RPG IV Socket Tutorial

Scott Klement

RPG IV Socket Tutorial

by Scott Klement

This tutorial strives to teach network programming using sockets to AS/400 or iSeries programmers who use the RPG IV programming language.

It is assumed that the reader of this tutorial is already familiar with the RPG IV language, including the use of prototypes, subprocedures, service programs and pointers in an RPG IV environment.

Table of Contents

Chapter 1. Introduction to TCP and Sockets

Written by Scott Klement.

1.1. TCP/IP Concepts and Terminology

This section is an introduction to TCP/IP programming using a sockets API. (Sockets can also be used to work with other network protocols, such as IPX/SPX and Appletalk, but that is beyond the scope of this document.) The standard socket API was originally developed in the Unix world, but has been ported to OS/400 as part of the "Unix-type" APIs, and a modified version was also ported to the Windows platform under the name "Windows Sockets" (or "Winsock" for short)

Usually when someone refers to "TCP/IP" they are referring to the entire suite of protocols, all based on the Internet Protocol ("IP"). Unlike a single network, where every computer is directly connected to every other computer, an "inter-network" (or "internet") is a collection of one or more networks. These networks are all connected together to form a larger "virtual network". Any host on this virtual network can exchange data with any other host, by referring to the hosts "address".

The address is a 32-bit number which is unique across the entire internet. Typically, this number is broken into 4 8-bit pieces, separated by periods to make it easier for humans to read. This human readable format is called "dotted-decimal" format, or just "dot-notation". An address displayed in this fashion looks something like "192.168.66.21".

Different parts of this "IP Address" are used to identify which network a host is located on, and the rest of the address is used to identify the host itself. Which part of the address and which part is the host is determined by a "network mask" (or "netmask" for short.) The netmask is another 32-bit number which acts like like a "guide" to the IP address. Each bit that is turned on in the netmask means that the corresponding bit in the IP address is part of the network's address. Each bit that is turned off means that the corresponding bit in the IP address is part of the host's address.

Here's an example of an IP address and netmask:

```
                    dotted-decimal   same number in binary format:
                    --------------   -----------------------------------
        IP Address:     192.168.66.21    11000000 10101000 01000010 00010101
           Netmask:     255.255.255.0    11111111 11111111 11111111 00000000

Network Address is: 192.168.66       11000000 10101000 01000010
   Host Address is:           .21                                00010101
```

A slightly more complicated example:

```
                    dotted-decimal   same number in binary format:
                    --------------   -----------------------------------
        IP Address: 192.168.41.175   11000000 10101000 00101001 10101111
           Netmask: 255.255.255.248  11111111 11111111 11111111 11111000

Network Address is: 192.168.41.168   11000000 10101000 00101001 10101
   Host Address is:              7                                    111
```

When a system sends data over the network using internet protocol, the data is sent fixed-length data records called datagrams. (These are sometimes referred to as "packets") The datagram consists of a "header" followed by a "data section". The header contains addressing information, much like an envelope that you send through your local postal service. The header contains a "destination" and a "return to" address, as well as other information used by the internet protocol. Another similarity between IP and your postal service is that each packet that gets sent isn't guarenteed to arrive at the destination. Although every effort is made to get it there, sometimes datagrams get lost or duplicated in transit. Furthermore, if you send 5 datagrams at once, there's no guarantee that they'll arrive at their destination at the same time or in the same order.

What's really needed is a straight-forward way to ensure that all the packets that get sent arrive at their destination. When they arrive, make sure they're in the same sequence, and that all duplicated datagrams get discarded. To solve this problem, the Transmission Control Protocol (TCP) was created. It runs on top of IP, and takes care of the chore of making certain that every packet that is sent will arrive at its destination. It also allows many packets to be joined together into a "continuous stream" of bytes, eliminating the need for you to split your data into packets and re-join them at the other end.

It's useful to remember that TCP runs "on top of" IP. That means that any data you send via TCP gets converted into one or more datagrams, then sent over the networking using IP, then is reassembled into a stream of data on the other end.

TCP is a "connection oriented protocol", which means that when you want to use it, you must first "establish a connection." To do this, one program must take the role of a "server", and another program must take the role of a "client." The server will wait for connections, and the client will make a connection. Once this connection has been established, data may be sent in both directions reliably until the connection is closed. In order to allow multiple TCP connections to and from a given host, "port" numbers are established. Each TCP packet contains an "origin port" and a "destination port", which is used to determine which program running under which of the system's tasks is to receive the data.

There are two other protocols that are used over IP. They are the "User Datagram Protocol (UDP)", and the "Internet Control Message Protocol" (ICMP).

UDP is similar to TCP, except that data is sent one datagram at a time. The major difference between the UDP datagrams and the "raw" IP datagrams is that UDP adds port numbers to the packets. This way, like TCP, many tasks on the system can use UDP at the same time. UDP is usually used when you know that you only want to send a tiny amount of data (one packets worth) at a time, and therefore you don't need all the extra overhead of TCP.

ICMP is used internally by the internet protocol to exchange diagnostic and error messages. For example, when you attempt to connect to a port on a remote machine, and that machine chooses to refuse your connection, it needs some way of telling you that the connection has been refused. This is done by sending ICMP messages. You never need to write or receive ICMP messages directly, they are always handled by the TCP/IP stack. They are strictly a "control message protocol."

Another important concept is that of a socket. A socket is an "endpoint for network communications." In other words, it's the virtual device that your program uses to communicate with the network. When you want to write bytes out over the network, you write them to a socket. When you read from the network, you read from that socket. In this way, it is similar to the way a "file" is used to interact with hard drive storage.

The last thing that I'd like to cover here is "domain names." As you read above, all TCP/IP communications are done using an "address." Without this address, no data can be sent or received. However, while addresses work very well for the computer, they're a little hard for people to remember. Perhaps you wanted to connect to a computer the computer that keeps track of inventory at Acme, Inc. How do you know it's address? If you knew it's IP address already, how would you remember it, along with all of the other addresses that you use? The answer is the "domain name system" or "DNS".

DNS is a large, distributed, database containing mappings between human readable names (such as "inventory.acme.com") and IP addresses (such as "199.124.84.12") When you ask the computer for the IP address for "inventory.acme.com", it follows these steps:

1. Checks to see if that host name is in the local computer’s "host table". (On the AS/400, you can type ’CFGTCP’ and choose opt#10 to work with the host table) If it finds an entry for "inventory.acme.com", it returns this to your program. If not, it tries step #2.
2. It tries to contact a DNS server. (The DNS server may be on the same machine, or on another machine on the network, it doesn’t matter)
3. The DNS server may already know the IP address that’s associated with "inventory.acme.com". Each time it looks up a new name, it "caches" it for a period of time. So, if this particular name is in it’s cache, it can return it right away. If not, it goes on to the next step.
4. Since there are so many millions (billons?) of host names in the world, you cannot store them all on one server. Instead, the names are served by an entire hierarchy of DNS servers. Each level of the hierarchy relates to a different component of the host’s name. The components are separated by periods.
5. So, "inventory.acme.com" gets separated into "inventory", "acme" and "com". The DNS server asks the "root level" DNS servers for the server that handles "com" domains. (The DNS server will cache these requests as well, so once it knows who handles "com" domains, it won’t ask again). The root server returns the IP address for "com" domains.
6. The DNS server then asks the server for "com" domains for the address of the server that handles "acme" domains. The "com" server will then return the address of acme’s DNS server. The DNS server caches this, as well.
7. The DNS server asks the "acme" server for the address of the "inventory" host. This address gets returned, and cached by your DNS server.
8. Finally, the DNS server returns this address to your program.

1.2. Overview of a TCP communications session

A good analogy to a TCP connection is a telephone call.

When you wish to place a telephone call, you first look up the telephone number of the person you wish to call. If there are many people who can be reached at that telephone number, you also look up an extension number. You then lift up the receiver and dial the number. When the person you wish to reach has answered the phone, you talk to that person. That person talks to you, as well. When your conversation is complete, each person hangs up the telephone.

A TCP connection is very similar. First you look up the IP address (telephone number) of the computer you wish to contact. For a TCP connection, there are always (the potential for) many different services at that address. So you have to look up the port (extension number). You then create a socket (pick up the receiver) and dial the number (connect to the IP and socket). Then the program you wish to contact has accepted your connection (answered the phone) you can send and receive data from that computer (hold a conversation), and finally each side closes their socket (hangs up the phone).

When you make a telephone call, you don't have to worry about how your voice gets converted to an electrical signal, or how it gets switched to the telephone of the person that you're calling. The telephones and the telephone company takes care of all of these details for you.

Likewise, when you connect to another computer using TCP, you don't have to worry about the details of how your data gets split up into different datagrams, or how it ensures that the data gets put back together in the correct order, or even how it gets routed to the computer that you're connecting to. The sockets API takes care of all of these details for you.

The main procedures that you call from the sockets API are listed here with a brief explanation:

- gethostbyname = Look up an IP address from a domain name (look up a person's telephone number)
- getservbyname = Look up a port number from a service name (look up a person's extension number)
- socket = create a socket (lift up the telephone handset)
- connect = connect to a given IP address and port number (dial the telephone number)
- bind = force the socket API to utilize a particular port or address. (doesn't really work with the telephone analogy, but this is used to tell the API which port number your socket will use on the local machine)
- listen = Tell the socket API that you wish to receive connections (switching on the "ringer" on your telephone)
- accept = Accept an incoming connection (answer a telephone call)
- send = send data out over a socket (talk into your telephone handset)
- recv = receive data from a socket (listen to your telephone handset)
- close = close a socket (hang up the telephone)

Therefore, a typical TCP session looks like this:

server	client
getservbyname()	
socket()	
bind()	
listen()	
	gethostbyname()
	getservbyname()
	socket()
accept()	connect()
send() & recv()	send() & recv()
close()	close()

The getservbyname() call asks the operating system which port number a server will accept connections on for a given service.

For the sake of an example, we'll use http, which is the protocol used to transport web pages across the internet.

The server will start by calling getservbyname to ask the operating system which port is used for the http requests over tcp. The getservbyname API will return port number 80, which happens to be the worldwide standard for the http service. The server will then call the socket() procedure, this socket will then be bound to port 80 by calling the bind() procedure. Now, when we call the listen() procedure, the socket API will cause port 80 to be "open" for receiving requests. The program then calls accept(), which waits for someone to connect to it. Once a client has connected, it can send and receive data. Finally, when it’s done, it closes the connection.

The client first asks the user for the host to connect to. Once it has a response from the user, it checks to see if the user supplied an IP address or if the user actually supplied a host name. If the user supplied a host name, it calls gethostbyname to find the IP address of the host it needs to connect to. Now, it needs to know which port to connect to, so it calls getservbyname to ask the operating system which port is used for http. The operating system will, of course, return 80. Now, it creates a socket which it can use to talk to the server by calling the socket() procedure. It uses the connect() procedure to connect that socket to the server, who is hopefully waiting for it to connect. Now it can send & receive data from the server. And finally, it calls close() to disconnect.

Chapter 2. Looking up host names & services

Written by Scott Klement.

2.1. Services and Ports

The previous sections described an overview of TCP/IP terminology as well as an overview of calling the sockets API for a TCP connection. This section describes the details of calling "getservbyname()" and explains how to understand the IBM UNIX-type API manual.

The Socket API is described in the IBM Information Center under the category "UNIX-type APIs." Unfortunately, the calling syntax that's described is for people using the ILE C/400 programming language.

The getservbyname() API is documented here:
 http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/gsrvnm.htm
(http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/gsrvnm.htm)

And it says that in order to call getservbyname, we need to specify parms like this:

```
struct servent *getservbyname(char *service_name,
                              char *protocol_name)
```

By now, unless you're familiar with programming in C, you're probably saying "huh? what does that mean?" So, I'll tell you :) Let's look at that statement again, but break it into pieces:

```
❶struct servent *❷getservbyname(❸char *service_name, ❹char *protocol_name)
```

❶ Each prototype in C starts with the prcoedure's return value. In this case, the '*' means that it returns a pointer, and the 'struct servent' means that this pointer points to a structure of type 'servent' (service entry).

❷ The name of the procedure to be called is 'getservbyname'.

❸ The first parameter is a pointer. (I know this because of the '*') and it points to a character variable called "service_name." Character variables are type 'A' in RPG and DDS.

❹ The second parameter is another pointer to a character variable. This one is called "protocol name".

Since names in C are case sensitive, we MUST supply the name 'getservbyname' as all lowercase. (That's important since RPG likes to convert everything to uppercase at compile-time).

So, to define this same prototype in RPG, we need to do this:

```
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++
D getservbyname   PR              *   ExtProc('getservbyname')
D   service_name                  *   value options(*string)
D   protocol_name                 *   value options(*string)
```

Note that all pointers passed to C procedures are always passed by "value". The keyword "options(*string)" is an option added to RPG to help make it compatible with C programs. Strings in C end with a "null" character (x'00') that allows it to recognize the end of a variable length string. Specifying options(*string) causes RPG to automatically add this trailing null when it makes the call.

If you look further down, (in the return value section) they describe this "servent" structure by showing us this:

```
struct servent {
  char            *s_name;    ❶
  char            **s_aliases;❷
  int             s_port;     ❸
  char            *s_proto    ❹
};
```

This means that the servent structure contains 4 pieces of data:

❶ A pointer to an area of memory containing an alphanumeric string. This piece of data is called "s_name".

❷ This one is quite difficult for your typical RPG programmer to understand. It is a pointer to an array. That array is an array of pointers. Each element of that array points to an alphanumeric string.

❸ An integer called "s_port".

❹ The last piece of data is a pointer to an alphanumeric string call "s_proto".

So, if we wanted to define this structure in RPG, we'd do it like this:

```
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++
D p_servent       S               *
D servent         DS                  based(p_servent)
D   s_name                        *
D   s_aliases                     *
D   s_port                      10I 0
D   s_proto                       *
```

As you can probably tell from this reading this page, so far, the hardest part of calling the UNIX-type APIs from RPG programs is converting things from C syntax to RPG syntax. That's why it's so frustrating that IBM doesn't provide a manual and header files specifically for RPG programmers.

For this tutorial, I will show you how to make the conversions from the C prototypes to the RPG prototypes. If you want to learn more about how to do this yourself, you might try this link: http://www.opensource400.org/callc.html

But, anyway... I'll get off the soap box and get back to the tutorial...

Once we have set up this data structure and procedure prototype, we can call the getservbyname procedure simply by coding:

```
CL0N01Factor1+++++++Opcode&ExtExtended-factor2+++++++++++++++++++++++++
C                   eval      p_servent = getservbyname('http': 'tcp')
```

And then check the value of s_port to find out which port http is located on. As you can see, defining the prototype and data structure is the hard part. Once we've got that, making the call is easy.

So, here's a simple program that accepts a service name as a parameter, and displays the corresponding port number:

```
Member: QRPGLESRC, SERVPORT

     H DFTACTGRP(*NO) ACTGRP(*NEW)

     D getservbyname   PR              *   ExtProc('getservbyname')
     D  service_name                   *   value options(*string)
     D  protocol_name                  *   value options(*string)

     D p_servent       S               *
     D servent         DS                  based(p_servent)
     D   s_name                        *
     D   s_aliases                     *
     D   s_port                      10I 0
     D   s_proto                       *

     D service         S             10A
     D msg             S             50A

     c     *entry        plist
     c                   parm                    service

     c                   eval      p_servent = getservbyname(
     c                                 %trim(service): 'tcp')    ❶

     c                   if        p_servent = *NULL             ❷
     c                   eval      msg = 'No such service found!'
     c                   else
     c                   eval      msg = 'port = ' + %editc(s_port:'L')
     c                   endif

     c                   dsply                   msg

     c                   eval      *inlr = *on
```

Compile this by typing: `CRTBNDRPG SERVPORT SRCFILE(SOCKTUT/QRPGLESRC)`

Run it by typing `CALL SERVPORT PARM('http')`

A few notes:

❶ Note that we must do a %trim() on the service when we call getservbyname. Otherwise, it would search for a service called 'http ' instead of 'http'. Since service names can't legally have spaces in them, this wouldn't work.

❷ The IBM manual states that getservbyname will return NULL when it can't find a service. In RPG, the special value NULL can be represented by the special constant *NULL, so we check for *NULL to see that the service wasn't found.

2.2. Host names and addresses

The previous section described how to look up the port number of a service using the getservbyname() API. This section complements that section by desribing the procedure for looking up an IP address using the gethostbyname() procedure. This section will also describe the system routines for converting from "dotted decimal" IP address format to the 32-bit address format, and back again.

To start with, we'll explain the 'inet_addr' API. It's the system API to convert from dotted decimal formatted IP address to a 32-bit network address. This API is explained in IBM's manuals at the following link:
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/inaddr.htm
(http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/inaddr.htm)

IBM tells us that the prototype for the inet_addr procedure looks like this:

```
unsigned long inet_addr(char *address_string)
```

This means that the procedure is named 'inet_addr', and it accepts one parameter, called 'address_string', which is a pointer to a character string. It returns an unsigned long integer, which in RPG is expressed as '10U 0'.

This is pretty straight forward in RPG. We simply prototype it like this:

```
D inet_addr       PR            10U 0 ExtProc('inet_addr')
D  address_str                    *   value options(*string)
```

Reading further down the IBM page on this procedure, you'll see that it returns the 32-bit network address when successful. And returns a -1 when it's not successful.

Now you should be confused! If it returns an 'unsigned' integer, how can it possibly return a negative number? By definition, an unsigned integer CANNOT be a negative number. In C, when the return value is compared against -1, the computer actually generates a 32-bit integer with the value of -1, and then compares, bit for bit, whether they match, even though one is signed and one isn't, as long as they have the same bit values, it thinks they're the same.

So the question is, what 'unsigned' value has the same binary value as the signed integer value of -1? Well, if you're familiar with the twos complement format, you'll know in order to make a positive number negative, you take it's binary complement, and add 1. Therefore, x'00000001' is a positive 1, and it's binary complement would be x'FFFFFFFE'. If we add 1 to that, we get x'FFFFFFFF'. IF we convert that to decimal, we get the value 4294967295.

Do you still think this is straight-forward? :) To simplify things somewhat, we'll define a named-constant called 'INADDR_NONE' (which, though they dont tell us that, is the name of the same constant in the C header file) When we check for errors, we'll simply check against that constant.

```
D INADDR_NONE     C                   CONST(4294967295)
```

Once we have all the definitions straightened out, this is quite easy to call in RPG:

```
c                   eval      IP = inet_addr('192.168.66.21')
c                   if        IP = INADDR_NONE
c*  conversion failed
c                   endif
```

To convert an IP address from 32-bit network format back to dotted decimal uses a similar API called 'inet_ntoa'. The IBM manual for this is found at this link:
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/inntoa.htm

IBM tells us that the prototype for inet_ntoa looks like this:

```
char *inet_ntoa(struct in_addr internet_address)
```

The procedure is called inet_ntoa. It accepts one parameter, which is a 'in_addr' structure, passed by value. It returns a pointer to a character string. That's nice. But what the heck is an 'in_addr' structure?

You'll note that the return value doesn't even bother to explain this structure. If you have the "system openness includes" installed on your system, you'll be able find the definition for in_addr in the file `QSYSINC/NETINET` in member `IN`. It looks like this:

```
struct in_addr {
 u_long s_addr;
};
```

What this means is that the structure contains nothing but a single, unsigned, 32-bit integer. That's a bit confusing to try to figure out from IBM's documentation, but it's easy to code. The prototype for inet_ntoa in RPG will look like this:

```
D inet_ntoa       PR              *   ExtProc('inet_ntoa')
D  internet_addr                10U 0 value
```

Now, when you call inet_ntoa, it will either return NULL (which in RPG is called *NULL) if there's an error, or it will return a pointer to a character string. Unfortunately, we don't know how long this character string that it returns is! In C, you'd find the length of this string by searching for the 'null' character at the end of the string.

