's address space and all subsequent children subtasks are automatically dubbed. This dubbing method will cause the OMVS kernel to assign unique socket descriptors for the application and therefore there will be no duplicate descriptors within the application's address space. Socket descriptors will be assigned starting from zero to the maximum number of descriptors specified in the BPXPRMxx parmlib.

Assembler callable services (BPX1...)

Synchronous Calls

Sockets requiring synchronous operation must use the following format when calling OpenEdition assembler callable services.

```
CALL Service_name,(Parm_1,
                   Parm_2,
                   Parm-3,
                    .
                    .
                    .
                   Return_value,
                   Return_code,
                   Reason_code)
```

Parameters are positional and must be all coded within the parameter list

Values must be explicitly placed into parameters because callable services do not set defaults.

Applications directly calling assembler callable services must specify AMODE=31. This will ensure that linkage is done in 31-bit addressing mode.

For additional MVS OpenEdition sockets and details for socket call syntax, please reference the MVS/ESA Application Development Reference: assembler callable services for OpenEdition MVS (SC23-3020-02)

Asynchronous calls

The MVS OpenEdition has created a generic callable service for all socket calls requiring asynchronous operation. This callable service is called BXP1AIO. To perform asynchronous socket operation applications must follow the following format.

```
CALL BPX1AIO,(Aiocb_length,
              Aiocb,
              Return_value,
              Return_code,
              Reason_code)
```

The Aiocb is the main control block structure that is used for controlling the I/O operation. The Aiocb is mapped by BPXYAIO macro. The I/O operation is controlled by the values set into this control block and the caller is responsible for setting the desired values to control the I/O operation.

Currently, the Asynchio interface supports the following socket functions. All other functions not listed here should use synchronous interface by using the appropriate assembler callable service.

```
Read
Write
ReadV
WriteV
Rcv
Snd
RcvFrom
SndTo
RcvMsg
SndMsg
Accept
Connect
Cancel
Selpoll (select or poll)
```

For additional MVS OpenEdition sockets and details for socket call syntax, please reference the MVS/ESA Application Development Reference: assembler callable services for OpenEdition MVS (SC23-3020-02)

For asynchronous socket details, reference the ASYNCHIO - Async I/O for sockets publication.

The Select() call

Under IUCV environment, socket descriptors are assigned and maintained by an application. The application, after connecting to the well known port to listen() for incoming connections, issues select() call to monitor activity on a set of socket descriptors to see if any of the sockets is ready to do read,write and exception processing.

Monitoring an activity of a socket descriptor is done using a 32-bit string. Bit strings corresponding to specific descriptors are turned on and off to indicate that the descriptors are in use or not in use.

Managing socket descriptors using a 32-bit string is complex and often applications find it difficult implementing the IUCV select() call. Also part of the select() call, applications must do givesocket() call and takesocket() call to complete connections accepted from clients.

Under OpenEdition environment, socket descriptors are assigned and maintained by OMVS, and the select() is now replaced by a more efficient call, the selpoll(). The SelPoll() call is asynchronous and users can specifically indicate in the AIOCB if the call is for select or poll.

OpenEdition application users can use the SelPoll() call if they need to, but the OE accept() call can sufficiently be used for accepting connections from clients and there is no need of using selpoll() on top of that.

The accept() call is classified as asynchronous. But there is a mechanism in which it can be invoked as synchronous and blocking. What this means is that the accept() call goes into a wait state until a connection request is queued to the listening port for processing. To issue a synchronous accept() call, users need to set the AioASYNC bit in the AIOCB control block.

If you don't like struggling with SelPoll(), use accept(). We have used it and it works very well.

To use the accept() call effectively and efficiently, user applications must be capable of being concurrent servers. A Concurrent server is a processes in which there is one master task and multiple subtasks. The master task, after accepting the client request, creates a subtask for the request, or assign the request to an existing subtask. Once the request is given to a subtask, then the master task can continue to accept the next client's request.

The diagram below illustrates how a concurrent server uses accept() call to serve its clients. The master task is dedicated to only issue the accept() call and pass the rest of the connection in request to a subtask for further processing.

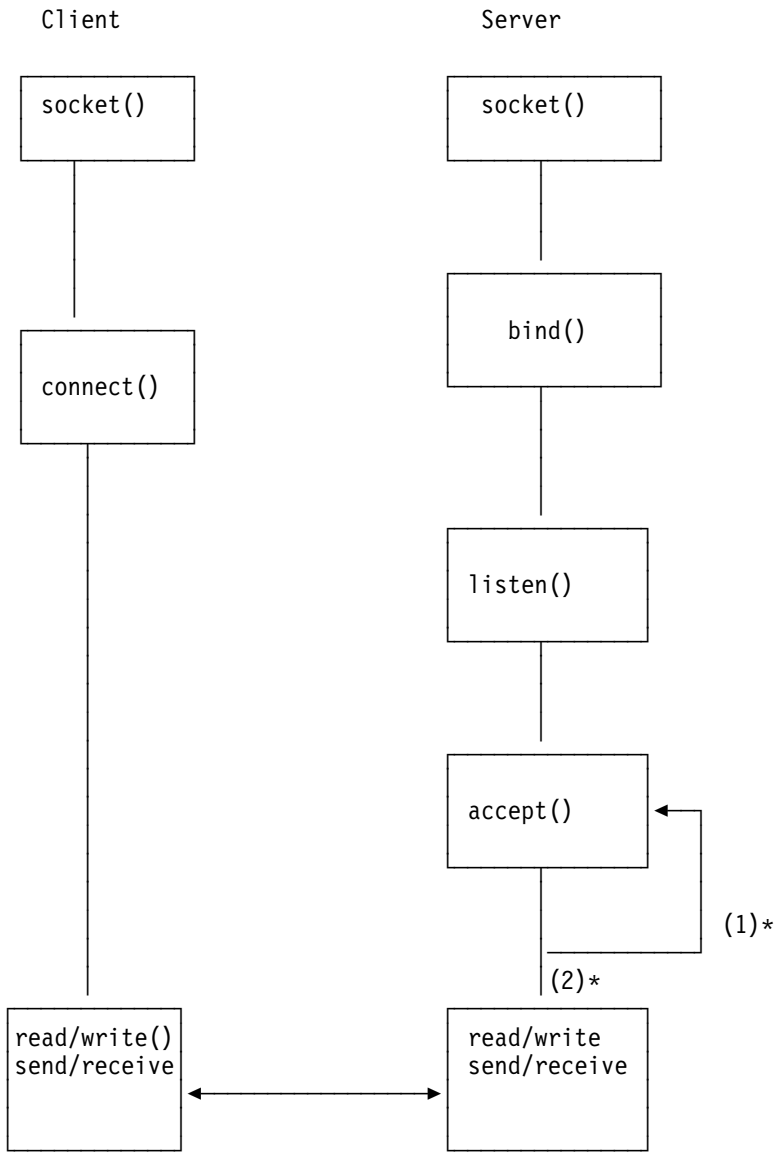

Figure 1. Concurrent server using sync ACCEPT()