Fortunately, the RPG compiler developers created a BIF for just that purpose. This BIF is called %str, and it accepts a pointer, and returns the string beginning with the pointer, and ending with the trailing null. So we can call inet_ntoa like this:

```
c                   eval      ptr = inet_ntoa(IP)
c                   if        ptr = *NULL
c*  conversion failed
c                   else
c                   eval      dottedIP = %str(ptr)
c                   endif
```

The final API in this chapter is the 'gethostbyname' API, which does a DNS lookup on a given host name, and returns a 'host entry structure' that contains the details about that host. The main piece of data that we're interested in from that structure is the host's IP address.

IBM's manual page for this procedure is found at this link:
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/ghostnm.htm

The prototype IBM lists for this API looks like this:

```
struct hostent *gethostbyname(char *host_name)
```

Which means that the API is 'gethostbyname', it accepts one parameter which is a pointer to a character string containing the host name. It returns a pointer to a 'hostent' structure.

So, the RPG prototype for the gethostbyname API looks like this:

```
D gethostbyname   PR              *   extproc('gethostbyname')
D   host_name                     *   value options(*string)
```

Looking down at the "return value" of the API, it tells us that it either returns NULL (or, in RPG, *NULL) if there's an error, or it returns a pointer to a 'hostent' structure. The hostent structure is described as being in this format:

```
struct hostent {
  char   *h_name;
  char   **h_aliases;
  int    h_addrtype;
  int    h_length;
  char   **h_addr_list;
};

#define h_addr  h_addr_list[0]
```

This means that the hostent data structure contains 5 different items, a pointer to a character string containing the 'correct' hostname, a pointer to an array of pointers, each element pointing to an 'alias' of the hostname, an integer containing the address type, an integer containing the length of each address, and a pointer to an array of pointers, each one pointing to a character string.

The '#define' statement tells us that whenever we refer to 'h_addr' in this structure, that it will actually return the first pointer in the h_addr_list array.

If you read on, under the "Return" value, it explains that both of the arrays pointed to here are 'NULL-terminated lists'. This means that if you loop thru the array of pointers, you'll know you've reached the end when you find a 'NULL' (or *NULL in RPG).

It further tells you that each element of the h_addr_list array actually points to a structure of type 'in_addr', rather than a character string, as the structure definition implies. Why does it do that? Because gethostbyname can be used with other protocols besides TCP/IP. In the case of those other protocols, it might point to something besides an 'in_addr' structure. In fact, we could use the 'h_addrtype' member of the structure to determine what type of address was returned, if we so desired. However, this document is only intended to work with TCP/IP addresses.

Note, also, that we already know from our experiences with inet_ntoa above that 'in_addr' structures are merely an unsigned integer in disguise. Therefore, we can define the 'hostent' structure as follows:

```
D p_hostent       S               *
D hostent         DS                  Based(p_hostent)
D   h_name                        *
D   h_aliases                     *
D   h_addrtype                  10I 0
D   h_length                    10I 0
D   h_addr_list                   *
D p_h_addr        S               *   Based(h_addr_list)
```

```
D h_addr          S             10U 0 Based(p_h_addr)
```

So, if we want to refer to the first IP address listed under h_addr_list, we can now just refer to h_addr. If we need to deal with later addresses in the list (which is very unusual, in my experiences) we can do so like this:

```
D addr_list       S               *   DIM(100) Based(h_addr_list)
D p_oneaddr       S               *
D oneaddr         S             10U 0 based(p_oneaddr)

CL0N01Factor1+++++++Opcode&ExtFactor2+++++++Result++++++++Len++D+HiLo
C                   do        100           x
C                   if        addr_list(X) = *NULL
C                   leave
C                   endif
C                   eval      p_oneaddr = addr_list(X)
C***  oneaddr now contains the IP address of this
C***   position of the array, use it now...
C                   enddo
```

As I stated, however, it's very unusual to need to work with anything but the first address in the list. Therefore, 'h_addr' is usually what we'll refer to when we want to get an IP address using gethostbyname.

And we'd normally call gethostbyname like this:

```
c                   eval      p_hostent = gethostbyname('www.ibm.com')
c                   if        p_hostent <> *NULL
c                   eval      IP = h_addr
c                   endif
```

Now that we know the basic syntax to call these APIs, lets put them together and write a program that uses them. When we write a client program for TCP/IP, usually the first thing we need to do is figure out the IP address to connect to. Generally, this is supplied by the user, and the user will either supply an IP address directly (in the dotted-decimal format) or he'll supply a hostname that needs to be looked up.

The way that we go about doing this is by first calling 'inet_addr' to see if the host is a valid dotted-decimal format IP address. If it is not, we'll actually call 'gethostbyname' to look the address up using DNS and/or the host table. In our example program, we'll then either print a 'host not found' error message, or we'll call 'inet_ntoa' to get a dotted-decimal format IP address that we can print to the screen.

So here's our example of looking up an IP address for a hostname:

File: SOCKTUT/QRPGLESRC, Member: DNSLOOKUP

```
H DFTACTGRP(*NO) ACTGRP(*NEW)

D inet_addr       PR            10U 0 ExtProc('inet_addr')
D  address_str                    *   value options(*string)

D INADDR_NONE     C                   CONST(4294967295)

D inet_ntoa       PR              *   ExtProc('inet_ntoa')
```

```
     D  internet_addr                10U 0 value

     D p_hostent       S               *
     D hostent         DS                  Based(p_hostent)
     D   h_name                        *
     D   h_aliases                     *
     D   h_addrtype                  10I 0
     D   h_length                    10I 0
     D   h_addr_list                   *
     D p_h_addr        S               *   Based(h_addr_list)
     D h_addr          S             10U 0 Based(p_h_addr)

     D gethostbyname   PR              *   extproc('gethostbyname')
     D   host_name                     *   value options(*string)

     D host            S             32A
     D IP              S             10U 0
     D Msg             S             50A

     c     *entry        plist
     c                   parm                    host

     c                   eval      IP = inet_addr(%trim(host))

     c                   if        IP = INADDR_NONE
     c                   eval      p_hostent = gethostbyname(%trim(host))
     c                   if        p_hostent <> *NULL
     c                   eval      IP = h_addr
     c                   endif
     c                   endif

     c                   if        IP = INADDR_NONE
     c                   eval      Msg = 'Host not found!'
     c                   else
     c                   eval      Msg = 'IP = ' + %str(inet_ntoa(IP))
     c                   endif

     c                   dsply                   Msg

     c                   eval      *inlr = *on
```

This program can be compiled with: `CRTBNDRPG DNSLOOKUP SRCFILE(xxx/QRPGLESRC)`

And you can call it like this: `CALL DNSLOOKUP PARM('www.yahoo.com')` or `CALL DNSLOOKUP PARM('www.scottklement.com')` or with a dotted-decimal address like `CALL DNSLOOKUP PARM('192.168.66.21')`

After all that explanation, the code itself doesn't seem so complicated, does it? :)

Chapter 3. Socket API calls to create our first client program

Written by Scott Klement.

3.1. The socket() API call

The previous sections explained how to find out the port number for a service name, and how to get the IP address for a host name. This section will utilize that information to create a simple client program.

The socket() API is used to create a socket. You can think of a socket as being the virtual device that is used to read & write data from a network connection.

The IBM manual page that documents the socket API is at this link:
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/socket.htm

It lists the following as the prototype for the socket() API:

```
int socket(int address_family,
           int type,
           int protocol)
```

This tells us that the name of the procedure to call is ’socket’, and that it accepts 3 parameters. Each one is an integer, passed by value. It also returns an integer. Therefore, the RPG prototype for the socket() API looks like this:

```
D socket          PR            10I 0 ExtProc('socket')
D  addr_family                  10I 0 value
D  type                         10I 0 value
D  protocol                     10I 0 value
```

It’s important to realize that the socket APIs can be used for other networking protocols besides TCP/IP. When we create a socket, we need to explain to the socket API that we wish to communicate using the IP protocol, and that we wish to use TCP on top of the IP protocol.

For address family, the manual tells us that we need to specify a value of ’AF_INET’ if we wish to do network programming in the ’Internet domain’. Therefore, when we specify a value of ’AF_INET’, what we’re really telling the API is to ’use the IP protocol’.

Under the ’type’ parameter, it allows us to give values of ’SOCK_DGRAM’, ’SOCK_SEQPACKET’, ’SOCK_STREAM’ or ’SOCK_RAW’. The TCP protocol is the standard streaming protocol for use over IP. So, if we say ’SOCK_STREAM’, we’ll use the TCP protocol. As you might imagine, SOCK_DGRAM is used for the UDP protocol and SOCK_RAW is used for writing raw IP datagrams.

Finally, we specify which protocol we wish to use with our socket. Note that, again, we can specify IPPROTO_TCP for TCP, IPPROTO_UDP for UDP, etc. However, this isn’t necessary! Because we already specified that we wanted a ’stream socket’ over ’internet domain’, it already knows that it should be using TCP. Therefore, we can specify ’IPPROTO_IP’ if we want, and the API will use the default protocol for the socket type.

Now, we just have one problem: We don't know what integer values AF_INET, SOCK_STREAM and IPPPROTO_IP are! IBM is referencing named constants that they've defined in the appropriate header files for C programs, but we don't have these defined for us in RPG! But, if you do a bit of snooping into the 'System Openness Includes' library, you'll find that AF_INET is defined to be '2', SOCK_STREAM is defined to be '1', and IPPROTO_IP is defined as '0'. To make this easier for us, we'll make named constants that match these values, like so:

```
D AF_INET         C                   CONST(2)
D SOCK_STREAM     C                   CONST(1)
D IPPROTO_IP      C                   CONST(0)
```

Now we can call the socket() API like so:

```
c                   eval      s = socket(AF_INET:SOCK_STREAM:IPPROTO_IP)
```

3.2. The connect() API call

Once we have a socket to work with, we need to connect it to something. We do that using the connect() API, which is documented in IBM's manual at this location:
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/connec.htm

It tells us here that the prototype for the connect() API looks like this:

```
int connect(int socket_descriptor,
            struct sockaddr *destination_address,
            int address_length)
```

So, as you can see, the procedure is named 'connect', and it accepts 3 parameters. An integer, a pointer to a 'sockaddr' structure, and another integer. It also returns an integer. This means that the RPG prototype will look like this:

```
D connect         PR            10I 0 ExtProc('connect')
D  sock_desc                    10I 0 value
D  dest_addr                      *   value
D  addr_len                     10I 0 value
```

Looking further down the manual, we see that the a 'sockaddr' structure is defined as follows:

```
struct sockaddr {
   u_short sa_family;
   char    sa_data[14];
};
```

Remember, the purpose of this structure is to tell the API which IP address and port number to connect to. Why, then, doesn't it contain fields that we can put the address and port numbers into? Again, we have to remember that the socket APIs can work with many different network protocols. Each protocol has a completely different format for how addresses work. This 'sockaddr' structure is, therefore, a generic structure. It contains a place to put the identifying address family, along with a generic "data" field that the address can be placed in, regardless of the format of the address.

Although it's not documented on IBM's page for the connect() API, there is actually a different structure called 'sockaddr_in' which is designed especially for internet addresses. The C definition for sockaddr_in can be found in the file QSYSINC/NETINET, member IN, if you have the System Openness Includes loaded. It looks like this:

```
struct sockaddr_in {            /* socket address (internet)   */
    short  sin_family;          /* address family (AF_INET)    */
  u_short  sin_port;            /* port number                 */
   struct  in_addr  sin_addr;   /* IP address                  */
     char  sin_zero[8];         /* reserved - must be 0x00's   */
};
```

To make it easier to use these structures in RPG, I like to make them based in the same area of memory. This means that you can look at the data as a 'sockaddr', or the same data as a 'sockaddr_in' without moving the data around. Having said that, here's the definition that I use for the sockaddr & sockaddr_in structures:

```
D p_sockaddr      S               *
D sockaddr        DS                  based(p_sockaddr)
D   sa_family                    5I 0
D   sa_data                     14A
D sockaddr_in     DS                  based(p_sockaddr)
D   sin_family                   5I 0
D   sin_port                     5U 0
D   sin_addr                    10U 0
D   sin_zero                     8A
```

Before we can call the connect() API, we need to ask the operating system for some memory that we can store our sockaddr structure into. Then, we can populate the sockaddr_in structure, and actually call the connect() API. Like so:

```
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++
D p_connto        S               *
D addrlen         S             10I 0

C* Ask the operating system for some memory to store our socket
C* address into:
C                   eval      addrlen = %size(sockaddr)
C                   alloc     addrlen       p_connto

C* Point the socket address structure at the newly allocated
C*   area of memory:
C                   eval      p_sockaddr = p_connto

C* Populate the sockaddr_in structure
C*  Note that IP is the ip address we previously looked up
C*  using the inet_addr and/or gethostbyname APIs
C*  and port is the port number that we looked up using the
C*  getservbyname API.
```

```
C                   eval      sin_family = AF_INET
C                   eval      sin_addr = IP
C                   eval      sin_port = PORT
C                   eval      sin_zero = *ALLx'00'

C*  Finally, we can connect to a server:
C                   if        connect(s: p_connto: addrlen) < 0
C*** Connect failed, report error here
C                   endif
```

3.3. The send() and recv() API calls

Once we've made a connection, we'll want to use that connection to send and receive data across the network. We'll do that using the send() and recv() APIs.

IBM's manual page for the send() API can be found at this link:
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/send.htm

It tells us that the prototype for the send() API looks like this:

```
int send(int socket_descriptor,
         char *buffer,
         int buffer_length,
         int flags)
```

Yes, the procedure is called 'send', and it accepts 4 parameters. Those parameters are an integer, a pointer, an integer and another integer. The send() API also returns an integer. Therefore, the RPG prototype for this API is:

```
D send            PR            10I 0 ExtProc('send')
D   sock_desc                   10I 0 value
D   buffer                        *   value
D   buffer_len                  10I 0 value
D   flags                       10I 0 value
```

You may have noticed that for other 'char *' definitions, we put the 'options(*string)' keyword in our D-specs, but we didn't this time. Why? Because the send() API doesn't use a trailing null-character to determine the end of the data to send. Instead, it uses the buffer_length parameter to determine how much data to send.

That is a useful feature to us, because it means that we are able to transmit the null-character over the network connection as well as the rest of the string, if we so desire.

The flags parameter is used for 'out of band data', and for sending 'non-routed' data. You'll almost never use these flags. Why? Because 'out of band data' has never been widely adopted. Many TCP/IP stacks don't even implement it properly. In fact, for a long time, sending 'out-of-band' data to a Windows machine caused it to crash. The popular program called 'winnuke' does nothing more than send some out-of-band data to a Windows machine. The other flag, 'dont route' is really only used when writing routing applications. In all other situations, you want your packets to be routed! Therefore, it's very rare for us to specify anything but a 0 in the flags parameter.

The return value of the send() API will be the number of bytes sent, or a negative number if an error occurred.

Consequently, we typically call the send() API like this:

```
D miscdata        S             25A
D rc              S             10I 0

C                   eval      miscdata = 'The data to send goes here'
C                   eval      rc = send(s: %addr(miscdata): 25: 0)
c                   if        rc < 25
C*  for some reason we weren't able to send all 25 bytes!
C                   endif
```

The recv() API is used to receive data over the network. IBM has documented this API here: http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/recv.htm

recv() is very similar to send(). In fact, the prototype for recv is nearly identical to send(), the only difference is the name of the procedure that you call. The prototype looks like this:

```
int recv(int socket_descriptor,
         char *buffer,
         int buffer_length,
         int flags)
```

And, just like send, the RPG prototype looks like this:

```
D recv            PR            10I 0 ExtProc('recv')
D   sock_desc                   10I 0 value
D   buffer                        *   value
D   buffer_len                  10I 0 value
D   flags                       10I 0 value
```

The obvious difference between send() and recv() is what the system does with the memory pointed to by the 'buffer' parameter. When using send(), the data in the buffer is written out to the network. When using recv(), data is read from the network and is written to the buffer.

Another, less obvious, difference is how much data gets processed on each call to these APIs. By default, when you call the send() API, the API call won't return control to your program until the entire buffer has been written out to the network. By contrast, the recv() API will receive all of the data that's currently waiting for your application.

By default, recv() will always wait for at least one byte to be received. But, if there are more bytes, it will return them all, up to the length of the buffer that you've requested.

In the send() example above, 25 bytes are always written to the network unless an error has occurred. In the recv() example below, we can receive anywhere from 1 to 25 bytes of data. We have to check the return code of the recv() API to see how much we actually received.

Here's a quick example of calling recv():

```
D miscdata        S             25A
D rc              S             10I 0

C                   eval      rc = recv(s: %addr(miscdata): 25: 0)
c                   if        rc < 1
```

```
C*  Something is wrong, we didnt receive anything.
C                   endif
```

3.4. Translating from ASCII to EBCDIC

Almost all network communications use the ASCII character set, but the AS/400 natively uses the EBCDIC character set. Clearly, once we're sending and receiving data over the network, we'll need to be able to translate between the two.

There are many different ways to translate between ASCII and EBCDIC. The API that we'll use to do this is called QDCXLATE, and you can find it in IBM's information center at the following link: http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/QDCXLATE.htm

There are other APIs that can be used to do these conversions. In particular, the iconv() set of APIs does really a good job, however, QDCXLATE is the easiest to use, and will work just fine for our purposes.

The QDCXLATE API takes the following parameters:

Parm#	Description	Usage	Data Type
1	Length of data to convert	Input	Packed (5,0)
2	Data to convert	I/O	Char (*)
3	Conversion table	Input	Char (10)

And, since QDCXLATE is an OPM API, we actually call it as a program. Traditionally, you'd call an OPM API with the RPG 'CALL' statement, like this:

```
C                   CALL      'QDCXLATE'
C                   PARM      128           LENGTH            5 0
C                   PARM                    DATA            128
C                   PARM      'QTCPEBC'     TABLE            10
```

However, I find it easier to code program calls using prototypes, just as I use for procedure calls. So, when I call QDCXLATE, I will use the following syntax:

```
D Translate       PR                  ExtPgm('QDCXLATE')
D   Length                       5P 0 const
D   Data                     32766A   options(*varsize)
D   Table                       10A   const

C                   callp     Translate(128: Data: 'QTCPEBC')
```

There are certain advantages to using the prototyped call. The first, and most obvious, is that each time we want to call the program, we can do it in one line of code. The next is that the 'const' keyword allows the compiler to automatically convert expressions or numeric variables to the data type required by the call. Finally, the prototype allows the compiler to do more thorough syntax checking when calling the procedure.

There are two tables that we will use in our examples, QTCPASC and QTCPEBC. These tables are easy to remember if we just keep in mind that the table name specifies the character set that we want to translate the data into. In other words ’QTCPEBC’ is the IBM-supplied table for translating TCP to EBCDIC (from ASCII) and QTCPASC is the IBM supplied table for translating TCP data to ASCII (from EBCDIC).

3.5. The close() API call

This section documents the easiest, by far, socket API to call. The close() API. This API is used to disconnect a connected socket, and destroy the socket descriptor. (In other words, to use the socket again after calling close() you have to call socket() again).

Here’s the IBM manual page that describes the close() API:<
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/close.htm

The manual tells us that the prototype for close() looks like this:

```
int close(int fildes)
```

So, the procedure’s name is ’close’ and it accepts one parameter, an integer. It also returns an integer. The RPG prototype looks like this:

```
D close           PR            10I 0 ExtProc('close')
D  sock_desc                    10I 0 value
```

To call it, we can simply to:

```
C                   eval      rc = close(s)
C                   if        rc < 0
C*** Socket didn't close.  Now what?
c                   endif
```

Or, more commonly (because there isn’t much we can do if close() fails) we do something like this:

```
C                   callp     close(s)
```

Too easy to be a UNIX-Type API, right? Well, never fear, there’s one complication. The system uses the same close() API for closing sockets that it uses for closing files in the integrated file system.

This means that if you use both sockets and IFS reads/writes in your program, that you only need to define one prototype for close(). Handy, right? Unfortunately, most people put all of the definitions needed for socket APIs into one source member that they can /COPY into all of the programs that need it. Likewise, the IFS prototypes and other definitions are put into their own /COPY member.

When you try to use both /COPY members in the same program, you end up with a duplicate definition of the close() API, causing the compiler to become unhappy.

The solution is relatively simple... When we make a header file for either sockets or IFS, we use the RPG /define and /if defined directives to force it to only include the close() prototype once. So, our prototype will usually look like this:

```
D/if not defined(CLOSE_PROTOTYPE)
D close           PR            10I 0 ExtProc('close')
D  sock_desc                    10I 0 value
D/define CLOSE_PROTOTYPE
D/endif
```

3.6. Our first client program

We've learned a lot of new API calls over the past few sections. It's time to put these new APIs to use with an example program.

This program is a very simple http client. It connects to a web server on the internet, and requests that a web page (or another file) on the server be sent back to it. It then receives the data that the web server returns and displays it on the screen.

It's important to understand that most data that is sent or received over the internet uses the concept of 'lines of text.' A line of text is a variable-length string of bytes, you can tell the end of a line by looking for the 'carriage-return' and 'line-feed' characters. When these characters appear in the text, it means that its time to start a new line.

In ASCII, the 'carriage-return' character (CR) is x'0D', and the 'line-feed' character is x'0A'. When translated to EBCDIC, these are x'0D' and x'25', respectively.

Therefore, the pseudocode for this client program looks like this:

1. look up the port number for the HTTP service and store it into the variable 'port'.
2. look up the IP address for the hostname that was passed as a parameter, and store it in the variable 'IP'.
3. call the socket() API to create a socket that we can use for communicating with the HTTP server.
4. create a socket address structure ('sockaddr') that will tell the connect() API which host & service to connect to.
5. call the connect() API to connect to the HTTP server.
6. place a two-line 'request' into a variable. The first line will contain the phrase "GET /pathname/filename HTTP/1.0" which tells the HTTP server that we wish to get a file from it, and also tells the HTTP server where that file is. The "HTTP/1.0" means that we're using version 1.0 of the HTTP specifications (more about that later) The second line of the request is blank, that's how we tell the server that we're done sending requests.
7. Translate our request to ASCII so the server will understand it.
8. Call the send() API to send our request to the server.
9. Call the recv() API to read back 1 byte of the server's reply.
10. If an error occurred (that is, the server disconnected us) then we're done receiving, jump ahead to step 13.
11. If the byte that we've read is not the 'end-of-line' character, and our receive buffer isn't full, then add the byte to the end of the receive buffer, and go to step 9.

12. Translate the receive buffer to EBCDIC so we can read it, and display the receive buffer. Then go back to step 9 to get the next line of data.
13. Close the connection.
14. Pause the screen so the user can see what we received before the program ends.

Without further ado, here's the sample program, utilizing all of the concepts from the past few sections of this tutorial:

```
File: SOCKTUT/QRPGLESRC, Member: CLIENTEX1

     H DFTACTGRP(*NO) ACTGRP(*NEW)

     D getservbyname   PR               *   ExtProc('getservbyname')
     D  service_name                    *   value options(*string)
     D  protocol_name                   *   value options(*string)

     D p_servent       S                *
     D servent         DS                   based(p_servent)
     D   s_name                         *
     D   s_aliases                      *
     D   s_port                     10I 0
     D   s_proto                        *

     D inet_addr       PR           10U 0 ExtProc('inet_addr')
     D  address_str                     *   value options(*string)

     D INADDR_NONE     C                    CONST(4294967295)

     D inet_ntoa       PR               *   ExtProc('inet_ntoa')
     D  internet_addr               10U 0 value

     D p_hostent       S                *
     D hostent        DS                    Based(p_hostent)
     D   h_name                         *
     D   h_aliases                      *
     D   h_addrtype                 10I 0
     D   h_length                   10I 0
     D   h_addr_list                    *
     D p_h_addr        S                *   Based(h_addr_list)
     D h_addr          S            10U 0 Based(p_h_addr)

     D gethostbyname   PR               *   extproc('gethostbyname')
     D   host_name                      *   value options(*string)

     D socket          PR           10I 0 ExtProc('socket')
     D  addr_family                 10I 0 value
     D  type                        10I 0 value
     D  protocol                    10I 0 value

     D AF_INET         C                    CONST(2)
     D SOCK_STREAM     C                    CONST(1)
     D IPPROTO_IP      C                    CONST(0)
```

```
D connect         PR              10I 0 ExtProc('connect')
D  sock_desc                      10I 0 value
D  dest_addr                        *   value
D  addr_len                       10I 0 value

D p_sockaddr      S                 *
D sockaddr        DS                    based(p_sockaddr)
D   sa_family                      5I 0
D   sa_data                       14A
D sockaddr_in     DS                    based(p_sockaddr)
D   sin_family                     5I 0
D   sin_port                       5U 0
D   sin_addr                      10U 0
D   sin_zero                       8A

D send            PR              10I 0 ExtProc('send')
D   sock_desc                     10I 0 value
D   buffer                          *   value
D   buffer_len                    10I 0 value
D   flags                         10I 0 value

D recv            PR              10I 0 ExtProc('recv')
D   sock_desc                     10I 0 value
D   buffer                          *   value
D   buffer_len                    10I 0 value
D   flags                         10I 0 value

D close           PR              10I 0 ExtProc('close')
D  sock_desc                      10I 0 value

D translate       PR                    ExtPgm('QDCXLATE')
D   length                         5P 0 const
D   data                       32766A   options(*varsize)
D   table                         10A   const

D msg             S               50A
D sock            S               10I 0
D port            S                5U 0
D addrlen         S               10I 0
D ch              S                1A
D host            s               32A
D file            s               32A
D IP              s               10U 0
D p_Connto        S                 *
D RC              S               10I 0
D Request         S               60A
D ReqLen          S               10I 0
D RecBuf          S               50A
D RecLen          S               10I 0

C*************************************************
C* The user will supply a hostname and file
```

```
C*  name as parameters to our program...
C*************************************************
c     *entry        plist
c                   parm                    host
c                   parm                    file

c                   eval      *inlr = *on

C*************************************************
C* what port is the http service located on?
C*************************************************
c                   eval      p_servent = getservbyname('http':'tcp')
c                   if        p_servent = *NULL
c                   eval      msg = 'Can''t find the http service!'
c                   dsply                   msg
c                   return
c                   endif

c                   eval      port = s_port

C*************************************************
C* Get the 32-bit network IP address for the host
C*  that was supplied by the user:
C*************************************************
c                   eval      IP = inet_addr(%trim(host))
c                   if        IP = INADDR_NONE
c                   eval      p_hostent = gethostbyname(%trim(host))
c                   if        p_hostent = *NULL
c                   eval      msg = 'Unable to find that host!'
c                   dsply                   msg
c                   return
c                   endif
c                   eval      IP = h_addr
c                   endif

C*************************************************
C* Create a socket
C*************************************************
c                   eval      sock = socket(AF_INET: SOCK_STREAM:
c                                           IPPROTO_IP)
c                   if        sock < 0
c                   eval      msg = 'Error calling socket()!'
c                   dsply                   msg
c                   return
c                   endif

C*************************************************
C* Create a socket address structure that
C*   describes the host & port we wanted to
C*   connect to
C*************************************************
c                   eval      addrlen = %size(sockaddr)
c                   alloc     addrlen       p_connto
```

```
c                   eval      p_sockaddr = p_connto
c                   eval      sin_family = AF_INET
c                   eval      sin_addr = IP
c                   eval      sin_port = port
c                   eval      sin_zero = *ALLx'00'

C*************************************************
C* Connect to the requested host
C*************************************************
C                   if        connect(sock: p_connto: addrlen) < 0
c                   eval      msg = 'unable to connect to server!'
c                   dsply                   msg
c                   callp     close(sock)
c                   return
c                   endif

C*************************************************
C* Format a request for the file that we'd like
C* the http server to send us:
C*************************************************
c                   eval      request = 'GET ' + %trim(file) +
c                               ' HTTP/1.0' + x'0D25' + x'0D25'
c                   eval      reqlen = %len(%trim(request))
c                   callp     Translate(reqlen: request: 'QTCPASC')

C*************************************************
c*  Send the request to the http server
C*************************************************
c                   eval      rc = send(sock: %addr(request): reqlen:0)
c                   if        rc < reqlen
c                   eval      Msg = 'Unable to send entire request!'
c                   dsply                   msg
c                   callp     close(sock)
c                   return
c                   endif

C*************************************************
C* Get back the server's response
C*************************************************
c                   dou       rc < 1
C                   exsr      DsplyLine
c                   enddo

C*************************************************
C*  We're done, so close the socket.
C*   do a dsply with input to pause the display
C*   and then end the program
C*************************************************
c                   callp     close(sock)
c                   dsply                   pause             1
c                   return
```

```
C*===============================================================
C* This subroutine receives one line of text from a server and
C*  displays it on the screen using the DSPLY op-code
C*===============================================================
CSR   DsplyLine     begsr
C*-------------------------
C************************************************
C* Receive one line of text from the HTTP server.
C*  note that "lines of text" vary in length,
C*  but always end with the ASCII values for CR
C*  and LF.  CR = x'0D' and LF = x'0A'
C*
C* The easiest way for us to work with this data
C* is to receive it one byte at a time until we
C* get the LF character.   Each time we receive
C* a byte, we add it to our receive buffer.
C************************************************
c                   eval      reclen = 0
c                   eval      recbuf = *blanks

c                   dou       reclen = 50 or ch = x'0A'
c                   eval      rc = recv(sock: %addr(ch): 1: 0)
c                   if        rc < 1
c                   leave
c                   endif
c                   if        ch<>x'0D' and ch<>x'0A'
c                   eval      reclen = reclen + 1
c                   eval      %subst(recbuf:reclen:1) = ch
c                   endif
c                   enddo

C************************************************
C* translate the line of text into EBCDIC
C* (to make it readable) and display it
C************************************************
c                   if        reclen > 0
c                   callp     Translate(reclen: recbuf: 'QTCPEBC')
c                   endif
c     recbuf        dsply
C*-------------------------
Csr                 endsr
```

Compile this program with: `CRTBNDRPG PGM(CLIENTEX1) SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)`

Run the program by typing: `CALL CLIENTEX1 PARM('ods.ods.net' '/index.html')`

(You should be able to use this to retrieve just about any web page)

There are a lot of things that we could improve about this client. We'll discuss these, and start implementing the improvements, in the upcoming sections.

Chapter 4. Improving our client program

Written by Scott Klement.

4.1. improvement #1: header files

Now that we've had a "basic example" of sending and receiving some information from a server program, let's take a look at how we can improve our example.

The first improvement that we should make will involve converting all of the prototypes, constants and data structures that we use for sockets into a "header file", and this will be described in this section:

First of all, our example contains about 75 lines of code that are simply prototypes, constants and data structures which will be used in just about every program that we write using sockets. One way that we can improve our program is to place them into their own source member, use the /copy compiler directive to add them into our programs. This gives us the following benefits:

- Each time we want to use them, we just /copy them. We don't have to re-type them, and if we find a mistake in on of our definitions, we only have to fix it in one place.
- We don't have to remember every detail of every API we call, by putting the prototypes into a /copy member, we can use that copy member as a "cheat-sheet" of how to call a given socket function. In fact, with judicious use of comments, we can often save ourselves a trip to the manuals...
- programmers also have "header files" which contain the definition that they use for calling system functions. Using the same type of header files in RPG makes the API more consistent across languages, and makes the IBM reference manuals more applicable to our situation.
- In fact, in each of the IBM manual pages, we'll see lines that say '#include <sys/socket.h>' or '#include <errno.h>' and these are telling the C programmer which header files are necessary for a given API call.

When I create an "API header file", I generally give the member a descriptive name followed by a "_H", which means "header". So my header member relating to sockets is called "SOCKET_H".

Here is an example of what a prototype generally looks like in my header member:

```
 **-----------------------------------------------------------------
 **  struct servent *getservbyname(char *service_name,
 **                                char *protocol_name);
 **
 **   Retrieve's a service entry structure for a given service
 **   and protocol name.  (Usually used to get a port number for
 **   a named service)
 **
 **   service_name = name of service to get (e.g.: 'http' or 'telnet')
 **  protocol_name = protocol that service runs under ('tcp', 'udp')
 **
 **  Returns *NULL upon failure, otherwise a pointer to the service
 **     entry structure
 **-----------------------------------------------------------------
D getservbyname   PR              *   ExtProc('getservbyname')
D  service_name                   *   value options(*string)
```

```
D  protocol_name                  *   value options(*string)
```

So you can see that it tells me a lot of information. How the prototype was defined in C, a basic description of the function that I'm calling, a description of the parameters, and a description of the return values.

When I need constants in my header file, I'll usually group them according to the API that uses them, and give each a short description, like this:

```
 * Address Family of the Internet Protocol
D AF_INET         C                   CONST(2)
 * Socket type for reading/writing a stream of data
D SOCK_STREAM     C                   CONST(1)
 * Default values of Internet Protocol
D IPPROTO_IP      C                   CONST(0)
```

When I place data structures into a header file, I'll do it like this:

```
**
** Service Database Entry (which service = which port, etc)
**
**            struct servent {
**              char   *s_name;
**              char   **s_aliases;
**              int    s_port;
**              char   *s_proto;
**            };
**
D p_servent       S               *
D servent         DS                  Based(p_servent)
D   s_name                        *
D   s_aliases                     *
D   s_port                      10I 0
D   s_proto                       *
```

I'm planning (when time permits) to go through each of my data structures and add a description of each field in the structure, as well. I simply haven't had a chance yet.

You'll also notice that I always declare my data structures (or at least, those I use with the UNIX-type APIs) to be BASED() on a pointer. I do that for three reasons:

1. many of these structures (in particular, servent and hostent) are "returned" from an API. That means that the API actually allocates the memory for them, rather than my program. So, I need to use a pointer so I can refer to the memory that was allocated by that API.
2. As alluded to in point #1, when something is BASED() the system doesn't allocate memory to it. Since the header will usually have many different structures that may not be used in every program, it saves memory to only allocate it when needed.
3. In all but the simplest programs, we will need more than one 'instance' of each data structure. For example, if we wanted to both call bind() to bind our socket to a given port number, and call connect() to connect to a

remote port in the same program, we'll need two copies of the sockaddr_in data structure. One containing the address & port number to bind to, the other containing the address & port number to connect to.

By using BASED() data structures we can do that, simply by allocating two different buffers, one called "connto" and one called "bindto". We can set the address of the sockaddr_in data structure to the area of memory that bindto is in when we want to change or examine the contents of the bindto data, and likewise set the address of sockaddr_in to the area of memory that connto is in when we want to examine or change the connto area.

Point #3 is often difficult for an RPG programmer to understand. You can think of it as being similar in concept to using the MOVE op-code to put data from a character string into a data structure to break the data into fields. It's a little different, however, in that you're not copying all of the data from one place to another, instead you're referencing it in it's original place in memory. Consequently, you don't have to move it back after you've changed it.

Here's a quick example of code snippets that use the functionality that I'm trying to describe in Point #3:

```
D connto          S               *
D bindto          S               *
D length          S             10I 0

C* How much memory do we need for a sockaddr_in structure?
C*
c                   eval      length = %size(sockaddr_in)

C* Allocate space for a 'connect to' copy of the structure:
C*
c                   alloc(e)  length        connto
c                   if        %error
C**** No memory left?!
c                   endif

C* Allocate space for a 'bind to' copy of the structure:
C*
c                   alloc(e)  length        bindto
c                   if        %error
C**** No memory left?!
c                   endif

C* Set the values of the connto structure:
c                   eval      p_sockaddr = connto
c                   eval      sin_family = AF_INET
c                   eval      sin_addr   = some_ip_address
c                   eval      sin_port   = some_port_number
c                   eval      sin_zero   = *ALLx'00'

C* Set the values of the bindto structure:
C*  Note that each time we do a 'p_sockaddr = XXX' we are effectively
C*  "switching" which copy of the sockaddr_in structure we're working
C*  with.
c                   eval      p_sockaddr = bindto
c                   eval      sin_family = AF_INET
c                   eval      sin_addr   = INADDR_ANY
c                   eval      sin_port   = some_port_number
```

```
c                   eval      sin_zero   = *ALLx'00'

C* call connect()
c                   if        connect(sock1: connto: length) < 0
c* Error!
c                   endif

C* call bind()
c                   if        bind(sock2: bindto: length) < 0
c* Error!
c                   endif

C* examine the contents of connect() version
c                   eval      p_sockaddr = connto
c                   if        sin_port = 80
c*  Do somethign special
c                   endif
```

Hopefully, by now, you have an idea of how our /copy member (i.e. "API header member") should work. You should be able to go back to the example programs in previous topics in this tutorial and create your own header file.

If you prefer, I will make available the header file that I use for my own sockets programs. This will save you having to create your own, if you do not wish to. You can find my header file here: http://www.scottklement.com/rpg/socktut/qrpglesrc.socket_h

After this point, all of the example programs will use my socket_h header member. Each section that explains a new API call will describe what entries you need to add to your own header, if you're writing your own.

4.2. improvement #2: error handling

The next improvement that our client program certainly needs is the ability to give more specific and meaningful error messages when an API call does not work. This section explains how error handling works in the UNIX-type APIs and explains how to use this in your RPG programs.

In C, there is a global variable called 'errno' that is set to an error number whenever a procedure has an error. Almost all C procedures return either an integer or a pointer, therefore, by convention, when an error occurs a C procedure will return either '-1' or 'NULL'. The programmer will use this as an indication that he needs to check the value of the 'errno' global variable to see which error occurred. (Although most C procedures follow this convention, some don't. Check the manual page for the particular procedure before handling the error)

The 'errno' variable is an integer. It can only contain a number, which is usually different on different platforms. To make things more compatible across platforms, constants are placed into a C 'header file', and programmers are encouraged to use constants when checking for errors.

As an example, lets go back to the IBM manual page for the socket() API. For convienience, it can be found here: http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/socket.htm

Page down to the section entitled 'Error Conditions', it states: `When socket() fails, errno can be set to one of the following:`

The proceeds to list things like "EACCES", "EAFNOSUPPORT", "EIO", "EMFILE", along with descriptions of what errors they refer to. Each one of these is actually a constant defined in in a C header file that is translated by the compiler into a number.

In fact, if you have the System Openness Includes licensed program installed, you can bring up the ERRNO member of the file called SYS in library QSYSINC and you'll be able to see how this is done in C programs. EACCESS corresponds to the number 3401, EMFILE to 3452, etc. Unfortunately, some of these are not specifically "UNIX-Type API" errors, but are generic C language errors. So, for those values, you also need to look at the member called ERRNO in the file H in library QCLE. This file is part of the ILE C/400 licensed program, and is only available if you have that licpgm installed.

If you are fortunate enough to have the file 'H' in 'QCLE' installed on your machine, you'll also see the following definition the end:

```
#define errno (*__errno())
```

This is a macro that runs at compile time that changes the word 'errno' wherever it's used in a C program, to actually be '(*__errno())'. This is very useful to us because it means that whenever a C program refers to the global variable 'errno', it's actually referring to the return value from a procedure called '__errno()'. That means, if we want to retrieve errno in an RPG program, all we have to do is call that __errno() function ourselves!