NOTE: The arrow which is indicated by (1)* in diagram 1 is the main task dedicated to do only accept() function. Once the client connection request is met, then the main task assigns the rest of the connection

request process to another subtask(2)*. That way the main task does not waste time to get the next client waiting in the queue to connect.

Async I/O - BPX1AIO

The asyncio callable service gives application users the capability to issue socket calls asynchronously. Applications use BPXYAIO as the input when calling BPX1AIO. BPXYAIO is an MVS control block specifically designed to serve as an interface for applications using asyncio socket calls.

Socket calls supported under asyncio can be called from SRB mode routines using the same conventions as when called from task mode. However, when called from SRB mode the process associated with the running SRB must be a dubbed process. The SRB is associated to the running process by placing the adress of the OAPB to register 2, and the OAPB is obtained from the PRLI control block. The PRLI contains process related information. The PRLI is addressed as follows:
TcbStcb->StcbOtcb->OtcbThli->ThliPri->PrliOapb

Note that the routine calling socket call in SRB mode must place the address of OAPB in register 2.

When users dubb thier application, whichever method they use, the address of the OAPB can be saved in a storage in the application address space. This storage should be in a fixed location accessible to all tasks/threads within the address space. An example is shown below. Assume the address defined by the application to contain the OAPB is a field called USER_OAPB then:

USER_OAPB = PSATOLD->TCBSTCB->STCBOTCB->OTCBTHLI->THLIPRLI->PRLIOAPB

So, whenever a socket call is issued in SRB mode, the calling module must put the address pointed to by USER_OAPB in register 2 (GPR02), otherwise OMVS will issue an abend0C4.

The BPXYAIO contains an 8 byte field to be used by the application when issuing asyncio supported socket calls. The field is used for correlating requests and responses and is not used by OMVS.

There are two ways an application can be notified when an I/O is completed, through an exit routine or through an ECB. And the application must specify the address of the exit routine or the ECB in an Aiocb and at the same time tell OMVS that the notification desired is through an exit routine or an ECB. The notification cannot be both.

PART II

IUCV communication

As was mentioned in part I, section 3, an IUCV communication is described by the following steps:

- Register to use IUCV services
- Create iucv path to connect to TCP/IP address space
- Send socket API request to TCP/IP over a given IUCV path
- Terminate an IUCV path
- Unregister or detach from IUCV service

*

And to do the above steps, users used the following IUCV macros:

- IUCVMINI SET - to register with IUCV
- IUCV CONNECT - to convert IUCV connect request to IPARML format
- IUCVMCOM CONNECT - to create an IUCV connection to TCP/IP
- IUCV SEND - to convert a socket request to IPARML format
- IUCVMCOM SEND - to send socket request to TCP/IP over an IUCV connection
- IUCV SEVER - to sever an IUCV connection
- IUCVMINI CLR - to unregister and detach from IUCV

In addition, there are important aspects of IUCV that should be addressed when migrating to OE sockets environment. These include:

- the IUCV input parameter list, IPARML
- the external interrupt routine
- the external interrupt buffer, which receives message indicating success or failure of the socket request.

Remove IUCV communication

To achieve clean migration to OE sockets, users should remove all references to IUCV macros, IPARML, IUCVPTRS, message interrupt buffers, message completion messages and other IUCV related data areas.

The IUCV external interrupt routine, with some modifications, may be used in the asyncio OE socket connections. The asyncio I/O, Aiocb structure has a field in which the user can put the address of the interrupt routine. That way, OMVS knows where to get the address of the interrupt routine when it is ready to notify the user when an asynchronous event is completed.

IUCV socket vs OE socket syscalls

Basically there is no difference between an IUCV socket call and an OE socket call as far as functionality is concerned. The difference is the method used to communicate with TCP/IP. Under OE sockets, the communication is between the user and OMVS and the communication is direct using assembler callable services. The user has full control of the communication. Parameter declaration and parameter passing to assembler callable services are the responsibility of the user. Assembler callable services are directly called from the user's program and the calling mechanism follows standard MVS save area conventions.

Parameter passing for each socket call is unique and must be strictly followed. For all synchronous sockets, the assembler callable service book is a good reference material. For all asynchronous sockets the BPXYAIO macro should be used. The internal structure of BPXYAIO is defined as AIOCB. The AIOCB is a control block structure which is used as input to BPX1AIO for requests and output for responses. Because the AIOCB is reusable, it should be defined within a major control block. A good major control block to imbed the AIOCB in is a connection control block, which represents a TCP connection.

For migration purposes, the following commonly used socket calls are compared in both IUCV and OE environments.

First, there are many OE structures that should be used when issuing socket calls. Here are some of the most common structures that users should know how to use them correctly.