For your convienience (and, for mine!) I've defined all of the C errno values from both QCLE and QSYSINC into a single RPG /copy member. I also used some RPG compiler directives to include an errno procedure that will make it much more intuitive to use errno in our RPG programs.

You can get a copy of this /copy member (called errno_h) at this link:
http://www.scottklement.com/rpg/socktut/qrpglesrc.errno_h

At first glance, you might be wondering "Why didn't you simply make this a part of the existing socket_h member?" The reason is quite simple: These error routines are used for all UNIX-Type APIs, not just sockets. And, in fact, these routines are also used when calling any function from the ILE C runtime libraries. Consequently, there are many times when we'll want to use the ERRNO_H header member when we aren't working with sockets at all.

To use this, you need to /copy the member twice. Once where you need the definitions of the constants (i.e., in your D-specs), and once where you need the errno procedure (i.e. at the end of the program).

Here's a quick code snippet:

```
H BNDDIR('QC2LE')

D/copy qrpglesrc,socket_h
D/copy qrpglesrc,errno_h

   .... more code is found here ....

c                   if        socket(AF_INET:SOCK_STREAM:IPPROTO_IP)<0
c
c                   select
c                   when      errno = EACCES
c                   eval      Msg = 'Permission Denied'

c                   when      errno = EAFNOSUPPORT
c                   eval      Msg = 'Address Family not supported!'
```

```
c                   when      errno = EIO
c                   goto      TryAgainMaybe

   ... etc, etc, etc ...

c                   endsl

   .... more code is found here ....

 /define ERRNO_LOAD_PROCEDURE
 /copy libsor/qrpglesrc,errno_h
```

Note that because '__errno()' is part of the C-runtime library (as opposed to being part of the UNIX-Type APIs) we need to include the QC2LE binding directory when compiling our program. Otherwise, it won't find the API.

You see that in this example, the program is trying again (maybe) when it gets an I/O error. For the other errors, it's simply assigning a human-readable error message that could later be used to tell the user and/or programmer what the problem was.

As you can imagine, it can get quite tedious to try to have the program handle every possible error that can occur. Especially when, in most cases, all we do is assign the human-readble error message to a variable. Fortunately, there is already a procedure we can call from the C runtime library that will return a human-readable error message for any errno value.

This procedure is called 'strerror' ("string error") and like all C procedures that return strings, it returns a pointer to a an area of memory containing a string. The string is variable length, and we know when we've reached the end of the string because we encounter a 'NULL' character (x'00'). As we've done before, we can use the RPG built-in-function "%str" to decode this string for us.

So, to call strerror from an RPG program (the prototype is already defined in that /copy member I mentioned earlier) we can simply do something like this:

```
H BNDDIR('QC2LE')

D/copy qrpglesrc,socket_h
D/copy qrpglesrc,errno_h

   .... more code is found here ....

c                   if        socket(AF_INET:SOCK_STREAM:IPPROTO_IP)<0
c
c                   if        errno = EIO
c                   goto      TryAgainMaybe
c                   else
c                   eval      Msg = %str(strerror(errno))
c                   endif
c
c                   dsply                   Msg
c
c                   endif

   .... more code is found here ....
```

```
/define ERRNO_LOAD_PROCEDURE
/copy libsor/qrpglesrc,errno_h
```

This will return the human readable error message for everything except for the ’EIO’ error. For EIO, it’ll go back and ’Try Again Maybe’.

4.3. improvement #3: Creating a Read Line utility

Now, we’ve got two "API header members", one for socket calls, and one for error handling, this will allow us to greatly simplify our sockets programs, making them much easier for other programmers to read.

However, looking at the code, there’s still one area that’s a bit difficult to read, and that’s the process of reading one "line" at a time from a socket.

You may recall that almost all almost all of the communications that we’ll have with internet servers will involve sending and receiving "lines ASCII of text." In fact, this is SO common that I couldn’t come up with a simple client program that didn’t use them!

Since most RPG programmers are used to thinking of data in terms of "records" and "fields", it might be useful to think of a line of ASCII text as being a "record". These records are, however, almost always variable-length. You determine where the end of the record is by looking for an ’end of line sequence’, which consists of one or two control characters.

There are 3 popular end of line sequences. The first one is that each line ends with the CR (Carriage Return) character. This is most commonly used in Apple computers. It is based on the way a typewriter works, when you press the ’Carriage Return’ key on a typewriter, the paper feeds up by one line, and then returns to the leftmost position on the page. CR is ASCII value 13 (or Hex x’0D’)

The next end of line sequence is the one used by Unix. The ASCII standard states that ends of lines of text should end with the LF (’line feed’) character. Many people, especially C programmers, will refer to the LF character as the ’Newline’ character. LF is ASCII value 10 (or Hex x’0A’)

Finally, we have the end of line sequence used by DOS and Windows. As noted above, a typewriter feeds to the next line when CR is pressed. However, most early printers could "overtype" a line of text, so when you send the CR character it would go back to the start of the SAME line. The LF sequence would feed the paper out one line, but would not return the carriage to the leftmost position. So, to start a new line of text required two characters, a CR followed by an LF. DOS (and eventually Windows) kept this standard by requiring a line of text to end with both the CR character, followed by the LF character. This was a nice compromise between the up & coming UNIX operating system that DOS was inspired by, and the Apple II DOS, which was at that time the most popular home computer in the world.

So enough history, already! The point of all this is, there are many different ways to send and receive lines of text, what we need is a way to read them in an RPG program. In some cases, we’ll want to be able to specify different end-of-line sequences, and in some cases we’ll want to have the text automatically translated from ASCII to EBCDIC for us to work with.

So, lets write a routine to do this. We’ll make it a subprocedure and put it into a service program, since we’ll want to do this in virtually every sockets program that we write.

To make this as much like the recv() API as possible, we'll allow the caller to pass us a socket, a pointer, and a max length as our first 3 parameters. We'll also return the length of the data read, just as recv() does, or -1 upon error. So, the prototype for this new routine will look like this (so far):

```
D RdLine          PR            10I 0
D   peSock                      10I 0 value
D   peLine                        *   value
D   peLength                    10I 0 value
```

(Note that in the naming conventions of the company that I work for, 'pe' at the start of a variable means 'parameter')

Next, we'll give optional parameters that can be used to tell our subprocedure to translate the data to EBCDIC, and/or change what the characters for line feed and newline are. This will make our prototype look more like this:

```
D RdLine          PR            10I 0
D   peSock                      10I 0 value
D   peLine                        *   value
D   peLength                    10I 0 value
D   peXLate                      1A   const options(*nopass)
D   peLF                         1A   const options(*nopass)
D   peCR                         1A   const options(*nopass)
```

Now, if the optional parameters are not passed to us, we need to use reasonable default values. That is, we'll use x'0D' for CR, x'0A' for LF and *OFF (do not translate) for the peXLate parm. This is easy to do by checking how many parms were passed. Like so:

```
c                   if        %parms > 3
c                   eval      wwXLate = peXLate
c                   else
c                   eval      wwXLate = *OFF
c                   endif

c                   if        %parms > 4
c                   eval      wwLF = peLF
c                   else
c                   eval      wwLF = x'0A'
c                   endif

c                   if        %parms > 5
c                   eval      wwCR = peCR
c                   else
c                   eval      wwCR = x'0D'
c                   endif
```

(Note that in the naming conventions of the company that I work for, 'ww' at the start of a variable means 'work field that is local to a subprocedure')

Then, just like in the DsplyLine routine that we used in our example client, we'll read one character at a time until we receive a LF character. To make the routine simple, we'll simply drop any CR characters that we come across. (That way, if LF=x'0A' and CR=x'0D', this routine will read using either UNIX or Windows end of line characters without any problems. If you wanted to make it work with Apple or Windows end of lines, you'd simply reverse the values passed as CR and LF)

Finally, if the XLate parameter is set to ON, we'll translate the line to EBCDIC. If not, we'll leave it as it was originally sent. So, the finished routine will look like this:

(This should go in member SOCKUTILR4 in the file QRPGLESRC)

```
P RdLine          B                   Export   ❶
D RdLine          PI            10I 0
D   peSock                      10I 0 value
D   peLine                        *   value
D   peLength                    10I 0 value
D   peXLate                      1A   const options(*nopass)
D   peLF                         1A   const options(*nopass)
D   peCR                         1A   const options(*nopass)

D wwBuf           S          32766A   based(peLine)
D wwLen           S             10I 0
D RC              S             10I 0
D CH              S              1A
D wwXLate         S              1A
D wwLF            S              1A
D wwCR            S              1A

 ❷

 ** Set default values to unpassed parms:
c                   if        %parms > 3
c                   eval      wwXLate = peXLate
c                   else
c                   eval      wwXLate = *OFF
c                   endif

c                   if        %parms > 4
c                   eval      wwLF = peLF
c                   else
c                   eval      wwLF = x'0A'
c                   endif

c                   if        %parms > 5
c                   eval      wwCR = peCR
c                   else
c                   eval      wwCR = x'0D'
c                   endif

 ** Clear "line" of data:  ❸
c                   eval      %subst(wwBuf:1:peLength) = *blanks

c                   dow       1 = 1

 ** read 1 byte:
c                   eval      rc = recv(peSock: %addr(ch): 1: 0)
c                   if        rc < 1
```

```
c                   if        wwLen > 0
c                   leave
c                   else
c                   return    -1
c                   endif
c                   endif

 ** if LF is found, we're done reading:
c                   if        ch = wwLF
c                   leave
c                   endif

 ** any other char besides CR gets added to the string:
c                   if        ch <> wwCR
c                   eval      wwLen = wwLen + 1
c                   eval      %subst(wwBuf:wwLen:1) = ch
c                   endif

 ** if variable is full, exit now -- there's no space left to read data into
c                   if        wwLen = peLength ❹
c                   leave
c                   endif

c                   enddo

 ** if ASCII->EBCDIC translation is required, do it here
c                   if        wwXLate=*ON  and wwLen > 0
c                   callp     Translate(wwLen: wwBuf: 'QTCPEBC')
c                   endif

 ** return the length
c                   return    wwLen
P                 E
```

Personally, I learn things better by typing them in, rather than reading them. Therefore, I recommend that you type the code examples in this tutorial in yourself. However, if you'd like, you can download my copy of SOCKUTILR4 here: http://www.scottklement.com/rpg/socktut/qrpglesrc.sockutilr4

A few notes:

❶ the opening P-spec declares this procedure with "Export", so we can make it callable from outside of our service program.

❷ The 'Translate' prototype isn't defined here. That's because we'll make it global to the entire service program.

❸ Before the recv() loop, we clear the buffer, but we only clear from the start to the 'length' that was passed into our program. This is very important, since if the calling program should send us a variable that's not the entire 32k, we don't want to clear data that might not be allocate to the variable...

❹ Although we let the caller pass a line length, as written, this procedure has an effective maximum line length of 32,766 chars per line. We COULD make it do an 'unlimited' number by using pointer math, but this would make it more difficult to clear the variable at the start, and at any rate, it'd be very unusual for a program to need to read a line of text bigger than 32k.

4.4. improvement #4: Creating a Write Line utility

In the last section, we wrote a routine that would read lines of text into a variable. This would make a nice utility routine that we can put into a service program.

In this section, we'll add another utility routine, one that can be used to send lines of text.

The requirements of this routine are a little different, because it's output-only. That means we don't need to write any data to a buffer, just take data that we already have and send it out. Since the routines that are calling this will often want to pass literals, constants, and expressions, it makes sense for us to accept an 'const' parameter, rather than a pointer.

Also, it'd be a bit of a pain, especially when working with literals, for the calling routine to have to calculate the length of the string each time it calls us. So, instead we'll figure that out ourselves by looking for the last non-blank character. We can still give an optional 'length' parameter, just in case the caller wishes to override this.

This method of searching for the last 'non-blank' character will be slow in most cases with a 32k buffer. Since 99% of the time, a line is less than 80 bytes long, this means that it'd have to search over 32,000 blank spaces each time we send a line. To make this a bit more realistic, the buffer on a call to the 'send line' procedure will only be 256 bytes.

Like the 'read line' routine, though, we still want to allow the caller the option of specifying different end-of-line characters. In RdLine() if the caller only wants one character to signify end-of-line, they can still do so... just make the extra parm be the same char, it won't hurt anything. When sending, however, that would cause problems. So, we'll determine how many chars to add for end-of-line based on how many parms are passed, instead of trying to use both like we did in RdLine().

I think that pretty much sums up the requirements. Here's the routine I actually came up with, check it out:

(This should be added to the member SOCKUTILR4 in the file QRPGLESRC)

```
P WrLine          B                   Export
D WrLine          PI            10I 0
D  peSock                       10I 0 value
D  peLine                      256A   const
D  peLength                     10I 0 value options(*nopass)
D  peXLate                       1A   const options(*nopass)
D  peEOL1                        1A   const options(*nopass)
D  peEOL2                        1A   const options(*nopass)

D wwLine          S            256A
D wwLen           S             10I 0
D wwXlate         S              1A
D wwEOL           S              2A
D wwEOLlen        S             10I 0
D rc              S             10I 0

C*******************************************
C* Allow this procedure to figure out the
C*  length automatically if not passed,
C*  or if -1 is passed.
C*******************************************
c                   if        %parms > 2 and peLength <> -1
c                   eval      wwLen = peLength
c                   else
c                   eval      wwLen = %len(%trim(peLine))
```

```
c                   endif

C******************************************
C* Default 'translate' to *ON.  Usually
C*  you want to type the data to send
C*  in EBCDIC, so this makes more sense:
C******************************************
c                   if        %parms > 3
c                   eval      wwXLate = peXLate
c                   else
c                   eval      wwXLate = *On
c                   endif

C******************************************
C* End-Of-Line chars:
C*   1) If caller passed only one, set
C*         that one with length = 1
C*   2) If caller passed two, then use
C*         them both with length = 2
C*   3) If caller didn't pass either,
C*         use both CR & LF with length = 2
C******************************************
c                   eval      wwEOL = *blanks
c                   eval      wwEOLlen = 0

c                   if        %parms > 4
c                   eval      %subst(wwEOL:1:1) = peEOL1
c                   eval      wwEOLLen = 1
c                   endif

c                   if        %parms > 5
c                   eval      %subst(wwEOL:2:1) = peEOL2
c                   eval      wwEOLLen = 2
c                   endif

c                   if        wwEOLLen = 0
c                   eval      wwEOL = x'0D0A'
c                   eval      wwEOLLen = 2
c                   endif

C******************************************
C* Do translation if required:
C******************************************
c                   eval      wwLine = peLine
c                   if        wwXLate = *On and wwLen > 0
c                   callp     Translate(wwLen: wwLine: 'QTCPASC')
c                   endif

C******************************************
C* Send the data, followed by the end-of-line:
C* and return the length of data sent:
C******************************************
c                   if        wwLen > 0
```

```
c                   eval      rc = send(peSock: %addr(wwLine): wwLen:0)
c                   if        rc < wwLen
c                   return    rc
c                   endif
c                   endif

c                   eval      rc = send(peSock:%addr(wwEOL):wwEOLLen:0)
c                   if        rc < 0
c                   return    rc
c                   endif

c                   return    (rc + wwLen)
P                 E
```

Personally, I learn things better by typing them in, rather than reading them. Therefore, I recommend that you type the code examples in this tutorial in yours elf. However, if you'd like, you can download my copy of SOCKUTILR4 here: http://www.scottklement.com/rpg/socktut/qrpglesrc.sockutilr4

4.5. Packaging our utilities into a service program

Now that we've written our RdLine() and WrLine() procedures, a few other details should be ironed out:

1. We need a prototype for Translate() added to the top of the service program.
2. We need to include our socket_h header file in our service program.
3. We need binding language source to use when creating the service program.

I'll assume at this point that you've already coded the RdLine() and WrLine() procedures from the previous chapters. At the top of the member that you placed those subprocedures in, you'll need the following code:

```
H NOMAIN

D/COPY SOCKTUT/QRPGLESRC,SOCKET_H
D/COPY SOCKTUT/QRPGLESRC,SOCKUTIL_H

D Translate       PR                  ExtPgm('QDCXLATE')
D    peLength                    5P 0 const
D    peBuffer                32766A   options(*varsize)
D    peTable                    10A   const
```

You'll also want to create a member that contains the prototypes for the RdLine and WrLine functions. We'll call this SOCKUTIL_H, and put that in QRPGLESRC as well. It will look like this:

```
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 * RdLine(): This reads one "line" of text data from a socket.
 *
 *   peSock = socket to read from
 *   peLine = a pointer to a variable to put the line of text into
 *   peLength = max possible length of data to stuff into peLine
```

```
 *   peXLate = (default: *OFF) Set to *ON to translate ASCII -> EBCDIC
 *   peLF (default: x'0A') = line feed character.
 *   peCR (default: x'0D') = carriage return character.
 *
 *  returns length of data read, or -1 upon error
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
D RdLine          PR            10I 0
D   peSock                      10I 0 value
D   peLine                        *   value
D   peLength                    10I 0 value
D   peXLate                      1A   const options(*nopass)
D   peLF                         1A   const options(*nopass)
D   peCR                         1A   const options(*nopass)


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  WrLine() -- Write a line of text to a socket:
 *
 *      peSock = socket descriptor to write to
 *      peLine = line of text to write to
 *    peLength = length of line to write (before adding CRLF)
 *            you can pass -1 to have this routine calculate
 *            the length for you (which is the default!)
 *     peXlate = Pass '*ON' to have the routine translate
 *            this data to ASCII (which is the default) or *OFF
 *            to send it as-is.
 *      peEOL1 = First character to send at end-of-line
 *            (default is x'0D')
 *      peEOL2 = Second character to send at end-of-line
 *            (default is x'0A' if neither EOL1 or EOL2 is
 *            passed, or to not send a second char is EOL1
 *            is passed by itself)
 *
 * Returns length of data sent (including end of line chars)
 *    returns a short count if it couldnt send everything
 *    (if you're using a non-blocking socket) or -1 upon error
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
D WrLine          PR            10I 0
D  peSock                       10I 0 value
D  peLine                      256A   const
D  peLength                     10I 0 value options(*nopass)
D  peXLate                       1A   const options(*nopass)
D  peEOL1                        1A   const options(*nopass)
D  peEOL2                        1A   const options(*nopass)
```

If you'd prefer to download my copy of SOCKUTIL_H instead of writing your own, you can get it here: http://www.scottklement.com/rpg/socktut/qrpglesrc.sockutil_h

Finally, we need to create binding source to tell the system how to create our service program. If you're not familiar with this, I'll explain it a little.

When creating a service program, the system calculates a ’signature’ for your service program. This works in a similar manner to the way ’record format level checks’ work on a database file -- if something in the service program changes, they prevent you from possibly accessing it incorrectly.

Binding language serves to tell the system which procedures are exported, and also to tell the system which procedures were exported LAST time you ran CRTSRVPGM, and maybe the time before that, and so on. If you don’t specify binding language for your service program, each time a procedure changes in it, you’ll have to recompile ALL of the programs that call that service program. If you don’t, you’ll get "Signature Violation" errors when trying to call them! But, thanks to the ability to keep track of ’previous exports’ in a service program, you can make your service program backwards compatible.

I hope that’s clear enough -- if not, you probably want to read the ’ILE Concepts’ manual, as this tutorial really isn’t intended to teach all of the nuances of service programs and signatures.

At any rate, it’s very easy to create binding source. It looks like this:

```
Member SOCKUTILR4 of file QSRVSRC:

STRPGMEXP PGMLVL(*CURRENT)
     EXPORT SYMBOL(RDLINE)
     EXPORT SYMBOL(WRLINE)
ENDPGMEXP
```

That’s it... it’s that simple... we’ve told that this is the program exports for the current level of our service program, and which procedures to export. It can hardly be easier!

Now, we compile our service program by typing:

`CRTRPGMOD MODULE(SOCKUTILR4) SRCFILE(SOCKTUT/QRPGLESRC) DBGVIEW(*LIST)`

`CRTSRVPGM SRVPGM(SOCKUTILR4) EXPORT(*SRCFILE) SRCFILE(SOCKTUT/QSRVSRC)`

And, while we’re at it, lets create a binding directory to make our service program available to the CRTBNDRPG command. We do this by typing:

`CRTBNDDIR BNDDIR(SOCKTUT/SOCKUTIL) TEXT(’Socket Utility Binding Directory’)`

`ADDBNDDIRE BNDDIR(SOCKTUT/SOCKUTIL) OBJ((SOCKTUT/SOCKUTILR4 *SRVPGM))`

4.6. improvement #5: Stripping out the HTTP response

One of the things that this tutorial has completely glossed over so far is the HTTP protocol itself. This section attempts to answer the following questions about the HTTP protocol:

- How do we know that we need to send ’GET ’ followed by a filename, followed by ’HTTP/1.0’ to request a file?
- Why do we send an extra x’0D0A’ after our request string?
- Why is there extra data appearing before the actual page that I tried to retrieve?

First, a bit of background:

If you listen to the hype and the buzzwords circulating through the media, you might get the idea that ’The Internet’ and ’The World Wide Web’ are synonyms. In fact, they really aren’t.

The Internet is the large, publicly-accessible, network of computers that are connected together using the TCP/IP protocol suite. The Internet is nothing more or less than a very large network of computers. All it does is allow programs to talk to other programs.

The World Wide Web is one application that can be run over The Internet. The web involves hypertext documents being transported from a web server to a PC where they can be viewed in a web browser. So, you have two programs communicating with each other. A client program ("the web browser") and a server program ("the web server").

The client and server programs use the TCP protocol to connect up with each other. Once they are connected, the client will need to know what data actually needs to be sent to the server in order to get back the document that it wants to display. Likewise, the web server has to understand the client’s requests, and has to know what an acceptable response will be. Clearly, a common set of "command phrases" and "responses" must be understood by both the client and the server in order for them to get meaningful work done. This set of phrases and responses is called an ’application-level protocol.’ In other words, it’s the protocol that web applications use to talk to each other.

The web’s protocol is offically called ’HyperText Transfer Protocol’ (or HTTP for short) and it is an official Internet standardized protocol, developed by the Internet Engineering Task Force. The exact semantics of HTTP are described in a document referred to as a ’Request For Comments’ document, or RFC for short.

The web is, of course, only one of thousands of applications that can utilize the Internet. All of these have their own Application Level Protocol. Most of these are also based on an official internet standard described as an RFC. Some examples of these other protocols include: File Transfer Protocol (FTP), Telnet Protocol, Simple Mail Transport Protocol (SMTP), Post Office Protocol (POP), etc.

All of the RFCs that describe Internet Protocols are available to the public at this site: http://www.rfc-editor.org

(Whew!) Back to the questions, then:

Q: "How do we know that we need to send ’GET ’ followed by a filename, followed by ’HTTP/1.0’ to request a file?"

A: We know how the HTTP protocol works by looking at RFC 2616. There are many, many exact details that should be taken into account to write a good HTTP client, so if you’re looking to write a professional quality application, you should make sure that you’ve read RFC 2616 very carefully.

Q: Why do we send an extra x’0D0A’ after our request string?

A: RFC 2616 refers to a ’request chain’, and a ’response chain’. Which involves the client sending a request (in our case a GET request) followed by zero or more pieces of useful information for the web server to use when delivering the document. The request chain ends when a blank line is send to the server. Since lines of ASCII text always end with x’0D’ (CR) followed by x’0A’ (LF), it looks like a blank line to the web server, and terminates our ’request chain.’

Q: Why is there extra data appearing before the actual page that I tried to retrieve?

A: The server replies with a ’response chain’. Like the ’request chain’ it contains a list of one or more responses, terminated with a blank line. However, in our simple client program, we were not trying to interpret these responses, but merely displaying them to the user.

Finally! Let’s make an improvement!

Completely implementing the HTTP protocol in our sample client would be a bit too much for this tutorial. After all, this is a tutorial on socket programming, not HTTP! However, a simple routine to strip the HTTP responses from the data that we display should be simple enough, so lets do that.

Instead of the ’DsplyLine’ subroutine in our original client, we’ll utilize our nifty new ’rdline’ procedure. We’ll read back responses from the server until one of them is a blank line. Like this:

```
c                   dou       recbuf = *blanks
C                   eval      rc = rdline(sock: %addr(recbuf):
c                                         %size(recbuf): *On)
c                   if        rc < 0
C                   eval      msg = %str(strerror(errno))
c                   dsply                   msg
c                   callp     close(sock)
c                   return
c                   endif
c                   enddo
```

And then, we will receive the actual data, which ends when the server disconnects us. We'll display each line like this:

```
c                   dou       rc < 0
c                   eval      rc = rdline(sock: %addr(recbuf):
c                                         %size(recbuf): *On)
c                   if        rc >= 0
c     recbuf        dsply
c                   endif
c                   enddo
```

The result should be the web page, without the extra HTTP responses at the start.

4.7. improvement #6: Sending back escape messages

This time, we're going to improve the way error messages are returned to the calling program or user.

Currently, we're sending back responses using the DSPLY op-code. Yuck. Blech. Ewww.

Instead, error messages should be sent back to the calling program using a typical AS/400 *ESCAPE message! This way, the program that called us will know that we ended abnormally, and can even monitor for a message to handle.

The most convienient way to implement this is to create a subprocedure that gives us a simple interface to the QMHSNDPGM (Send Program Message) API. The subprocedure will fill in all of the details that the API needs, except the actual message data -- we'll pass the message data as a parm.

The result is a subprocedure that looks like this:

```
P die             B
D die             PI
D   peMsg                      256A   const

D SndPgmMsg       PR                  ExtPgm('QMHSNDPM')
D   MessageID                    7A   Const
D   QualMsgF                    20A   Const
D   MsgData                    256A   Const
D   MsgDtaLen                   10I 0 Const
D   MsgType                     10A   Const
D   CallStkEnt                  10A   Const
D   CallStkCnt                  10I 0 Const
```

```
D   MessageKey                   4A
D   ErrorCode                32766A   options(*varsize)

D dsEC            DS
D  dsECBytesP            1      4I 0 INZ(256)
D  dsECBytesA            5      8I 0 INZ(0)
D  dsECMsgID             9     15
D  dsECReserv           16     16
D  dsECMsgDta           17    256

D wwMsgLen        S             10I 0
D wwTheKey        S              4A

c                   eval      wwMsgLen = %len(%trimr(peMsg))
c                   if        wwMsgLen<1
c                   return
c                   endif

c                   callp     SndPgmMsg('CPF9897': 'QCPFMSG   *LIBL':
c                               peMsg: wwMsgLen: '*ESCAPE':
c                               '*PGMBDY': 1: wwTheKey: dsEC)

c                   return
P                 E
```

Now, instead of code that looks like this:

```
c                   eval      sock = socket(AF_INET: SOCK_STREAM:
c                                           IPPROTO_IP)
c                   if        sock < 0
c                   eval      msg = %str(strerror(errno))
c                   dsply                   msg
c                   return
c                   endif
```

We'll make our error handling look more like this:

```
c                   eval      sock = socket(AF_INET: SOCK_STREAM:
c                                           IPPROTO_IP)
c                   if        sock < 0
c                   callp     die('socket(): ' + %str(strerror(errno)))
c                   return
c                   endif
```

Cute, huh? The die() procedure will cause it to send back an escape message when the socket() API fails. The format of the parms in the die() subprocedure makes it very easy to insert into our code.

Having said that, however, here's a slightly more complicated usage:

```
C                   if        connect(sock: p_connto: addrlen) < 0
c                   eval      err = errno
```

```
     c                   callp     close(sock)
     c                   callp     die('connect(): '+%str(strerror(err)))
     c                   return
     c                   endif
```

The close() API is a UNIX-type API and can return an error via errno just like the connect() API can! Therefore, we save the value of errno before calling close() just to make sure that we don't lose the value.

You should also note that the 'die' procedure will actually end the program. Unless something goes wrong with the 'die' procedure, the 'return' statement will never be executed. This means that if you have something to close, such as a socket, you need to make sure you do it before calling 'die'.

I'll now go through and replace DSPLY with die() throughout my program.

4.8. improvement #7: Displaying the data using DSPF

I am writing a tutorial on how to use stream files from RPG, but as of the time that I'm writing this page, I haven't yet finished it. Even so, though, I don't want to take away from the socket tutorial to try to teach stream files, so... I'll just assume that you either already know how stream files work, or that you've been able to find a tutorial somewhere.

Since data is being returned from the socket in a stream format, and we want to write it to our stream file in a stream format, it's very easy to write the web page data to a stream file! In fact, we won't even bother breaking the data that we receive into 'lines of text', but rather just dump it all (as-is) into the stream file.

When the DSPF command is run, it'll do the job of breaking things up into lines of text.

So, first we will open a stream file to contain the data we receive:

```
     c                   eval      fd = open('/http_tempfile.txt':
     c                                  O_WRONLY+O_TRUNC+O_CREAT+O_CODEPAGE:
     c                                  511: 437)
     c                   if        fd < 0
     c                   eval      err = errno
     c                   callp     close(sock)
     c                   callp     Die('open(): '+%str(strerror(err)))
     c                   return
     c                   endif
```

Next, we'll write whatever we receive from the socket into the stream file, without changing a thing:

```
     c                   dou       rc < 1
     c                   eval      rc = recv(sock: %addr(recbuf):
     c                                         %size(recbuf): 0)
     c                   if        rc > 0
     c                   callp     write(fd: %addr(recbuf): rc)
     c                   endif
     c                   enddo
```

After we've closed the file and the socket, we'll display the file using IBM's DSPF command:

```
c                   callp     close(fd)
c                   callp     close(sock)
c                   callp     Cmd('DSPF STMF("/http_tempfile.txt")':
c                                200)
```

Finally, we'll delete our temporary stream file using the 'unlink' API call:

```
c                   callp     unlink('/http_tempfile.txt')
```

WooHoo! We've gotten rid of all of the nasty DSPLY opcodes, now!

4.9. Our updated client program

The last several topics have added a lot of improvements to our client program. The result will be something that you could even adapt to your own needs if you wanted an AS/400 program that would fetch web documents.

Here's the new improved sample http client:

File: SOCKTUT/QRPGLESRC, Member: CLIENTEX2

```
H DFTACTGRP(*NO) ACTGRP(*NEW)
H BNDDIR('QC2LE') BNDDIR('SOCKTUT/SOCKUTIL')

D/copy socktut/qrpglesrc,socket_h
D/copy socktut/qrpglesrc,errno_h
D/copy socktut/qrpglesrc,sockutil_h

 *********************************************************
 * Definitions needed to make IFS API calls.  Note that
 * these should really be in a separate /copy file!
 *********************************************************
D O_WRONLY        C                   2
D O_CREAT         C                   8
D O_TRUNC         C                   64
D O_CODEPAGE      C                   8388608
D open            PR            10I 0 ExtProc('open')
D  filename                       *   value options(*string)
D  openflags                    10I 0 value
D  mode                         10U 0 value options(*nopass)
D  codepage                     10U 0 value options(*nopass)
D unlink          PR            10I 0 ExtProc('unlink')
D   path                          *   Value options(*string)
D write           PR            10I 0 ExtProc('write')
D  handle                       10I 0 value
D  buffer                         *   value
D  bytes                        10U 0 value
 *********************************************************
 * end of IFS API call definitions
 *********************************************************
```

```
D die             PR
D   peMsg                      256A   const

D cmd             PR                  ExtPgm('QCMDEXC')
D   command                    200A   const
D   length                      15P 5 const

D msg             S             50A
D sock            S             10I 0
D port            S              5U 0
D addrlen         S             10I 0
D ch              S              1A
D host            s             32A
D file            s             32A
D addr            s             10U 0
D p_Connto        S               *
D RC              S             10I 0
D RecBuf          S             50A
D RecLen          S             10I 0
D err             S             10I 0
D fd              S             10I 0

C*************************************************
C* The user will supply a hostname and file
C*  name as parameters to our program...
C*************************************************
c     *entry        plist
c                   parm                    host
c                   parm                    file

c                   eval      *inlr = *on

C*************************************************
C* what port is the http service located on?
C*************************************************
c                   eval      p_servent = getservbyname('http':'tcp')
c                   if        p_servent = *NULL
c                   callp     die('Can''t find the HTTP service!')
c                   return
c                   endif

c                   eval      port = s_port

C*************************************************
C* Get the 32-bit network IP address for the host
C*  that was supplied by the user:
C*************************************************
c                   eval      addr = inet_addr(%trim(host))
c                   if        addr = INADDR_NONE
c                   eval      p_hostent = gethostbyname(%trim(host))
c                   if        p_hostent = *NULL
c                   callp     die('Unable to find that host!')
c                   return
```

```
c                   endif
c                   eval      addr = h_addr
c                   endif

C************************************************
C* Create a socket
C************************************************
c                   eval      sock = socket(AF_INET: SOCK_STREAM:
c                                           IPPROTO_IP)
c                   if        sock < 0
c                   callp     die('socket(): ' + %str(strerror(errno)))
c                   return
c                   endif

C************************************************
C* Create a socket address structure that
C*   describes the host & port we wanted to
C*   connect to
C************************************************
c                   eval      addrlen = %size(sockaddr)
c                   alloc     addrlen       p_connto

c                   eval      p_sockaddr = p_connto
c                   eval      sin_family = AF_INET
c                   eval      sin_addr = addr
c                   eval      sin_port = port
c                   eval      sin_zero = *ALLx'00'

C************************************************
C* Connect to the requested host
C************************************************
C                   if        connect(sock: p_connto: addrlen) < 0
c                   eval      err = errno
c                   callp     close(sock)
c                   callp     die('connect(): '+%str(strerror(err)))
c                   return
c                   endif

C************************************************
C* Send a request for the file that we'd like
C* the http server to send us.
C*
C* Then we send a blank line to tell it we're
C* done sending requests, it can process them...
C************************************************
c                   callp     WrLine(sock: 'GET http://' +
c                               %trim(host) + %trim(file) +
c                               ' HTTP/1.0')
c                   callp     WrLine(sock: ' ')

C************************************************
C* Get back the server's response codes
C*
```

```
C* The HTTP server will send it's responses one
C* by one, then send a blank line to separate
C* the server responses from the actual data.
C************************************************
c                   dou       recbuf = *blanks
C                   eval      rc = rdline(sock: %addr(recbuf):
c                                         %size(recbuf): *On)
c                   if        rc < 0
c                   eval      err = errno
c                   callp     close(sock)
c                   callp     die('rdline(): '+%str(strerror(err)))
c                   return
c                   endif
c                   enddo

C************************************************
C* Open a temporary stream file to put our
C*   web page data into:
C************************************************
c                   eval      fd = open('/http_tempfile.txt':
c                                  O_WRONLY+O_TRUNC+O_CREAT+O_CODEPAGE:
c                                  511: 437)
c                   if        fd < 0
c                   eval      err = errno
c                   callp     close(sock)
c                   callp     Die('open(): '+%str(strerror(err)))
c                   return
c                   endif

C************************************************
C* Write returned data to the stream file:
C************************************************
c                   dou       rc < 1
c                   eval      rc = recv(sock: %addr(recbuf):
c                                         %size(recbuf): 0)
c                   if        rc > 0
c                   callp     write(fd: %addr(recbuf): rc)
c                   endif
c                   enddo

C************************************************
C*  We're done receiving, do the following:
C*       1) close the stream file & socket.
C*       2) display the stream file
C*       3) unlink (delete) the stream file
C*       4) end program
C************************************************
c                   callp     close(fd)
c                   callp     close(sock)
c                   callp     Cmd('DSPF STMF("/http_tempfile.txt")':
c                                200)
c                   callp     unlink('/http_tempfile.txt')
```

```
c                   return


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This ends this program abnormally, and sends back an escape.
 *   message explaining the failure.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P die             B
D die             PI
D   peMsg                      256A   const

D SndPgmMsg       PR                  ExtPgm('QMHSNDPM')
D   MessageID                    7A   Const
D   QualMsgF                    20A   Const
D   MsgData                    256A   Const
D   MsgDtaLen                   10I 0 Const
D   MsgType                     10A   Const
D   CallStkEnt                  10A   Const
D   CallStkCnt                  10I 0 Const
D   MessageKey                   4A
D   ErrorCode                32766A   options(*varsize)

D dsEC            DS
D  dsECBytesP             1      4I 0 INZ(256)
D  dsECBytesA             5      8I 0 INZ(0)
D  dsECMsgID              9     15
D  dsECReserv            16     16
D  dsECMsgDta            17    256

D wwMsgLen        S             10I 0
D wwTheKey        S              4A

c                   eval      wwMsgLen = %len(%trimr(peMsg))
c                   if        wwMsgLen<1
c                   return
c                   endif

c                   callp     SndPgmMsg('CPF9897': 'QCPFMSG   *LIBL':
c                               peMsg: wwMsgLen: '*ESCAPE':
c                               '*PGMBDY': 1: wwTheKey: dsEC)

c                   return
P                 E

 /define ERRNO_LOAD_PROCEDURE
 /copy socktut/qrpglesrc,errno_h
```

Chapter 5. Creating server programs

Written by Scott Klement.

5.1. Introduction to server programs

The last few chapters have explained how to write a program that acts as a client to an existing server program. Now, we will begin exploring the other side of the connection, the server-side.

Usually the role of a client program is to initiate the network connection, to request services from the server, and then to terminate the connection. Most of the time, a client program becomes active when it is run by a user, and is deactivated when it is done processing the data it received from the server, or when the user tells it to deactivate.

In contrast, a server program usually stays active at all times. It does not initiate any activity on its own, but rather waits for connections from client programs. When a client program has connected, it waits for the client program to make requests. When the server program realizes that the client program is done, it waits for the next connection.

There are minor differences in which API calls you need to make when writing a server program versus writing a client program. You still need to call socket() to create a socket to use for communications, but instead of issuing a connect() call, you will listen() for connections, and then accept() the connections as they come in.

It's also important to understand that a client program has to know where the server program is listening for connections. (pause to let that sink in) In other words, when a client issues a connect(), it has to tell the API which address & port to connect to -- consequently, we have to make sure that the server is, in fact, listening on the port & address that the client expects it to be listening on. This is called 'binding' to a port.

In addition to the differences in which API is called, there are a few other considerations that make server programs different from client programs:

- Server programs usually have to be capable of handling many client connections at once. Generally speaking, this isn't true of clients. This creates some significant hurdles to overcome when trying to write code.
- Because a server is available for connections at all times, the security of a server-side program is usually a much bigger concern. You have to be security conscious when writing a server program.
- For the same reasons, server programs frequently need to validate a user-id and password of the person connecting, and be careful to only give them access to what the security officer has deemed that a user should be able to access.
- Server programs usually aren't interactive applications. They have no screen associated with them, besides the one that might be found on the client side. Therefore, they tend to be a little trickier to debug.

The basic model for a server program looks something like this:

1. Call the getservbyname() API to find out the port number for the service you want to be a server for.
2. Call the socket() API to create a new socket.
3. Call the bind() API to bind the socket to the port that we found in step #1.
4. Call the listen() API to tell the system that you want to listen for connections with this socket. (This "opens up" the port so that people can connect to it)
5. Call the accept() API. The accept() API will wait until a client connects to the port, and then will create a new socket. The new socket will already be connected to the client.