- BPXYSOCK - sockaddr structure, used for accept(),socket(),connect()
- BPXYMSGH - Buffer structure, used for sendmsg() and recvmsg()
- BPXYIOV - Buffer structure, used for writev() and readv()
- BPXYPOLL - Structure used for polling descriptors, used for SelPoll(
- BPXYCID - Client structure, used for getclientid()readv()
- BPXYAIO - Asynchronous structure (Aiocb), used for all Async I/O

ACCEPT()

IUCV

```
input parameters:
1- call number (accept)
2- IUCV path ID
3- prmmsg_hi, set to zero
4- New socket descriptor
   socket descriptor of new socket being created

output:
1- IUCV return code
2-  socket api return code
    (See MVS TCPIP Programming for return code lists)
3-  Error number for the API call, if the api return code
    is -1.
4 - Address of Client
```

OE

Accept() is asynchronous call and the input structure is Aiocb.

```
Input parameters:

1 - Aio#MVS, notify type must be either Exit or ECB
2 - AioFd, socket descriptor
3 - Aio#Accept, command type
4 - AioSockAddrPtr, sockaddr for clients address
5 - AioSockAddrLen, length of client's sockaddr

6 - AioSync, can be turned on to indicate the request is synchronous


call BPX1AIO
             (length(accept_aiocb),
              accept_aiocb,
              return_value,
              return_code,
              reason_code);

Output:
1 - Return_value = 1 if success or -1 if failed
2 - Return_code, non zero if return_value = -1
3 - Reason_code, non zero if return_value = -1
4 - AioRv = socket descriptor if return_value = 1
```

BIND()

IUCV

```
input parameters:
1 - Call number
2 - IUCV path ID
3 - Socket descriptor being used for the connection
4 - A sockaddr to represent the
    address that's being bound to the socket.
output:
1 - IUCV return code
2 - socket api return code, As defined in the MVS
    TCPIP Programming for return code lists
3 - Error number for the API call, if the api return code
   is -1.
```

OE

Define a local address that maps sock_Inet_sockaddr and fill in the address family, the port number and the IP address of your application.

```
Input parameters:

1 - socket descriptor
2 - port number
3 - address family (AF_INET)
4 - IP address



call BPX1BND
            ( Socket_descriptor,
             length(local_address),
             local_address,
             return_value,
             return_code,
             reason_code);

Output:

1 - Return_value, 0 if successful -1 if failed
2 - Return_code non zero if return_value = -1
3 - Reason_code non zero if return_value = -1
```

CANCEL()

IUCV:

```
input parameters:
1 - Call number
2 - IUCV path ID
3 - Low order halfword of trgcls parameter of request to cancel
4 - Hi order half word of trgcls parameter of request to cancel.
5 - IUCV Message ID of call to be cancelled.

output:

1 - IUCV return code
2 - Socket api return code, as defined in the MVS TCP/IP
    (See MVS TCPIP Programming for return code lists)
3 -  Error number for the API call, if the api return code
    is -1.
```

OE

```
input parameters: the structure is Aiocb

1- Aio#MVS, notify type Exit or ECB
2- Aiocmd = Aio#Cancel
3- AioCancelNoWait ( Wait for cancel to complete)
4- AioSync ( Synchronous call)
5- AioBuffPtr, address of the specific AIOCB being canceled
   or AioFD, to cancel or AIOCBs associated with a single descriptor



CALL BPX1AIO
            (length(cancel_aiocb),
             cancel_aiocb,
             return_value,
             return_code,
             reason_code);

Output:

1 - Return_value, 0 if successful -1 if failed
2 - Return_code non zero if return_value = -1
3 - Reason_code non zero if return_value = -1
```

CLOSE()

IUCV

```
input parameters:
1 - Callnum
2 - IUCV path ID
3 - Socket descriptor being used for the connection
4 - prmmsg

output:

1 - IUCV return code
2 - Socket api return code, As defined in the MVS
    TCPIP Programming for return code lists)
3 - Error number for the API call, if the api return code
        is -1.
```

OE

```
Input parameters:

1- socket descriptor that is being closed


call BPX1CLO
             (socket descriptor,
              return_value,
              return_code,
              reason_code);

Output:
1- return_value, 0 if successful, -1 if failed
2- return_code, non zero if return_value = -1
3- reason_code, non zero if return_value = -1
```

CONNECT()

IUCV:

```
input parameters:
1 - Call number
2 - IUCV path ID
3 - Socket descriptor being used for the connection
4 - sockaddr_in representing remote address to connect to

output:

1 - IUCV return code
2 - Socket api return code, As defined in the MVS TCP/IP
    Programming for return code lists)
3 - Error number for the API call, if the api return code
    is -1.
```

OE

```
 Input parameters: the structure is Aiocb

 1- Define a storage which is mapped by sockaddr.
 2- Set the address family (AF_INET) in the defined storage
 3- Set the applications port in the defined storage
 4- Set the IP address of the client in the defined storage
 5- Set AioFd to the socket descriptor of the connection
 6- Set the notification type Aio#Mvs (EXIT or ECB)
 7- Set AioExitPtr = Addr(exit/ecb) for notification when I/O completes
 8- Set AioCmd = Aio#Connect
 9- Set AioSockAddrPtr to addr(defined storage mapped by sockaddr)
 10-Set AioSockAddrLen to  length of defined storage mapped by sockaddr


call BPX1AIO
            (length(connect_aiocb),
             Connect_aiocb,
             return_value,
             return_code,
             reason_code);

 Output:
 1- return_value, 0 if successful, -1 if failed
 2- return_code, non zero if return_value = -1
 3- reason_code, non zero if return_value = -1
```

GETCLIENTID()

IUCV:

```
input parameters:
1 - Call number
2 - IUCV path ID
3 - prmmsg
    Set to 0
output:
1 - IUCV return code
2 - Socket api return code, as defined in the MVS TCP/IP
    (See MVS TCPIP Programming for return code lists)
3 -  Error number for the API call, if the api return code
     is -1.
4 -   An clientid structure that represents the host
```

OE

NOTE: the getclientid function is not required if the Select() function is omitted.

```
input parameters:

1- Function code 1 or 2
2- Domain, AF_INET, AF_UNIX
3- Client ID structure mapped by BPXYCID

call BPX1GCL
            (Function_code,
             Domain,
             Clientid,
             return_value,
             return_code,
             reason_code);
Output:

1 - Return_value, 0 if successful -1 if failed
2 - Return_code non zero if return_value = -1
3 - Reason_code non zero if return_value = -1
```

GIVESOCKET()

IUCV:

```
input parameters:
1 - Call number
2 - IUCV path ID
3 - Socket descriptor of socket being written to
4 - Givesocket_request
    Initialized to a clientid structure representing the task
    that the socket should be given to.
output:
1 - IUCV return code
2 - Socket api return code, as defined in the MVS TCP/IP
    (See MVS TCPIP Programming for return code lists)
3 -  Error number for the API call, if the api return code
     is -1.
```