6. Here we process the newly connected client. Depending on whether we're only handling one connection at a time, or what model we're using to handle many connections, we'll do things very differently at this point.
7. Close the socket you got from accept()
8. Go back to step 5.

5.2. The bind() API call

Now that we've discussed the basic operation of a server program, let's start getting into specifics. You already know how to call getservbyname(), and socket(), so the first thing we need to know is bind().

The IBM manual page for the bind() API call is located here:
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/bind.htm

It tells us that the C language prototype for bind looks like this:

```
int bind(int socket_descriptor,
                struct sockaddr *local_address,
                int address_length);
```

The procedure is called 'bind'. It takes 3 parameters, an integer, a pointer to a sockaddr structure, and another integer. The API also returns an integer.

So, we'll add the following prototype to our SOCKET_H /copy member:

```
D bind            PR            10I 0 ExtProc('bind')
D   socket                      10I 0 Value
D   local_addr                    *   Value
D   addresslen                  10I 0 Value
```

In fact, bind() and connect() have the same exact prototype, (other than the procedure name.) So what's the difference between them?

Short answer: bind() specifies the address & port on the *local* side of the connection. connect() specifies the address & port of the *remote* side of the connection.

Long answer:

Each time an IP datagram containing TCP data is sent over the network, the datagram contains a 'local address', 'remote address', 'local port', and 'remote port'. This is the only information that IP has to figure out who ends up getting the packet.

So, both the client and the server port numbers need to be filled in before the connection can work. Data that is directed to the server needs a 'destination' port, so that the data can get sent to the appropriate program running on the server. Likewise, it needs a 'source' so that the server knows who to send data back to, and also so that if there are many connections from the same computer, the server can keep them separate by looking at the source port number.

Since the connection is initiated by the client program, the client program needs to know the server's port number before it can make a connection. For this reason, servers are placed on 'well-known' port numbers. For example, a telnet server is always on port 23. A http server is always on port 80.

The bind() API call assigns the ’local’ port number. That is, the port number that is used as the ’source port’ on outgoing datagrams, and the ’destination port’ on incoming datagrams. The connect() API call assigns the ’remote’ port number, which is the one that is the ’destination port’ on outgoing packets and the ’source port’ on incoming packets.

Okay, back to bind():

If you don’t call bind(), the operating system will automatically assign you an available port number. In the case of our client programs, we didn’t call the bind() API, and the operating system assigned us whatever port was available. It didn’t matter which one, since we were the client and we were initiating the connection. The server would know our port number because we were sending it to the server in every IP datagram that we sent.

For a server, however, we must use a ’well-known’ port number. Without it, no clients will ever find us! Therefore, a server program will invariably call the bind() API.

There are some named constants that we will want to use with bind(), so we’ll also want to define those in our SOCKET_H /copy member. They look like this:

```
D INADDR_ANY      C                   CONST(0)
D INADDR_BROADCAST...
D                 C                   CONST(4294967295)
D INADDR_LOOPBACK...
D                 C                   CONST(2130706433)
```

These constants are special values that can be assigned to the sin_addr parameter of the sockaddr_in structure. (You’ll recall that sockaddr_in is to be used in place of the sockaddr structure when doing TCP/IP related socket calls.)

The value "INADDR_ANY" means that we will bind to any/all IP addresses that the local computer currently has. Lets say that your AS/400 is dialed into the Internet using a modem and the PPP protocol. The PPP interface is given an IP address by your ISP. If you also have a network card, and a TCP/IP connection to your LAN, you’ll have an address for that interface as well. Also, the special TCP/IP interface called ’loopback’ has it’s own special address (127.0.0.1). If you use the value ’INADDR_ANY’ when calling bind(), you will be able to accept connections from people connecting to any of these addresses.

On the other hand, if you only want to allow people on your LAN to connect, you could bind specifically to the network card’s IP address. Likewise, if you only wanted to allow connections from programs on the same computer, you could specify ’INADDR_LOOPBACK’.

Whew... after all that explanation, calling bind() should be easy. :)

```
D/copy socktut/qrpglesrc,socket_h

D bindto          S               *
D addrlen         S             10I 0

 ** reserve memory for a sockaddr_in structure:
c                   eval      addrlen = %size(sockaddr_in)
c                   alloc     addrlen       bindto

 ** set sockaddr_in structure so that we can receive connections
 **   on port number "port_number" on any address.
c                   eval      p_sockaddr = bindto
c                   eval      sin_family = AF_INET
c                   eval      sin_addr = INADDR_ANY
```

```
c                   eval      sin_port = port_number
c                   eval      sin_zero = *ALLx'00'

 ** bind the socket!
c                   if        bind(socket: bindto: addrlen) < 0
C* bind() failed.  check errno
c                   endif
```

5.3. The listen() API call

Once we have a socket, and we've called bind() so that it's bound to a given port and address, we have to tell the system that we want to listen for connections.

The listen() API call is documented in IBM's manual at this location:
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/listen.htm

The manual tells us that the C language prototype for the listen() API looks like this:

```
         int listen(int socket_descriptor,
                    int back_log);
```

This is an easy one. The RPG data type for an 'int' is 10I 0. This prototype accepts two integers, and returns an integer. Easy. Looks like this:

```
D listen          PR            10I 0 ExtProc('listen')
D   socket_desc                 10I 0 Value
D   back_log                    10I 0 Value
```

All the listen() API does is tell the system that we're willing to accept incoming connections. In other words, it turns this socket into a 'server socket'.

The 'back_log' parameter tells the system how many clients will be queued up to wait for our server program. You can think of this as being similar to a print queue... In a print queue, each time a new report is created, it is placed into the queue. When the printer is ready to print, it takes the first report from the queue and prints it.

The back_log works the same way. When a client calls connect(), it gets put into our queue. If there are more than 'back_log' connections in our queue, the system sends back a 'Connection Refused' message to the client.

When we're ready to talk to the client, we call the accept() API, which takes the first connection off of the queue.

The listen API is quite simple to call:

```
c                   if        listen(socket: 100) < 0
c** listen() failed, check errno!
c                   endif
```

5.4. The accept() API call

I like the listen() API, don't you? However, it doesn't do us much good to queue up clients that have connected to us, if we don't ever take them off of the queue!

That's exactly what the accept() API does. It takes a new connection off of the queue and creates a new socket which will be used for talking to the client on the other end.

The IBM manual page for the accept() API is found here:
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/accept.htm

And it tells us that the C language prototype for accept() looks like this:

```
int accept(int socket_descriptor,
           struct sockaddr *address,
           int *address_length);
```

Is this format starting to look familiar yet? The accept() API has the same parameters as the connect() and bind() APIs do, with one exception. For the 'address_length' parameter, we pass a pointer to an integer, instead of passing the value of the integer.

This means that it accepts 3 parameters. An integer, a pointer and another pointer. And, it returns an integer.

You might recall that when you pass something by reference, you're not actually passing the data itself, but rather you're passing the address of that data. Since pointers are the variables that are used to store addresses, passing the address of a variable is exactly the same thing as passing the value of a pointer. So, instead of passing the third parameter as a pointer, we can safely pass an integer by reference.

The accept() prototype will, therefore, look like this:

```
D accept          PR            10I 0 ExtProc('accept')
D   sock_desc                   10I 0 Value
D   address                       *   Value
D   address_len                 10I 0
```

Passing an integer by reference has two advantages:

- It's less typing to code "accept(socket: connfrom: len)" than it is to code "accept(socket: connfrom: %addr(len))"
- The compiler can do better syntax checking, because it knows that only integers are allowed as the 2nd parameter.

So, as was mentioned earlier... the listen() API tells the system that we're willing to accept connections. When someone connects, the system puts them in a queue. The accept() API takes the connections off of the queue in First-In-First-Out (FIFO) order.

Each time you accept a new connection, the IP address and port number of the connecting client is placed in a sockaddr_in structure that is pointed to by the 'address' parameter.

The 'address_len' parameter has two purposes. On input to the accept() API, it contains the amount of memory that you've allocated to the 'address' parameter. Accept() uses this to ensure that it doesn't write data beyond what you've allocated. On output from the accept() API, it contains the actual number of bytes that the accept() API wrote into the area of memory that you supplied as 'address'.

It's important to understand that accept() creates a new socket. The original socket which you used with bind() and listen() will continue to listen for new connections and put them into the backlog queue.

The new socket created by accept() contains your TCP connection to the client program. Once you've accept()-ed it, you can use the send() and recv() APIs to hold a conversation with the client program.

When you're done talking to the client, you'll want to call close() for the descriptor returned by accept(). Again, remember that this is a separate socket from the one that's listen()-ing. So, when you close() the socket that was returned from accept(), your program will still be listening for more connections.

To stop listening for connections, you need to call close() for the original socket.

Here's how you call accept:

```
D connfrom        S                *
D len             S              10I 0

c                   eval      len = %size(sockaddr_in)
c                   alloc     len           connfrom

c                   eval      newsock = accept(sock: connfrom: len)
c                   if        newsock < 0
C* accept() failed.  Check errno
c                   endif
c                   if        len <> 16
C* illegal length for a TCP connection!
c                   endif

c                   eval      p_sockaddr = connfrom
c                   eval      msg = 'Received a connection from ' +
c                                  %str(inet_ntoa(sin_addr)) + '!'
```

5.5. Our first server program

Now that we've been familiarized with the API calls that we need to do a simple server program, lets give one a try.

To keep our first example from getting too complicated, we're only going to handle one simultanous client. This program is intended to be very simple, rather than being practical.

Here's some pseudocode that explains the basic idea behind this example:

1. Call socket() to make a socket which will listen for connections.
2. Call bind() to bind our socket to port number 4000.
3. Call listen() to indicate that we want to accept incoming connections.
4. Call accept() to create a new socket containing the next connection.
5. Reject anyone who doesn't send a valid address.
6. Send back the IP address of the connected socket, to demonstrate how to use the 'address' parameter of the accept() API.

7. Ask the client to send a name
8. Say ’hello <NAME>’
9. Say ’goodbye <NAME>’
10. After a second, disconnect the client
11. Go back to step 4 to get the next client.

> **Note:** We will continue to use the socket utilities, header files and techniques that we used in the chapters on client socket programs.

Here’s the code:

File: SOCKTUT/QRPGLESRC, Member: SERVEREX1

```
     H DFTACTGRP(*NO) ACTGRP(*NEW)
     H BNDDIR('SOCKTUT/SOCKUTIL') BNDDIR('QC2LE')

     D/copy socktut/qrpglesrc,socket_h
     D/copy socktut/qrpglesrc,errno_h
     D/copy socktut/qrpglesrc,sockutil_h

     D Cmd             PR                  ExtPgm('QCMDEXC')
     D   command                    200A   const
     D   length                      15P 5 const

     D die             PR
     D   peMsg                      256A   const

     D len             S             10I 0
     D bindto          S               *
     D connfrom        S               *
     D port            S              5U 0
     D lsock           S             10I 0
     D csock           S             10I 0
     D line            S             80A
     D err             S             10I 0
     D clientip        S             17A

     c                   eval      *inlr = *on

     c                   exsr      MakeListener

     c                   dow       1 = 1
     c                   exsr      AcceptConn
     c                   exsr      TalkToClient
     c                   callp     close(csock)
     c                   enddo
```

```
C*===============================================================
C* This subroutine sets up a socket to listen for connections
C*===============================================================
CSR   MakeListener  begsr
C*------------------------
C** Normally, you'd look the port up in the service table with
C**  the getservbyname command.   However, since this is a 'test'
C**  protocol -- not an internet standard -- we'll just pick a
C**  port number.   Port 4000 is often used for MUDs... should be
C**  free...
c                   eval      port = 4000

C* Allocate some space for some socket addresses
c                   eval      len = %size(sockaddr_in)
c                   alloc     len           bindto
c                   alloc     len           connfrom

C* make a new socket
c                   eval      lsock = socket(AF_INET: SOCK_STREAM:
c                                        IPPROTO_IP)
c                   if        lsock < 0
c                   callp     die('socket(): ' + %str(strerror(errno)))
c                   return
c                   endif

C* bind the socket to port 4000, of any IP address
c                   eval      p_sockaddr = bindto
c                   eval      sin_family = AF_INET
c                   eval      sin_addr = INADDR_ANY
c                   eval      sin_port = port
c                   eval      sin_zero = *ALLx'00'

c                   if        bind(lsock: bindto: len) < 0
c                   eval      err = errno
c                   callp     close(lsock)
c                   callp     die('bind(): ' + %str(strerror(err)))
c                   return
c                   endif

C* Indicate that we want to listen for connections
c                   if        listen(lsock: 5) < 0
c                   eval      err = errno
c                   callp     close(lsock)
c                   callp     die('listen(): ' + %str(strerror(err)))
c                   return
c                   endif
C*------------------------
CSR                 endsr


C*===============================================================
C* This subroutine accepts a new socket connection
C*===============================================================
```

```
CSR   AcceptConn    begsr
C*------------------------
c                   dou       len = %size(sockaddr_in)

C* Accept the next connection.
c                   eval      len = %size(sockaddr_in)
c                   eval      csock = accept(lsock: connfrom: len)
c                   if        csock < 0
c                   eval      err = errno
c                   callp     close(lsock)
c                   callp     die('accept(): ' + %str(strerror(err)))
c                   return
c                   endif

C* If socket length is not 16, then the client isn't sending the
C*  same address family as we are using...  that scares me, so
C*  we'll kick that guy off.
c                   if        len <> %size(sockaddr_in)
c                   callp     close(csock)
c                   endif

c                   enddo

c                   eval      p_sockaddr = connfrom
c                   eval      clientip = %str(inet_ntoa(sin_addr))
C*------------------------
CSR                 endsr


C*===============================================================
C*  This does a quick little conversation with the connecting
c*  client.   That oughta teach em.
C*===============================================================
CSR   TalkToClient  begsr
C*------------------------
c                   callp      WrLine(csock: 'Connection from ' +
c                                   %trim(clientip))

c                   callp      WrLine(csock: 'Please enter your name' +
c                                   ' now!')

c                   if         RdLine(csock: %addr(line): 80: *On) < 0
c                   leavesr
c                   endif

c                   callp      WrLine(csock: 'Hello ' + %trim(line))
c                   callp      WrLine(csock: 'Goodbye ' + %trim(line))

c                   callp     Cmd('DLYJOB DLY(1)': 200)
C*------------------------
CSR                 endsr
```

```
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This ends this program abnormally, and sends back an escape.
 *   message explaining the failure.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P die             B
D die             PI
D   peMsg                      256A   const

D SndPgmMsg       PR                  ExtPgm('QMHSNDPM')
D   MessageID                    7A   Const
D   QualMsgF                    20A   Const
D   MsgData                    256A   Const
D   MsgDtaLen                   10I 0 Const
D   MsgType                     10A   Const
D   CallStkEnt                  10A   Const
D   CallStkCnt                  10I 0 Const
D   MessageKey                   4A
D   ErrorCode                32766A   options(*varsize)

D dsEC            DS
D  dsECBytesP             1      4I 0 INZ(256)
D  dsECBytesA             5      8I 0 INZ(0)
D  dsECMsgID              9     15
D  dsECReserv            16     16
D  dsECMsgDta            17    256

D wwMsgLen        S             10I 0
D wwTheKey        S              4A

c                   eval      wwMsgLen = %len(%trimr(peMsg))
c                   if        wwMsgLen<1
c                   return
c                   endif

c                   callp     SndPgmMsg('CPF9897': 'QCPFMSG   *LIBL':
c                               peMsg: wwMsgLen: '*ESCAPE':
c                               '*PGMBDY': 1: wwTheKey: dsEC)

c                   return
P                 E

 /define ERRNO_LOAD_PROCEDURE
 /copy socktut/qrpglesrc,errno_h
```

5.6. Testing server programs

Usually server programs are not interactive programs. There is no screen to output details to, and no user to read those details. So how do we test them, how do we know if they work?

Since we're usually working with plain ASCII, line-oriented data, we can use a telnet client to debug our server code.

With a telnet client, we'll type in the code that we expect the client program to send, and we'll see the exact responses that the server gives us. This is an invaluable tool for debugging sockets programs :)

Unfortunately (at least as of V4R5) the AS/400's TELNET command doesn't work nicely for debugging sockets programs. However, the Microsoft Windows and Unix (Linux, FreeBSD, etc) telnet clients work just fine.

So, let's test the server program:

1. If you haven't done so already, compile the server program now.
2. Run the server by typing: CALL SERVEREX1 (or whatever name you compiled it as)
3. From Microsoft Windows, Click "Start", then "Run" then type: `telnet as400 4000`

   or, from a Linux/BSD/Unix shell, type: `telnet as400 4000`
4. When it says 'Please enter your name now!', type your name.

   Note: Depending on your telnet client, you may not be able to see what you type -- this is normal

5. It should respond with "hello" and "goodbye" and then disconnect.

this same technique can be used to debug just about any server program. For example, if you wanted to test our my web server, you might type:

1. telnet www.scottklement.com 80
2. Once connected, type: **`GET http://www.scottklement.com/rpg/socktut/ip_servertesting.txt HTTP/1.0`**
3. press enter twice.

It's always interesting just how much of the Internet can be experienced with a simple telnet client

This first example server program doesn't know how to end. It'll just keep running forever, locking out port 4000. Sure, you could use system request to end the program, or you could do a WRKACTJOB and then 'End Job', but if you do that, OS/400 won't ever take port 4000 out of listen mode!

Therefore, the easiest way to end this program is by using the `NETSTAT` command.

1. Sign on to your AS/400 from a different session and type: `NETSTAT *CNN`
2. It will show you a list of TCP/IP ports being listened on, as well as showing anyone who is connected, and which remote and local ports they are connected with.
3. If you have sufficient authority, you can place a '4' next to a given connection to end that connection.
4. To end our server program, find the line that says that it is in 'Listen' state on port 4000. Put a '4' next to that line, and end that connection.

5. The server program will end, it will say that it got an error: 'accept(): The protocol required to support the specified address family is not available at this time.' This is because OS/400 ends the connection by the protocol unavailable to the program for that particular socket.

5.7. Making our program end

The last topic showed us how to try out our server program. It also showed us that it's a bit of a pain to make our server program end.

So, why don't we make a quick improvement to our server program? We'll set it up so that if the person's name is 'quit', the program shuts itself down. This should be easy to do. It requires some code to be added in two places in the program:

In the "TalkToClient" subroutine, after the 'RdLine' group of code, we'll add a simple if, like this:

```
c                   if        line = 'quit'
c                   leavesr
c                   endif
```

And in the mainline of the program, if the user's name was 'quit', we need to close the socket that listens for connections, and end the program.

Right after the line that says 'callp close(csock)', we add this:

```
c                   if        line = 'quit'
c                   callp     close(lsock)
c                   return
c                   endif
```

Now, re-compile and run this program again. Test it with telnet. When the server program gets a user whos name is NOT quit, it should work as it did before. When the user's name *is* quit, it should end.

There is a problem with this, however. After entering a user-id of 'quit', the server program ends as it should. But if you immediately run it again, you get this error:

`bind(): Address already in use.`

But... it's not in use... Take a look at NETSTAT! In fact, if you wait two or three minutes before running your program again, it works just fine.

This is one of the most common "gotchas" in socket programming. The same port cannot be opened by two different programs, without taking special actions to make it work. Even though the socket from the first run of our sample program has already been closed, the system still waits on that socket for a period of time, before allowing you to re-use it.

The reason it waits has to do with how the TCP protocol works. TCP is a 'reliable' protocol, it ensures that anything sent by one side of the connection gets received by the other side of the connection. It sends the data in chunks called 'segments'. Each time a segment is received, the receiving side sends back an 'ACK' (acknowledgement) datagram, so that the sending side knows that the data was received. If it never gets an ACK, it re-sends the data.