OE

NOTE: the givesocket function is not required if the Select() function is omitted.

```
input parameters:
1- Socket descriptor
2- Clientid, a structure mapped by BPXYCID


call BPX1GIV
             (Socket descriptor,
              Clientid,
              return_value,
              return_code,
              reason_code);
Output:

1 - Return_value, 0 if successful -1 if failed
2 - Return_code non zero if return_value = -1
3 - Reason_code non zero if return_value = -1
```

GETPEERNAME()

IUCV:

```
input parameters:
1 - Call number
2 - IUCV path ID
3 - Socket descriptor being used for the connection
4 - Prmmsg

output:

1 - IUCV return code
2 - Socket api return code, as defined in the MVS TCP/IP
    (See MVS TCPIP Programming for return code lists)
3 - Error number for the API call, if the api return code
    is -1.
4 - Sockaddr structure of remote end of connection
```

OE

```
Input parameters:
1- Define a storage mapped by sockaddr
   the sockaddr storage will contain the address of the peer.
2- Set operation type to getpeername



call BPX1GNM
              (socket_descriptor,
               operation_type,
               sockaddr_length,
               sockaddr,
               return_value,
               return_code,
               reason_code);

Output:
1- return_value, 0 if successful, -1 if failed
2- return_code, non zero if return_value = -1
3- reason_code, non zero if return_value = -1
```

GETSOCKOPT()

IUCV:

```
input parameters:
1 - Call number
2 - IUCV path ID
3 - Socket descriptor being used for the connection
4 - prmmsg

output:

1 - IUCV return code
2 - Socket api return code, as defined in the MVS
    TCPIP Programming for return code lists)
3 - Error number for the API call, if the api return code
    is -1.
4 - Sockaddr structure of local end of connection
```

OE

```
Input parameters:

1- Socket descriptor
2- Set operation to sockopt
3- Set level
4- Set Optional name
5- Set optional data length
6- Set optional data



 call BPX1OPT
             (Socket_descriptor,
              Operation,
              Level,
              Option_name,
              Option_data_length,
              Option_data,
              return_value,
              return_code,
              reason_code);

Output:

1 - Return_value, 0 if successful -1 if failed
2 - Return_code non zero if return_value = -1
3 - Reason_code non zero if return_value = -1
```

LISTEN()

IUCV

```
input parameters:
1 - Call number
2 - IUCV path ID
3 - socket descriptor being used for the connection
4 - prmmsg_hi, set to zero
5 - soa_backlog
    Set to (maximum number of queued inboundconnections)

output:

1 - IUCV return code
2 - Socket api return code, as defined in the MVS TCP/IP
    (See MVS TCPIP Programming for return code lists)
3 -  Error number for the API call, if the api return code
    is -1.
```

OE

```
Input parameters:
1- Socket descriptor
2- Backlog (maximum connections that can be queued)


call BPX1LSN
            (Socket_descriptor,
             Backlog,
             return_value,
             return_code,
             reason_code);

Output:

1 - Return_value, 0 if successful -1 if failed
2 - Return_code non zero if return_value = -1
3 - Reason_code non zero if return_value = -1
```

READV()

IUCV:

```
input parameters:

1 - Call number
2 - IUCV path ID
3 - Socket descriptor being used for the connection
4 - prmmsg
5 - Pointer to buffer list of buffers to fill
6 - Length of buffer list

output:

1 - IUCV return code
2 - Socket api return code, as defined in the MVS TCP/IP
    (See MVS TCPIP Programming for return code lists)
3 -  Error number for the API call, if the api return code
         is -1.
```

OE

```
Input parameters: the structure is Aiocb


1 - AioFd, socket descriptor
2 - AioSockAddrLen, readV buffer length
3 - AioSockAddrPtr, readV buffer pointer
4 - AioExitPtr, address of interrupt exit routine
5 - AioNotifyType is Aio#MVS, Notify is using exit
6 - AioBuffAlet is zero,  No Alet
7 - Aiocmd = Aio#ReadV  Operation is READV

CALL BPX1AIO(length(ReadV_aiocb),
              ReadV_aiocb,
              return_value,
              return_code,
              reason_code);


Output:

1 - Return_value, 0 if successful -1 if failed
2 - Return_code non zero if return_value = -1
3 - Reason_code non zero if return_value = -1
```

RCVFROM()

IUCV:

```
input parameters:
1 - Call number
2 - IUCV path ID
3   Socket descriptor being used for the connection
4 - prmmsg_hi
       Set to 0
5 - prmmsg_lo
    flags value - set to scn_msg_oob or scn_msg_dontroute or 0
6 - Pointer to buffer list of buffers to fill
7 -  Length of buffer list
output:
1 - IUCV return code
2 - Socket api return code, as defined in the MVS TCP/IP
    (See MVS TCPIP Programming for return code lists)
3 - Error number for the API call, if the api return code
    is -1.
```

OE

```
Input parameters: the structure is Aiocb
1 - AioFd, socket descriptor
2 - AioSockAddrLen, recvfrom buffer length
3 - AioSockAddrPtr, recvfrom buffer pointer
4 - AioExitPtr, address of interrupt exit routine
5 - AioNotifyType is Aio#MVS, Notify is using exit
6 - AioBuffAlet is zero,  No Alet
7 - Aiocmd = Aio#Recvfrom  Operation is recvfrom

CALL BPX1AIO(length(recvfrom_aiocb),
              recvfrom_aiocb,
              return_value,
              return_code,
              reason_code);
Output:

1 - Return_value, 0 if successful -1 if failed
2 - Return_code non zero if return_value = -1
3 - Reason_code non zero if return_value = -1
```

SELECT()

IUCV:

```
input parameters:
1 - Call number
2 - IUCV path ID
3 - max(mip_max_desc, highest descriptor in use +1)
4 - time_out_flag
      Set to soa_no_check
5 - read_flag
      Set to either soa_check or soa_no_check
6 - write_flag
      Set to either soa_check or soa_no_check
7 - except_flag
      Set to either soa_check or soa_no_check
8 - time_out
      Set to 0
9 - read_list
      Descriptor list of sockets to check for reading
10 - write_list
       Descriptor list of sockets to check for writing
11 - except_list
       Descriptor list of sockets to check for exceptions
output:
1 - IUCV return code
2 - Socket api return code, as defined in the MVS TCP/IP
    (See MVS TCPIP Programming for return code lists)
3 -  Error number for the API call, if the api return code
     is -1.
4 -  A descriptor list of sockets that need to be read
5 -  A descriptor list of sockets that are ready to be written to
6 -  A descriptor list of sockets that have had exception conditions
```

OE - SelPoll()

NOTE - Starting OS/390 R4, the select() function is changed to Selpoll(), which is asynchronous and is issued for Select and POll operations. The interface structure for both functions is BPXYPOLL.

```
Input parameters: the input structure is Aiocb
1- AioCMD = Aio#SelPoll
2- AioBuffPtr, contains address of POLLFD array
3- AioBUFFsize, contains the number of elements in the array
For the Poll() function the PollFD structure is used the same
way the BPX1POL is used. And for the select() function the SelFlags
from the BPXYSEL structure are used. The SelFlags are mapped over
the POllEvents an POllRevents which are defined in the POllFD
structure.
Output:

1 - Return_value, 0 if successful -1 if failed
2 - Return_code non zero if return_value = -1
3 - Reason_code non zero if return_value = -1
```

SOCKET()

IUCV:

```
input parameters:
1 - Call number
2 - IUCV path ID
3 - Socket_domain
    Set to AF_INET
4 - Socket_type
    Always set to SOCK_STREAM
5 - Socket_protocol
    Always set to 0
6 - Socket descriptor of new socket
output:
1 - IUCV return code
2 - Socket api return code, as defined in the MVS TCP/IP
    (See MVS TCPIP Programming for return code lists)
3 -  Error number for the API call, if the api return code
     is -1.
```

OE

```
Input parameters:
1- Domain (AF_INET, UNIX_INET)
2- Type (stream, datagram)
3- Protocol (TCP,UPD)
4- Dimension (1 or 2)
5- Socket vector (the returned socket descriptor holder)


call BPX1SOC
             (Domain,
              Type,
              protocol,
              Dimension,
              Socket_vector
              return_value,
              return_code,
              reason_code);
Output:

1 - Return_value, 0 if successful -1 if failed
2 - Return_code non zero if return_value = -1
3 - Reason_code non zero if return_value = -1
```

TAKESOCKET()

IUCV:

```
input parameters:
1 - Call number
2 - IUCV path ID
3 - Soa_clientid structure representing task that's giving away the
    socket.
4 - Socket descriptor that giving task knows.
5 - Socket descriptor that taking task will use

output:
1 - IUCV return code
2 - Socket api return code, as defined in the MVS TCP/IP
    (See MVS TCPIP Programming for return code lists)
3 -  Error number for the API call, if the api return code
    is -1.
```

OE

NOTE: the takesocket function is not required if the Select() function is omitted.

```
input parameters:
1- Clientid, a structure mapped by BPXYCID
2- Socket descriptor


call BPX1TAK
            (Client_ID,
             Socket descriptor,
             return_value,
             return_code,
             reason_code);
Output:

1 - Return_value, 0 if successful -1 if failed
2 - Return_code non zero if return_value = -1
3 - Reason_code non zero if return_value = -1
```

SENDTO()

IUCV:

```
input parameters:
1 - Call number
2 - IUCV path ID
3 -  Socket descriptor of socket being written to
4 -  Socakaddr structure representing datagram destination
5 -  Pointer to buffer list
6 -  Length of buffer list

output:

1 - IUCV return code
2 - Socket api return code, as defined in the MVS TCP/IP
    (See MVS TCPIP Programming for return code lists)
3 -  Error number for the API call, if the api return code
     is -1.
```

OE

```
Input parameters: the structure is Aiocb
1 - AioFd, socket descriptor
2 - AioSockAddrLen, sendto buffer length
3 - AioSockAddrPtr, sendto buffer pointer
4 - AioExitPtr, address of interrupt exit routine
5 - AioNotifyType is Aio#MVS, Notify is using exit
6 - AioBuffAlet is zero,  No Alet
7 - Aiocmd = Aio#Sendto  Operation is Sendto

CALL BPX1AIO(length(sendto_aiocb),
              sendto_aiocb,
              return_value,
              return_code,
              reason_code);
Output:

1 - Return_value, 0 if successful -1 if failed
2 - Return_code non zero if return_value = -1
3 - Reason_code non zero if return_value = -1
```

SETSOCKOPT()

IUCV:

```
input parameters:

1 - Call number
2 - IUCV path ID
3 - Socket descriptor being used for the connection
4 - Flags describing option to set

output:

1 - IUCV return code
2 - Socket api return code, as defined in the MVS TCP/IP
    (See MVS TCPIP Programming for return code lists)
3 -  Error number for the API call, if the api return code
         is -1.
```

OE

```
input parameters:
1- Socket descriptor
2- Set operation to sockopt
3- Set level
4- Set Optional name
5- Set optional data length
6- Set optional data

call BPX1OPT
            (Socket_descriptor
             Operation
             Level
             Option_name
             Option_data_length
             Option_data
             return_value,
             return_code,
             reason_code);

Output:

1 - Return_value, 0 if successful -1 if failed
2 - Return_code non zero if return_value = -1
3 - Reason_code non zero if return_value = -1
```

WRITEV()

IUCV:

```
input parameters:
1 - Call number
2 - IUCV path ID
3 - Socket descriptor of socket being written to
4 - Pointer to buffer list
5 - Length of buffer list
output:
1 - IUCV return code
2 - Socket api return code, as defined in the MVS TCP/IP
    (See MVS TCPIP Programming for return code lists)
3 -  Error number for the API call, if the api return code
    is -1.
```

OE

```
Input parameters: the structure is Aiocb
1 - AioFd, socket descriptor
2 - AioSockAddrLen, WriteV buffer length
3 - AioSockAddrPtr, WriteV buffer pointer
4 - AioExitPtr, address of interrupt exit routine
5 - AioNotifyType is Aio#MVS, Notify is using exit
6 - AioBuffAlet is zero,  No Alet
7 - Aiocmd = Aio#Write  Operation is Write

CALL BPX1AIO(length(WriteV_aiocb),
              WriteV_aiocb,
              return_value,
              return_code,
              reason_code);

Output:

1 - Return_value, 0 if successful -1 if failed
2 - Return_code non zero if return_value = -1
3 - Reason_code non zero if return_value = -1
```

IUCV API to SE Macro API

The IUCV API evolved into the SE macro API as a step towards removing the VM aspect of the API. In its initial release (TCP/IP V3R1), it provided a major subset of the IUCV API functions; in TCP/IP V3R2, the remaining functions were added (e.g., the xxxxV functions like READV). Consequently, the two APIs are very similar.