When a connection is closed, each side sends a ’FIN’ (finished) datagram to the other. Once both sides have received a ’FIN’, they know that the connection is closed.

So, like everything else in TCP, after a FIN is received, the side that received it sends back an ’ACK’. But how does it know that the ACK was received? You can’t acknowledge an ’ACK’, that would create an endless chain of ACKs going back and forth.

So how does the side sending back the ’final ACK’ know that the ACK has been received?

The answer is... it doesn’t. But it needs to wait a period of time, just in case the ACK got lost, so that it can re-send the ACK in the event that it never got received. (It will know that the ACK never got received when the FIN datagram is re-sent) RFC 793, which is the standard for how TCP is to work, tells us that the socket which sends the ’final ACK’ is to wait for twice the Maximum Segment Lifetime (MSL) before closing completely.

Getting back to the original question, each time you close a socket, it must wait until the (MSL x 2) has passed before it can completely close the socket. During the time that it’s waiting, the AS/400 will show that socket as being in ’TIME-WAIT’ state. Since the socket that’s in TIME-WAIT state is still using our port! And, by default, two sockets can’t have the same port open! And THAT is why you receive the "Address already in use" error.

Fortunately, there is an easy fix. Each socket has many different options that can be set using the setsockopt() API. One of those options is called "SO_REUSEADDR", and it allows a socket to share the same address and port as an existing socket. If we turn this option on, we will be able to re-bind to our port immediately.

Many people have asked me, "isn’t it dangerous to allow more than one socket to bind to the same port? What if two different programs tried to use the same port at the same time?"

It’s not as dangerous as you might think. Although it’s possible for two programs to bind to the same port this way, it’s still NOT possible for two programs to LISTEN on the same port. Therefore, if you had two server programs trying to bind to the same port, the second one would still get an error, because the first one was already listening.

5.8. The setsockopt() API call

As I explained in the previous topic, there is an API that can be used to set or change certain attributes about a socket, called ’setsockopt’. Here we will detail using that API call in RPG.

The IBM manual page for the setsockopt() API is found here:
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/ssocko.htm

The ever-helpful IBM manual refers to setsockopt() as "Set Socket Options" and proceeds to tell us "The setsockopt() function is used to set socket options". Gee, thanks.

The C language prototype for setsockopt() looks like this:

```
int setsockopt(int socket_descriptor,
               int level,
               int option_name,
               char *option_value,
               int option_length);
```

By now, you’re probably an old-pro at converting these prototypes to RPG. The C prototype tells us that the setsockopt() procedure receives 5 parameters. These parameters are an integer, an integer, another integer, a pointer, and yet another integer. It also returns an integer.

The ’char *option_value’ parameter does require a bit more research. We know that it’s a pointer to a character variable. But is it supposed to be a null-terminated string of characters? or is it supposed to be a single byte? or what? So, we page down to where it describes the option_value, we find that it says "A pointer to the option value. Integer flags/values are required by setsockopt() for all the socket options except SO_LONGER, IP_OPTIONS, . . ."

Oh great. So, it tells us it’s a pointer to a character... but it really wants a pointer to an integer most of the time, and a pointer to other things at other times.

Fortunately, this doesn’t present a big problem for us. In RPG, pointers can point to any data type. So really all we needed to learn from this is that we do not want to make it a pointer to a null terminated string.

Here’s our RPG prototype. Hope it provides hours of enjoyment:

```
D setsockopt      PR            10I 0 ExtProc('setsockopt')
D   socket_desc                 10I 0 Value
D   level                       10I 0 Value
D   option_name                 10I 0 Value
D   option_value                  *   Value
D   option_length               10I 0 Value
```

Please add that prototype to your SOCKET_H member.

Now, we’ll need to define the constants for the ’level’ and ’option_name’ parameters, but first, a bit of explanation. TCP/IP is a "layered protocol". At the start of this tutorial, I explained a bit how this works, there’s the Internet Protocol (IP) that handles the transportation of data across an inter-network. Running "on top of" that is (in this case) the Transmission Control Protocol (TCP), which uses the IP protocol to send and receive data, but adds the extra functionality of operating like a reliable-stream of data. On top of that is the Application-Level Protocol that varies from program to program. We send and receive this application data by calling the send() and recv() options on a socket.

So, when setting the options that control how a socket works, we may be setting them at different levels. Some options are set at the IP level, others at the TCP level, and others may pertain to the socket itself, so we have a special "socket level" to deal with.

For example, there is a "Time To Live" (TTL) value associated with every packet sent over the internet. This TTL defines the maximum number of routers or gateways the packet can travel through before being dropped.

This is useful because sometimes people make mistakes and misconfigure a router. If packets didnt have a maxmimum time to live, it would be possible for packets in error to bounce around the internet forever. Since this ’TTL’ value relates to the routing of datagrams, it must be a part of the IP layer, therefore to change it with setsockopt, you set the ’level’ parameter to the constant that means "IP layer" and set the option_name parameter to the constant that means "TTL".

Here are the values for the different "levels". Note that some of these are already defined in your SOCKET_H member, because they’re used by the socket() API as well. You should add the other ones, though:

```
D*                                                Internet Protocol
D IPPROTO_IP      C                   CONST(0)
D*                                                Transmission Control
D*                                                Protocol
D IPPROTO_TCP     C                   CONST(6)
D*                                                Unordered Datagram
D*                                                Protocol
D IPPROTO_UDP     C                   CONST(17)
```

```
D*                                          Raw Packets
D IPPROTO_RAW     C                   CONST(255)
D*                                          Internet Control
D*                                          Msg Protocol
D IPPROTO_ICMP    C                   CONST(1)
D*                                          socket layer
D SOL_SOCKET      C                   CONST(-1)
```

Here are the values for the "option_name" parameter that are used at the "IP level":

```
D*                                            ip options
D IP_OPTIONS      C                   CONST(5)
D*                                            type of service
D IP_TOS          C                   CONST(10)
D*                                            time to live
D IP_TTL          C                   CONST(15)
D*                                            recv lcl ifc addr
D IP_RECVLCLIFADDR...
D                 C                   CONST(99)
```

Here are the values for the "option_name" parameter that are used at the "TCP level":

```
D*                                      max segment size (MSS)
D TCP_MAXSEG      C                   5
D*                                      dont delay small packets
D TCP_NODELAY     C                   10
```

Here are the values for the "option_name" parameter that are used at the "Socket Level":

```
D*                                      allow broadcast msgs
D SO_BROADCAST    C                   5
D*                                      record debug informatio
D SO_DEBUG        C                   10
D*                                      just use interfaces,
D*                                      bypass routing
D SO_DONTROUTE    C                   15
D*                                      error status
D SO_ERROR        C                   20
D*                                      keep connections alive
D SO_KEEPALIVE    C                   25
D*                                      linger upon close
D SO_LINGER       C                   30
D*                                      out-of-band data inline
D SO_OOBINLINE    C                   35
D*                                      receive buffer size
D SO_RCVBUF       C                   40
D*                                      receive low water mark
D SO_RCVLOWAT     C                   45
D*                                      receive timeout value
D SO_RCVTIMEO     C                   50
D*                                      re-use local address
```

```
D SO_REUSEADDR    C                   55
D*                                          send buffer size
D SO_SNDBUF       C                   60
D*                                          send low water mark
D SO_SNDLOWAT     C                   65
D*                                          send timeout value
D SO_SNDTIMEO     C                   70
D*                                          socket type
D SO_TYPE         C                   75
D*                                          send loopback
D SO_USELOOPBACK  C                   80
```

In addition to all of those options that you can set, there is one option that accepts a data structure as a parameter. That’s the ’SO_LINGER’ option. So, to be complete, we’ll also add the linger structure to our SOCKET_H header member:

```
D p_linger        S               *
D linger          DS                  BASED(p_linger)
D   l_onoff                     10I 0
D   l_linger                    10I 0
```

If you’re interested in seeing the C definitions for all of these items that we just added, you can find them in the following members in your QSYSINC ("System Openness Includes") library:

```
File: QSYSINC/SYS     Member: SOCKET
File: QSYSINC/NETINET Member: IN
File: QSYSINC/NETINET Member: TCP
```

Obviously, due to the large number of options that can be set using the setsockopt() API, the method of calling it will vary quite a bit. However, here are 3 different examples:

```
D value           S             10I 0
D len             S             10I 0
D ling            S               *


 *** Change datagram TTL to 16 hops:
 ***
c                   eval      value = 16
c                   if        setsockopt(s: IPPROTO_IP: IP_TTL:
c                                %addr(value): %size(value)) < 0
C* setsockopt() failed, check errno
c                   endif


 *** Allow re-use of port in bind():
 ***   (note that 1 = on, so we’re turning the ’re-use address’
 **      capability on)
c                   eval      value = 1
c                   if        setsockopt(s: SOL_SOCKET: SO_REUSEADDR:
```

```
c                                %addr(value): %size(value)) < 0
C* setsockopt() failed, check errno
c                   endif



 ** Make space for a linger structure:
c                   eval      len = %size(linger)
c                   alloc     len           ling
c                   eval      p_linger = ling

 *** tell the system to discard buffered data 1 minute
 ***  after the socket is closed:
c                   eval      l_onoff = 1
c                   eval      l_linger = 60

c                   if        setsockopt(s: SOL_SOCKET: SO_LINGER:
c                                   ling: %size(linger)) < 0
C* setsockopt() failed, check errno
c                   endif
```

5.9. The revised server program

The vast majority of options in the setsockopt() API are only used rarely. However, when running a server, it's a very good idea to set two of them:

- SO_REUSEADDR -- As we already discussed, if we want to use a specific port, but don't want to wait for the MSL timeout, we need to turn this option on in our server program.
- SO_LINGER -- Whenever the send() API is used to send data, it gets placed into a buffer, and is actually sent when the program on the other end of the connection is ready to recv() it. If we've sent some data, then called close() to drop the connection, what should the system do with data that's still in its send buffer? By default, it tries to send it forever. However, network errors could cause this to waste some memory indefinitely! So, for long running programs like a server, we should tell it how long to linger on that data...

Therefore, we'll insert these new options into our example server program. They'll go into the "MakeListner" subroutine, right after we've called the socket() API.

```
C* Tell socket that we want to be able to re-use port 4000
C*  without waiting for the MSL timeout:
c                   callp     setsockopt(lsock: SOL_SOCKET:
c                                SO_REUSEADDR: %addr(on): %size(on))

C* create space for a linger structure
c                   eval      linglen = %size(linger)
c                   alloc     linglen       ling
c                   eval      p_linger = ling

C* Data shouldnt need to linger on the "listener" socket, but
```

```
C*   give it one second, just in case:
c                   eval      l_onoff = 1
c                   eval      l_linger = 120
c                   callp     setsockopt(lsock: SOL_SOCKET: SO_LINGER:
c                                ling: linglen)
```

Now, remember, we have two sockets, the one we create in MakeListener, and the one that's returned by the accept() API in the AcceptConn subroutine. The SO_REUSEADDR option wouldn't make sense on the accept() socket (since we don't bind it) but we should set the SO_LINGER value!

So, insert this code, right after we call accept():

```
C* tell socket to only linger for 2 minutes, then discard:
c                   eval      l_onoff = 1
c                   eval      l_linger = 120
c                   callp     setsockopt(csock: SOL_SOCKET: SO_LINGER:
c                                ling: linglen)
```

After adding these options, we can go ahead and compile & test our program again. Notice that now, we can re-run the program immediately, after it ends, we don't have to wait 2 minutes...

Just to dispel any confusion, here's a full listing of our server program up to this point:

```
H DFTACTGRP(*NO) ACTGRP(*NEW)
H BNDDIR('SOCKTUT/SOCKUTIL') BNDDIR('QC2LE')

D/copy socktut/qrpglesrc,socket_h
D/copy socktut/qrpglesrc,errno_h
D/copy socktut/qrpglesrc,sockutil_h

D Cmd             PR                  ExtPgm('QCMDEXC')
D   command                    200A   const
D   length                      15P 5 const

D die             PR
D   peMsg                      256A   const

D len             S             10I 0
D bindto          S               *
D connfrom        S               *
D port            S              5U 0
D lsock           S             10I 0
D csock           S             10I 0
D line            S             80A
D err             S             10I 0
D clientip        S             17A
D ling            S               *
D linglen         S             10I 0
D on              S             10I 0 inz(1)

c                   eval      *inlr = *on
```

```
c                   exsr      MakeListener

c                   dow       1 = 1
c                   exsr      AcceptConn
c                   exsr      TalkToClient
c                   callp     close(csock)

c                   if        line = 'quit'
c                   callp     close(lsock)
c                   return
c                   endif

c                   enddo


C*===============================================================
C* This subroutine sets up a socket to listen for connections
C*===============================================================
CSR   MakeListener  begsr
C*------------------------
C** Normally, you'd look the port up in the service table with
C**  the getservbyname command.   However, since this is a 'test'
C**  protocol -- not an internet standard -- we'll just pick a
C**  port number.   Port 4000 is often used for MUDs... should be
C**  free...
c                   eval      port = 4000

C* Allocate some space for some socket addresses
c                   eval      len = %size(sockaddr_in)
c                   alloc     len           bindto
c                   alloc     len           connfrom

C* make a new socket
c                   eval      lsock = socket(AF_INET: SOCK_STREAM:
c                                            IPPROTO_IP)
c                   if        lsock < 0
c                   callp     die('socket(): ' + %str(strerror(errno)))
c                   return
c                   endif

C* Tell socket that we want to be able to re-use port 4000
C*  without waiting for the MSL timeout:
c                   callp     setsockopt(lsock: SOL_SOCKET:
c                                SO_REUSEADDR: %addr(on): %size(on))

C* create space for a linger structure
c                   eval      linglen = %size(linger)
c                   alloc     linglen       ling
c                   eval      p_linger = ling

C* tell socket to only linger for 2 minutes, then discard:
c                   eval      l_onoff = 1
c                   eval      l_linger = 1
```

```
c                   callp     setsockopt(lsock: SOL_SOCKET: SO_LINGER:
c                                ling: linglen)

C* bind the socket to port 4000, of any IP address
c                   eval      p_sockaddr = bindto
c                   eval      sin_family = AF_INET
c                   eval      sin_addr = INADDR_ANY
c                   eval      sin_port = port
c                   eval      sin_zero = *ALLx'00'

c                   if        bind(lsock: bindto: len) < 0
c                   eval      err = errno
c                   callp     close(lsock)
c                   callp     die('bind(): ' + %str(strerror(err)))
c                   return
c                   endif

C* Indicate that we want to listen for connections
c                   if        listen(lsock: 5) < 0
c                   eval      err = errno
c                   callp     close(lsock)
c                   callp     die('listen(): ' + %str(strerror(err)))
c                   return
c                   endif
C*------------------------
CSR                 endsr


C*===============================================================
C* This subroutine accepts a new socket connection
C*===============================================================
CSR   AcceptConn    begsr
C*------------------------
c                   dou       len = %size(sockaddr_in)

C* Accept the next connection.
c                   eval      len = %size(sockaddr_in)
c                   eval      csock = accept(lsock: connfrom: len)
c                   if        csock < 0
c                   eval      err = errno
c                   callp     close(lsock)
c                   callp     die('accept(): ' + %str(strerror(err)))
c                   return
c                   endif

C* tell socket to only linger for 2 minutes, then discard:
c                   eval      l_onoff = 1
c                   eval      l_linger = 120
c                   callp     setsockopt(csock: SOL_SOCKET: SO_LINGER:
c                                ling: linglen)

C* If socket length is not 16, then the client isn't sending the
C*  same address family as we are using...  that scares me, so
```

```
C*  we'll kick that guy off.
c                   if        len <> %size(sockaddr_in)
c                   callp     close(csock)
c                   endif

c                   enddo

c                   eval      p_sockaddr = connfrom
c                   eval      clientip = %str(inet_ntoa(sin_addr))
C*------------------------
CSR                 endsr


C*===============================================================
C*  This does a quick little conversation with the connecting
c*  client.   That oughta teach em.
C*===============================================================
CSR   TalkToClient  begsr
C*------------------------
c                   callp      WrLine(csock: 'Connection from ' +
c                                   %trim(clientip))

c                   callp      WrLine(csock: 'Please enter your name' +
c                                   ' now!')

c                   if         RdLine(csock: %addr(line): 80: *On) < 0
c                   leavesr
c                   endif

c                   if        line = 'quit'
c                   leavesr
c                   endif

c                   callp      WrLine(csock: 'Hello ' + %trim(line))
c                   callp      WrLine(csock: 'Goodbye ' + %trim(line))

c                   callp     Cmd('DLYJOB DLY(1)': 200)
C*------------------------
CSR                 endsr


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This ends this program abnormally, and sends back an escape.
 *   message explaining the failure.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P die               B
D die               PI
D   peMsg                      256A   const

D SndPgmMsg         PR                ExtPgm('QMHSNDPM')
D   MessageID                    7A   Const
D   QualMsgF                    20A   Const
D   MsgData                    256A   Const
```

```
D   MsgDtaLen                     10I 0 Const
D   MsgType                       10A   Const
D   CallStkEnt                    10A   Const
D   CallStkCnt                    10I 0 Const
D   MessageKey                     4A
D   ErrorCode                  32766A   options(*varsize)

D dsEC              DS
D  dsECBytesP               1      4I 0 INZ(256)
D  dsECBytesA               5      8I 0 INZ(0)
D  dsECMsgID                9     15
D  dsECReserv              16     16
D  dsECMsgDta              17    256

D wwMsgLen          S             10I 0
D wwTheKey          S              4A

c                     eval      wwMsgLen = %len(%trimr(peMsg))
c                     if        wwMsgLen<1
c                     return
c                     endif

c                     callp     SndPgmMsg('CPF9897': 'QCPFMSG   *LIBL':
c                                 peMsg: wwMsgLen: '*ESCAPE':
c                                 '*PGMBDY': 1: wwTheKey: dsEC)

c                     return
P                   E

 /define ERRNO_LOAD_PROCEDURE
 /copy socktut/qrpglesrc,errno_h
```

Chapter 6. Handling many sockets at once using select()

Written by Scott Klement.

6.1. Overview of handling many clients

There are three different approaches to making a server program be capable of handing many simultaneous clients. These approaches are:

- Have a single program, running as a single process, that switches between all of the connected clients.
- Have a "listener" program that listens for new connections, and then hands off each new client to a separate job.
- Use threads to allow a single, multi-threaded program to handle many clients. (However, threads are not available in RPG, at least not in V4R5)

There are pros and cons to each of these approaches, and special issues that need to be dealt with. This chapter will deal with the first approach, and the next chapter will deal with the second approach.

6.1.1. The "single program" approach:

This approach involves a single program that reads from many clients, and writes to many clients in a loop. The program needs to be careful not to "stop" on any one client's work, but to keep switching between all of the involved clients so they all appear to be working.

The pros of this approach:

- It is efficient. There is no "startup overhead" such as waiting for a new job to be submitted, and it only needs one set of resources.
- It allows for easy communication between all of the connected clients. For example, if you wanted to write a chat room where everything being said had to echoed to all of the connected clients, you'd want to use this approach, because all of the sockets are together in one program where data can be copied from one to the other.

The cons of this approach:

- It's very hard to limit each user's access according to the userid they signed in with, since all actions for all users are being taken by a single program.
- You can't really call other programs to implement functionality, since your program would have to give up control to the program that you call, and when that happened, all of the socket processing would stop.
- It's very easy for your code to become very complicated and diffiult to maintain.

Up to this point in the tutorial, all of our socket operations have used "blocking" operations. When I say "blocking", I mean that each operation doesn't return control to our program until it has completed. For example, if we call recv(), the computer will wait until there is data to recv() from the other side of the connection. It doesn't stop waiting until the other side has disconnected, or until some data actually appears.