This section will address the areas where the correlation between the two APIs is not intuitively obvious. In general, once the connection to TCP/IP is established, the standard BSD functions (socket(), bind(), listen(), ...) match up one-to-one. The areas that need explanation are:

- Initialization/termination
- Apitype=3 processing
- Async processing
- Storage areas for subtasks
- Global storage area

Initialization/Termination

For both IUCV API and SE API, the initialization process accomplishes three tasks:

- Defines the apitype (2 or 3)
- Defines the maximum number of sockets per process
- Identifies the TCP/IP stack for connection

The main task of the termination process is to sever the connection to the TCP/IP.

Initialization and termination is accomplished as follows:

IUCV API

- IUCVMINI SET
- IUCV CONNECT
  - USERID === userid of TCP/IP address space
- IUCVMCOM CONNECT
  - NAME === unique name to identify path
- IUCV SEND (initial message)

```
offset  8 === max sockets
offset 10 === apitype
offset 12 === subtaskid
```

- IUCV SEVER
  - NAME === unique name from IUCVMCOM CONNECT

The SE INITAPI essentially replaces the IUCV SET, CONNECT and initial message SEND. The SE TERMAPI essentially replaces the IUCV SEVER.

APITYPE=3

For SE API, the apitype=3 is specified on the INITAPI (parm APITYPE); for IUCV API, it is specified as the 2-byte filed at offset 10 on the Initial Message SEND. Both APIs have the CANCEL call to cancel a previously issued socket call.

ASYNC Processing

External interrupts POST ECBs or driver exits in both APIs. IN IUCV API, the ECB is part of the IPARML and the exit is specified on the IUCVMINI SET macro. In SE API, the ECB is a EZASMI macro keyword and the exit is specified as an address in the INITAPI ASYNC parm.

Task Storage Area

In the SE API, a TASK macro is provided for allocating a storage area that is addressable to all socket users communicating across a particular connection. This is a feature that was not available in the IUCV API.

Global Storage Area

In the SE API, a GLOBAL macro is provided for allocating a storage area that is addressable to all socket users in an address space. This is a feature that was not available in the IUCV API.

VMCF API to SE API

Readers, notify us if you require this section.

VMCF API to BPX API

Readers, notify us if you require this section.

SE Macro API to BPX API

When converting from the SE macro API to the OE BPX API, there are several areas that need special attention. In general, the standard BSD socket functions are similar and have the same parameters. As noted in earlier sections, asnyc functions in the BPX API use an AIOCB control block to pass the parameters. Using the AIOCB macro, one can easily see the matchup of the keyword parameters on the SE API functions (e.g., "S" for socket descriptor) and the BPX API functions (e.g., "AioFd" for file descriptor).

The areas that need explanation are:

- Initialization/termination
- Apitype=3 processing
- Async processing
- Storage areas for subtasks
- Global storage area
- IOV/ALETs

Initialization/Termination

The BPX API does not require functions like SE API's INITAPI and TERMAPI. However, for outstanding requests to be CANCELed, one should issue BPX1AIO with the CANCEL command in the logic where the TERMAPI was done. To establish the TCP/IP with which to connect, use the BPX1IOC with SIOCSETRTTD.

APITYPE=3 Processing

Not a problem: BPX API naturally handles the multiple requests since it does not have the concept of an INITAPI connection.

Async processing

The major difference here is that not all BPX functions are asynchronous whereas all SE API calls can be asynchronous. The asynchronous BPX calls are:

- accept
- connect
- read
- write
- readv
- writev
- recv
- send
- recvfrom
- sendto

- recvmsg
- sendmsg
- selpoll
- cancel

Storage areas for subtasks

There are no special BPX socket calls associated with memory areas per connection since there is no concept equivalent to this SE API concept in the BPX API. Shared Memory calls are available if needed.

Global storage area

There are no special BPX socket calls associated with memory areas for address spaces. Shared Memory calls are available if needed.

IOV/ALETs

In the BPX API, there is one ALET associated with the recvmsg/sendmsg iov. One technique for converting a SE API iov-type call to the BPX API is to break the call into subgroups of iov's, one per unique ALET.

TCP/IP C Sockets to OE C sockets

The C sockets APIs (OE and non-OE) are Berkeley-based, albeit the non-OE C API is BSD 4.3 and the OE C sockets is BSD 4.4. For most applications, there is no API change when converting from non-OE to OE C sockets. The socket calls and their parameters follow the Berkeley standard and therefore are basically platform-independent. There are 13 additional socket functions that TCP/IP provides on the OS/390-MVS platforms and were listed previously, but these are available and identical on both OE and non-OE C socket APIs.

And now for the exceptions to the above generalization. There are some code changes that are required when converting from non-OE to OE C sockets, but they are mainly for setting up the environment and choosing the correct header files and do not involved logic changes. This is a summary list of source code changes. The specifics will follow later in this section.

- #include headers . For example, OE C does not use a "manifest.h", but non-OE C sockets does require it.
- AF_IUCV. Used in non-OE but replaced with AF_UNIX in OE.
- #defines . For example, OE uses #define _ALL_SOURCE .
- errno values. There are some minor differences.

The are two parts to the process of converting to OE C sockets. The first part involves possible modification to the actual source code. The second part involves the environment (i.e., JCL, OE shell). These changes are described below.

Source and JCL Differences

There are a few key items that allow one to build an OE C socket app versus a non-OE C socket app via JCl.

- OE C app
  - do not #include manifest.h in your C source
  - #include errno.h, not tcperrno.h, in your C source
  - #include types.h, not bsdtypes.h in your C source
  - define _ALL_SOURCE during compile. Other variations of defines are also adequate, depending upon your needs. For example, _XOPEN_SOCKETS, _OE_SOCKETS.
  - remove tcpip.SEZACMAC from the SYSLIB of compile step (and from the LINKLST)
  - remove tcpip.SEZARNT1 from the SYSLIB of prelink step (and from the LINKLST)
  - remove tcpip.SEZACMTX from the SYSLIB of linkedit step (and from the LINKLST)
- non-OE C app
  - #include manifest.h as the first header in your C source
  - #include bsdtypes.h, not types.h, in your C source
  - #include tcperrno.h, not errno.h, in your C source
  - add tcpip.SEZACMAC at the front of the SYSLIB of compile step
  - add tcpip.SEZARNT1 at the front of the SYSLIB of prelink step
  - add tcpip.SEZACMTX at the front of the SYSLIB of linkedit step

Exceptions

There is one issue that may actually affect the code logic, contrary to the blanket statement made above: AF_IUCV domain sockets.

In non-OE, the AF_IUCV domain is used for interprocess communication (IPC). IPC-type socket are used to communicate between processes that reside on the same local host. In this environment, TCP/IP provided AF_IUCV domain that would use the VMCF/IUCV address space to communicate between local processes; the TCP/IP address space is not involved in the communications. This AF_IUCV domain has been available from the TCP/IP V2 release, but it requires that the VMCF/IUCV address space be up and available.

In the OE environment, there is no AF_IUCV domain; its OE counterpart is AF_UNIX. Applications that used AF_IUCV domain sockets in the non-OE environment have to change code to use the equivalent AF_UNIX domain sockets in the OE environment. The structure sockaddr_un is simpler than the structure sockaddr_iucv, so modifying code to use it (e.g., for bind()) is straightforward.

```
/*-------------------------------------------------------------*/
/* non-OE AF_IUCV socket example                               */
/* (pseudo-code, just the key points noted)                    */
/*-------------------------------------------------------------*/
 . . .
struct sockaddr_iucv saiu;                  /* name buffer   */
 . . .
memset((&saiu, 0, sizeof(saiu));
strcpy(saiu.siucv_nodeid, "        ", 8);
strcpy(saiu.siucv_userid, "MVSUSER1", 8);
strcpy(saiu.siucv_name,   "APPL", 4);
saun.sun_family = AF_IUCV;
 . . .
s = socket(AF_IUCV, SOCK_STREAM, 0);
 . . .
rc = bind(s, (struct sockaddr *)&saiu, sizeof(saiu));
 . . .

/*-------------------------------------------------------------*/
/* OE AF_UNIX socket example                                   */
/* (pseudo-code, just the key points noted)                    */
/*-------------------------------------------------------------*/
 . . .
struct sockaddr_un saun;                    /* name buffer   */
char sock_nameû¾ = "/tmp/ralph";
 . . .
memset((&saun, 0, sizeof(saun));
strcpy(saun.sun_path, sock_name);
saun.sun_len    = sizeof(saun.sun_path);
saun.sun_family = AF_UNIX;
 . . .
s = socket(AF_UNIX, SOCK_STREAM, 0);
 . . .
rc = bind(s, (struct sockaddr *)&saun, SUN_LEN(&saun));
```

Also be aware of BSD 4.3 versus 4.4 differences. The one that immediately comes to mind is the change for struct sockaddr_in. In BSD 4.4, the first bytes is now a length field. Our TCP/IP successfully handles this difference and can accept both structure formats on calls such as bind().

Related Topics

The following sections are related to converting APIs. They are included here to give the reader a better understanding of the environment or to help analyze problems that occur during conversion.

IBM Documents

In addition to the TCP/IP and OS/390 IBM manuals, IBM also provides "redbooks" (they have red covers) that explain topics in a more user-oriented way. The manuals and redbooks below are just a subset of the books that you may find useful.

- SG24-4721, "Accessing OS/390 OpenEdition MVS from the Internet"
- GG24-4473, "Porting Apps to the OpenEdition MVS Platform"
- GG24-2561, "Beginner's Guide typ MVS TCP/IP Socket Programming"
- SC23-3020, "Assembler Callable Services for OpenEdition MVS"
- SC31-7187, "TCP/IP for MVS API Reference"

OE Shell environment

In the OE environment, one has the option of using either JCL or using commands from the OE shell to perform actions like compiling, linking or running code. For those people familiar with a UNIX-like environment, the OE shell provides that look-and-feel, and it has many utilities (e.g., make) that are available on other platforms. For specifics, see the series of OS/390 OE manuals (SC28-18xx).

c89 psuedo-JCL

The following example is the verbose output from running "c89" in the OE shell to print the pseudo-jcl that is generated. It is included here for illustration purposes only. During normal OE shell operations, this is transparent to the user.

```
# c89 -vv -W "0,OPT(1)" -D_OE_SOCKETS -D_ALL_SOURCE tst.c
//* c89 ----------------------------------------------------
//COMPILE   EXEC  PGM=CBCDRVR,
//  PARM='POSIX(OFF) ENVAR(__CBC_EXTENDED=1)/
//  "./" "./tst.c" "*.c"
//  OE,NOLSEARCH,NOUSERLIB,NOSYSLIB,
//  NOGENPCH(./),NOUSEPCH(./),
//  START,LONGNAME,RENT,NODECK,TERMINAL,
//  OBJECT(tst.o),
//  NOCSECT,HALT(16),NOMARGINS,NESTINC(255),NOSEQUENCE,NOSSCOM,
//  NOEVENTS,NOALIAS,TARGET(LE),NODLL,NOEXPORTALL,
//  ARGPARSE,EXECOPS,PLIST(HOST),REDIR,
//  LOCALE(POSIX),
//  LANGLVL(ANSI),NOUPCONV,NOLIBANSI,ANSIALIAS,
//  NOINLINE(AUTO,REPORT,100,1000),MEMORY,MAXMEM(*),SPILL(128),
//  NOTEST,NOGONUMBER,
//  NOLIST(/dev/fd1),NOSOURCE(/dev/fd1),
//  NOAGGREGATE,NOEXPMAC,NOOFFSET,NOSHOWINC,NOXREF,
//  NOCHECKOUT,FLAG(W)
//  ,OPT(1),
//  DEFINE(_OE_SOCKETS=1),
//  DEFINE(_ALL_SOURCE=1),
//  DEFINE(errno=\\(*__errno\\(\\)\\)),
//  DEFINE(_OPEN_DEFAULT=1),
//  NOSEARCH,SEARCH(/usr/include/,
//    //''CEE.OSV1R4.SCEEH.+''),
//  NOPPONLY(NOCOMMENTS,NOLINES,/dev/fd1,2048),
//  OPTIMIZE(0),NOIPA'
//*STEPLIB DD  DSN=CEE.OSV1R4.SCEERUN,DISP=SHR
//*        DD  DSN=CBC.OSV1R4.SCBCCMP,DISP=SHR
//* c89 ----------------------------------------------------
//LINKEDIT  EXEC  PGM=LINKEDIT,
//  PARM='AMODE=31,RMODE=ANY,TERM=YES,
//  COMPAT=CURRENT,DYNAM=DLL,ALIASES=NO,UPCASE=NO,
//  LIST=OFF,MAP=NO,XREF=NO,MSGLEVEL=4,
//  REUS=RENT,EDIT=YES,AC=0,CALL=YES,CASE=MIXED'
//SYSLIB   DD  DSN='CEE.OSV1R4.SCEELKEX',DISP=SHR,DCB=DSORG=DIR
//         DD  DSN='CEE.OSV1R4.SCEELKED',DISP=SHR,DCB=DSORG=DIR
//         DD  DSN='SYS1.CSSLIB',DISP=SHR,DCB=DSORG=DIR
//C8920    DD  UNIT=SYSDA,SPACE=(32000,(30,30)),
//             STORCLAS=,MGMTCLAS=,DATACLAS=,DSNTYPE=,
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//C8961    DD  DSN='CEE.OSV1R4.SCEEOBJ',DISP=SHR,DCB=DSORG=DIR
//SYSPRINT DD  PATH='/dev/fd1',
//             PATHOPTS=(ORDWR,OCREAT,OAPPEND),
//             PATHMODE=(SIROTH,SIRGRP,SIRUSR,SIWOTH,SIWGRP,SIWUSR)
//SYSTERM  DD  PATH='/dev/fd2',
//             PATHOPTS=(ORDWR,OCREAT,OAPPEND),
//             PATHMODE=(SIROTH,SIRGRP,SIRUSR,SIWOTH,SIWGRP,SIWUSR)
//SYSLMOD  DD  PATH='a.out',
//             PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//             PATHMODE=(SIRWXO,SIRWXG,SIRWXU)
//SYSLIN   DD  *
  INCLUDE C8920
  INCLUDE './tst.o'
  AUTOCALL C8961
/*
```

Common non-OE JCL Errors

We have all been there. It looks right, but the damn jcl is still complaining! Well, here are some typical errors that you might see. These are not necessarily errors related to OE C socket conversion, but you may encounter them when converting your JCL to the OS/390 compiler and environment.

- "IEFC621I EXPECTED CONTINUATION NOT RECEIVED" followed by a bunch of other nasty messages.

  Reason: a line of JCL had a comma (i.e., continuation indicator) at the end, but no appropriate line of JCL following it. Typically this happens when you are copying PROC lines. Remember - the last PROC KEYWORD line does not have a comma.

- "IEFC657I THE SYMBOL RALPH WAS NOT USED"

  Reason: you have a KEYWORD 'RALPH' on your PROC, but you never accessed it via '&RALPH' in your JCL. MVS doesn't like that - so remove the reference to it from your PROC statement.

- You have inconsistent dataset characteristics.

  Messages like "cant open dataset" may be associated with dataset format. For example,

  – Source code PDSs are usually FB, 80, 3200 (i.e., RECFM, LRECL, BLKSIZE)
  – Object PDSs are usually FB, 80, 3200
  – Load module PDSs are usually U, 0, 32760

- Concatenated DD cards have strict requirements about LRECL and BLKSIZE. The dataset characteristics of the first one in the search apply to subsequent ones.
- Header search order. You usually get what you ask for, so be careful. For example, hlq.SEZACMAC has the non-OE C socket header files. Since these contain the same-name headers as other PDSs, be sure to have it first in your SYSLIB DD during compiles for non-OE applications.
- Error message about incompatible file formats

  Reason: when concatenating datasets, the first dataset must indicate the largest blocksize. If the first dataset does not have the largest blocksize, then you can overtly specify the largest blocksize via "BLKSIZE=32760" (or whatever you need) on that first dataset DD.

DD Cards

Below is a list of some important JCL DD cards that you may want to use. The description is brief, but should give you the flavor of the card's purpose.

- JOBLIB - specifies private program libraries to be used in the search during each jobstep of the job. Put it right after JOB card.
- STEPLIB - specifies private program libraries to be used in the search during the specific jobstep in which it is put. It overrides the JOBLIB card.
- SYSIN - input dataset for the compile
- SYSLIN - output dataset from the compile; input to the linkedit step
- SYSLMOD - output dataset from the linkedit
- SYSLIB - during compile, specifies search order for C header files included via <> (e.g., "#include <bozo.h>")

- SYSLIB - during linkedit, specifies search order for object files referenced in the compile but not yet resolved.
- USERLIB - during compile, specifies search order for C header files included via quotes (e.g., #include "bozo.h")

Compiler/RTL Libraries

For our development environment, the following PDSs were used.

- CBC.OSVxRy.SCBCPRC - contains compiler procedures.
  - EDCC - compile a C program
  - CBCC - compile a C++ program
  - EDCCL - compile and link a C program
  - CBCCL - compile, prelink and link a C++ program
- CBC.OSVxRy.SCBCCMP - compiler modules specified in STEPLIB of compile.
- CBC.OSVxRy.SCBCSAM - sample C and C++ code.
- CEE.OSVxRy.SCEEH.* - contains RTL C headers.
- CEEL.OSVxRy.SCEERUN - runtime library. Specified in STEPLIB of compile.