When working with many different clients, however, this can be a problem. We need to be able to read from whichever client actually sends data to us, without being stuck waiting for one that's not sending data to us.

To solve all these problems, the select() API was created. The select API allows us to specify 3 different "groups" or "sets" of sockets. A "read" set, a "write" set and an "exceptional" set. It can tell us when there is data to be read on each of the sockets, as well as telling us when it's safe to write to each of the sockets. Select() also has a "timeout" value which lets us regain control even when there has been no activity.

Extending our server program into one based on this model will be difficult, so we'll start with something a bit simpler than that.

You'll recall that we tested our server programs by using a TELNET client to simply connect to the port we were listening on. But, what if we wanted to test a client program instead of a server program? We couldn't do that because a TELNET client only connects to servers, not to clients...

What we need is a special server program that would accept a connection from a client program as well as a telnet client. It could then copy the data sent from the client to the telnet socket, and vice-versa.

This program would be relatively easy to write, and would be a good starting example of how to deal with multiple connected clients -- as well as a useful tool -- so that's what we'll start with in this chapter.

6.2. The select() API call

The most important API to use when trying to work with multiple clients in the same instance of the same program is very definitely the select() API.

The IBM manual page for the select() API is found here:
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/sselect.htm

The manual tells us that the C language prototype for select() looks like this:

```
int select(int max_descriptor,
           fd_set *read_set,
           fd_set *write_set,
           fd_set *exception_set,
           struct timeval *wait_time);
```

This tells us that the select() API accepts 5 parameters, and they are an integer, a pointer to a 'fd_set', another pointer to an 'fd_set', another pointer to an 'fd_set' and a pointer to a 'timeval' structure.

Therefore, the RPG prototype for the select() API looks like this:

```
D select          PR            10I 0 extproc('select')
D   max_desc                    10I 0 VALUE
D   read_set                      *   VALUE
D   write_set                     *   VALUE
D   except_set                    *   VALUE
D   wait_time                     *   VALUE
```

Read the IBM page over, the information here should give you a basic idea of what exactly the select() API does.

The ’fd_set’ data type is a string of 224 bits, each representing a descriptor to check for data in. In C, the fd_set data type is represented as an array of integers. The reason for this is that it’s easy to bits on & off in an integer variable in C. In RPG, however, I prefer to work with a simple string of character variables, since that’s what the bitwise operations in RPG work with.

In C, there are 4 ’preprocessor macros’ that are included to make it easy to work with the data in a descriptor set. In order to do the same thing in RPG, we’ll have to write subprocedures of our own that mimic these macros.

I’ll describe the ’fd_set’ data type, and the subprocedures that work with it in the next topic. Suffice it to say, for now, that I use a descriptor set that’s defined as ’28A’ in RPG.

The last parameter to select() is of type ’struct timeval’. You could call it a ’time value structure’ if you like.

The ’timeval’ structure in C looks like this:

```
struct timeval {
   long  tv_sec;                 /* seconds       */
   long  tv_usec;                /* microseconds  */
 };
```

It’s nothing more than 2 32-bit integers in a structure. The first integer is the number of seconds to wait before the select() operation times out. The second integer specifies the number of microseconds (1,000,000th of a second) to wait before timing out.

So, the timeval structure in RPG would look like this:

```
D p_timeval       S               *
D timeval         DS                  based(p_timeval)
D   tv_sec                      10I 0
D   tv_usec                     10I 0
```

Make sure you add the prototype for the select() API as well as the definition of the ’timeval’ structure to your SOCKET_H header file.

You call the select() API in RPG like this:

```
D readset         S             28A
D writeset        S             28A
D excpset         S             28A
D tv_len          S             10I 0
D tv              S               *

 * create a timeval struct, set it to 10.5 seconds:

C                   eval      tv_len = %size(timeval)
C                   alloc     tv_len        tv
C                   eval      p_timeval = tv

C                   eval      tv_sec = 10
C                   eval      tv_usec = 500000

 * call select.
```

```
C                   eval      rc = select(maxsock+1: %addr(readset):
c                                    %addr(writeset): %addr(excpset):
c                                    p_timeval)
```

6.3. Utility routines for working with select()

Working with descriptor sets (the 'fd_set' data type in C) is a little tricky in RPG. We will want to add utility routines to our 'sockutil' service program to help us work with them.

As you know, the socket API returns an integer. This integer that it returns is called a socket descriptor. The first descriptor opened in a job will always return a 0. The next one will be a 1, then a 2, etc. Of course, the system itself, along with every program in the job, can open some of these -- so you really never know what number you'll get.

Remember that the select() API is able to tell you which sockets are 'readable', 'writable' or 'have a pending exception'. That means that you need a way to give select() a whole list of sockets, and it needs a way to return a whole list of results.

You do this by using a bit string. For example, if you called socket() twice, and the first time it returned a 1, and the second time it returned a 5, you'll set on bit #1, and bit #5 in the bit string. Now select will know that it needs to check to see if there's data to read on descriptors 1 and 5.

This string of bits is called an 'fd_set' (which stands for 'file descriptor set', since select() can also be used with stream files)

In C, the fd_set data type is an array of unsigned integers. Each integer is 32-bits long (that is, 4 bytes) and the array always has at least 7 integers in it. Therefore, the minimum size for a fd_set array is 28 bytes long.

The bits of each integer of the array are numbered from 0 to 31, starting with the rightmost (least significant) bit. Therefore, if we were to number the descriptors, it'd look something like this:

First integer in array represents the first 32 descriptors:

```
3     2    2    1    1
1.....5....0....5....0....5....0
```

Second integer in array represents the next 32-descriptors:

```
6  6    5    5    4    4    3  3
3..0....5....0....5....0....5..2
```

And so on... until we got to descriptor #223.

Since bitwise operations in RPG are actually easier to do on character fields than they are on integer fields, I implemented the 'fd_set' data type in RPG as a simple 28-byte long alphanumeric field.

Each bit in that field must be numbered the same way that the array of integers was for the C programs, however, since we're calling the same select() API that C programs will call.

Therefore, I wrote the following subprocedure that will calculate which will take a descriptor number, and calculate which byte in the ’28A’ field needs to be checked, and a bitmask which can be used to set the specific bit within that byte on, or off, of whatever you want to do with it.

This subprocedure looks like this:

```
P CalcBitPos      B
D CalcBitPos      PI
D    peDescr                    10I 0
D    peByteNo                    5I 0
D    peBitMask                   1A
D dsMakeMask      DS
D   dsZeroByte            1      1A
D   dsMask                2      2A
D   dsBitMult             1      2U 0 INZ(0)
C     peDescr       div       32            wkGroup           5 0
C                   mvr                     wkByteNo          2 0
C                   div       8             wkByteNo          2 0
C                   mvr                     wkBitNo           2 0
C                   eval      wkByteNo = 4 - wkByteNo
c                   eval      peByteNo = (wkGroup * 4) + wkByteNo
c                   eval      dsBitMult = 2 ** wkBitNo
c                   eval      dsZeroByte = x’00’
c                   eval      peBitMask = dsMask
P                 E
```

This routine will be called by many of the subprocedures that will write to replace the C-language ’pre-processor macros’.

The first pre-processor macro that we need to duplicate will be the ’FD_ZERO’ macro. All this does is zero out all of the bits in a ’fd_set’. To do this in RPG, using a 28A field is child’s play. We just do this:

```
P*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P* Clear All descriptors in a set.  (also initializes at start)
P*
P*      peFDSet = descriptor set
P*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P FD_ZERO         B                   EXPORT
D FD_ZERO         PI
D   peFDSet                     28A
C                   eval      peFDSet = *ALLx’00’
P                 E
```

next, we’ll need to do ’FD_SET’. First of all, remember that C is case-sensitive, so ’FD_SET’ is a different name from ’fd_set’. The FD_SET macro in C is used to turn on one bit (one descriptor) in a descriptor set.

To do this in RPG, we’ll have to call the CalcBitPos routine that we described above to find out which byte & bitmask to use, then we’ll use that bitmask to turn on the appropriate bit. Therefore, our FD_SET subprocedure will look like this:

```
P FD_SET          B                   EXPORT
D FD_SET          PI
```

```
D   peFD                        10I 0
D   peFDSet                     28A
D wkByteNo        S              5I 0
D wkMask          S              1A
D wkByte          S              1A
C                   callp     CalcBitPos(peFD:wkByteNo:wkMask)
c                   eval      wkByte = %subst(peFDSet:wkByteNo:1)
c                   biton     wkMask        wkByte
c                   eval      %subst(peFDSet:wkByteNo:1) = wkByte
P                 E
```

The opposite of FD_SET is FD_CLR. FD_CLR turns a single bit off in a descriptor set. The code to do this will be nearly the same thing as FD_SET, except that we turn the bit off instead of on. Like so:

```
P FD_CLR          B                   EXPORT
D FD_CLR          PI
D   peFD                        10I 0
D   peFDSet                     28A
D wkByteNo        S              5I 0
D wkMask          S              1A
D wkByte          S              1A
C                   callp     CalcBitPos(peFD:wkByteNo:wkMask)
c                   eval      wkByte = %subst(peFDSet:wkByteNo:1)
c                   bitoff    wkMask        wkByte
c                   eval      %subst(peFDSet:wkByteNo:1) = wkByte
P                 E
```

The last utility routine that we need is one that can be used to check to see if a bit is still on after calling select(). To make this work nicely in RPG, it'll help to return an indicator field -- so that we can simply check for *ON or *OFF when we call the routine.

This is done with the FD_ISSET macro in C. We simulate it in RPG like this:

```
P FD_ISSET        B                   EXPORT
D FD_ISSET        PI             1N
D   peFD                        10I 0
D   peFDSet                     28A
D wkByteNo        S              5I 0
D wkMask          S              1A
D wkByte          S              1A
C                   callp     CalcBitPos(peFD:wkByteNo:wkMask)
c                   eval      wkByte = %subst(peFDSet:wkByteNo:1)
c                   testb     wkMask        wkByte                   88
c                   return    *IN88
P                 E
```

Each of these subprocedures listed here should be placed in your SOCKUTILR4 service program. In addition, the prototypes for the FD_xxx procedures should be placed in the SOCKUTIL_H source member so that they are available when you call them from other programs.

The CalcBitPos prototype should be placed at the top of the SOCKUTILR4 member so that routines within that service program can call it. Nobody should need to call CalcBitPos from outside the service program.

You might also find it helpful to define a 28A field in your SOCKUTIL_H member called 'fdset'. (can't call it fd_set, since the name is already taken by the subprocedure) Then you can define your descriptors as being 'like(fdset)'.

For example, add this to SOCKUTIL_H:

```
D fdset           S              28A
```

Then use it like this:

```
D readset         S                    like(fdset)
D writeset        S                    like(fdset)
D excpset         S                    like(fdset)
```

If you prefer, all of the code for SOCKUTIL_H, SOCKUTILR4, SOCKET_H and ERRNO_H are on my website, and you can simply download them and use them directly.
http://www.scottklement.com/rpg/socktut/qrpglesrc.sockutil_h
http://www.scottklement.com/rpg/socktut/qrpglesrc.sockutilr4
http://www.scottklement.com/rpg/socktut/qrpglesrc.socket_h
http://www.scottklement.com/rpg/socktut/qrpglesrc.errno_h

> **Note:** Since I originally posted my FD_xxx code, and advice for using the select() API, I've received a little bit of criticism for using a character string for 'fd_set' instead of an array of integers. However, I stand by my decision to use a 28A field. I think it's nicer, especially because of the ability to use "like" to set the data type.

Now that you've added these routines to the SOCKUTILR4 service program, you'll need to recompile it. You want the new FD_xx routines to be exported so that they can be called from other programs.

To do that, edit the binding language source that you created in chapter 4, so that they now look like this:

```
STRPGMEXP PGMLVL(*CURRENT)
     EXPORT SYMBOL(RDLINE)
     EXPORT SYMBOL(WRLINE)
     EXPORT SYMBOL(FD_SET)
     EXPORT SYMBOL(FD_CLR)
     EXPORT SYMBOL(FD_ISSET)
     EXPORT SYMBOL(FD_ZERO)
ENDPGMEXP
STRPGMEXP PGMLVL(*PRV)
     EXPORT SYMBOL(RDLINE)
     EXPORT SYMBOL(WRLINE)
ENDPGMEXP
```

As you can see... we've changed the '*CURRENT' section to list all of the procedures that we're now exporting. We've also added a '*PRV' section that will create a 'previous version signature' for the binder. This previous version only contains the procedures that we had before our changes. Therefore, the service program will remain backward compatible, like we discussed back in Chapter 4.

Type the following commands to re-compile the SOCKUTILR4 service program:

```
CRTRPGMOD SOCKUTILR4 SRCFILE(SOCKTUT/QRPGLESRC) DBGVIEW(*LIST)
```

```
CRTSRVPGM SOCKUTILR4 EXPORT(*SRCFILE) SRCFILE(SOCKTUT/QSRVSRC) TEXT('Socket Utility
Service Pgm') ACTGRP(*CALLER)
```

6.4. A combination client/server example

Now that we've had a fairly thorough discussion of how the select API works, and we've put the appropriate utilities into our SOCKUTILR4 service program, we're ready to do an example program using select.

This example program is the "proxy" that was described in the introduction to this chapter. It's main purpose will be to allow us to use a telnet client to debug other client programs.

In fact, you'll be able to use it to interact with just about any client program around -- including your web browser... :)

Here's how this program will work:

1. We accept two port numbers as parms from the command line.
2. As we've done in the previous chapter, set up a socket to bind() and listen() to each of the two ports.
3. accept() a connection on the first port. Then close the listener for that port (we'll only handle one connection per port)
4. accept() a connection on the second port. Then close the listener for THAT port.
5. Set things up so that select() will detect when there is either data to read or an exceptional condition on either of these sockets.
6. If there was data to read on the first socket, simply send() that data to the other socket. and vice-versa.
7. If we receive any errors or exceptional conditions, exit the program. (It probably means that one side or the other has disconnected)
8. If we didn't receive any errors, go back to step 5.

Here's the source to this new example program. At the bottom of this page, I'll show you how to play with it. :)

File: SOCKTUT/QRPGLESRC Member: PROXY1

```
     H DFTACTGRP(*NO) ACTGRP(*NEW)
     H BNDDIR('SOCKTUT/SOCKUTIL') BNDDIR('QC2LE')

     D/copy socktut/qrpglesrc,socket_h
     D/copy socktut/qrpglesrc,sockutil_h
     D/copy socktut/qrpglesrc,errno_h

     D die             PR
     D   peMsg                      256A   const

     D NewListener     PR            10I 0
     D   pePort                       5U 0 value
     D   peError                    256A
```

```
D copysock        PR            10I 0
D  fromsock                     10I 0 value
D  tosock                       10I 0 value

D s1              S             10I 0
D s2              S             10I 0
D s1c             S             10I 0
D s2c             S             10I 0
D port1           S             15P 5
D port2           S             15P 5
D len             S             10I 0
D connfrom        S               *
D read_set        S                   like(fdset)
D excp_set        S                   like(fdset)
D errmsg          S            256A
D max             S             10I 0

c                   eval      *inlr = *on

C     *entry        plist
c                   parm                    port1
c                   parm                    port2

c                   if        %parms < 2
c                   callp     die('This program requires two port ' +
c                             'numbers as parameters!')
c                   return
c                   endif

C*************************************************
C* Listen on both ports given
C*************************************************
c                   eval      s1 = NewListener(port1: errmsg)
c                   if        s1 < 0
c                   callp     die(errmsg)
c                   return
c                   endif

c                   eval      s2 = NewListener(port2: errmsg)
c                   if        s2 < 0
c                   callp     die(errmsg)
c                   return
c                   endif

C*************************************************
C* Get a client on first port, then stop
C*   listening for more connections
C*************************************************
c                   eval      len = %size(sockaddr_in)
c                   eval      s1c = accept(s1: connfrom: len)
c                   if        s1c < 0
c                   eval      errmsg = %str(strerror(errno))
c                   callp     close(s1)
```

```
c                   callp     close(s2)
c                   callp     die(errmsg)
c                   return
c                   endif
c                   callp     close(s1)

C*************************************************
C* Get a client on second port, then stop
C*   listening for more connections
C*************************************************
c                   eval      len = %size(sockaddr_in)
c                   eval      s2c = accept(s2: connfrom: len)
c                   if        s2c < 0
c                   eval      errmsg = %str(strerror(errno))
c                   callp     close(s1c)
c                   callp     close(s2)
c                   callp     die(errmsg)
c                   return
c                   endif
c                   callp     close(s2)

c                   eval      max = s1c
c                   if        s2c > max
c                   eval      max = s2c
c                   endif

C*************************************************
C*  Main loop:
C*    1) create a descriptor set containing the
C*        "socket 1 client" descriptor and the
C*        "socket 2 client" descriptor
C*
C*    2) Wait until data appears on either the
C*        "s1c" socket or the "s2c" socket.
C*
C*    3) If data is found on s1c, copy it to
C*         s2c.
C*
C*    4) If data is found on s2c, copy it to
C*         s1c.
C*
C*    5) If any errors occur, close the sockets
C*         and end the proxy.
C*************************************************
c                   dow       1 = 1

c                   callp     FD_ZERO(read_set)
c                   callp     FD_SET(s1c: read_set)
c                   callp     FD_SET(s2c: read_set)
c                   eval      excp_set = read_set

c                   if        select(max+1: %addr(read_set): *NULL:
c                                %addr(excp_set): *NULL) < 0
```

```
c                   leave
c                   endif

c                   if        FD_ISSET(s1c: excp_set)
c                               or FD_ISSET(s2c: excp_set)
c                   leave
c                   endif

c                   if        FD_ISSET(s1c: read_set)
c                   if        copysock(s1c: s2c) < 0
c                   leave
c                   endif
c                   endif

c                   if        FD_ISSET(s2c: read_set)
c                   if        copysock(s2c: s1c) < 0
c                   leave
c                   endif
c                   endif

c                   enddo

c                   callp     close(s1c)
c                   callp     close(s2c)
c                   return
```

```
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  Create a new TCP socket that's listening to a port
 *
 *       parms:
 *         pePort = port to listen to
 *        peError = Error message (returned)
 *
 *    returns: socket descriptor upon success, or -1 upon error
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P NewListener     B
D NewListener     PI            10I 0
D   pePort                       5U 0 value
D   peError                    256A

D sock            S             10I 0
D len             S             10I 0
D bindto          S               *
D on              S             10I 0 inz(1)
D linglen         S             10I 0
D ling            S               *

C*** Create a socket
c                   eval      sock = socket(AF_INET:SOCK_STREAM:
c                                           IPPROTO_IP)
```

```
c                   if        sock < 0
c                   eval      peError = %str(strerror(errno))
c                   return    -1
c                   endif

C*** Tell socket that we want to be able to re-use the server
C***  port without waiting for the MSL timeout:
c                   callp     setsockopt(sock: SOL_SOCKET:
c                                SO_REUSEADDR: %addr(on): %size(on))

C*** create space for a linger structure
c                   eval      linglen = %size(linger)
c                   alloc     linglen       ling
c                   eval      p_linger = ling

C*** tell socket to only linger for 2 minutes, then discard:
c                   eval      l_onoff = 1
c                   eval      l_linger = 120
c                   callp     setsockopt(sock: SOL_SOCKET: SO_LINGER:
c                                ling: linglen)

C*** free up resources used by linger structure
c                   dealloc(E)              ling

C*** Create a sockaddr_in structure
c                   eval      len = %size(sockaddr_in)
c                   alloc     len           bindto
c                   eval      p_sockaddr = bindto

c                   eval      sin_family = AF_INET
c                   eval      sin_addr = INADDR_ANY
c                   eval      sin_port = pePort
c                   eval      sin_zero = *ALLx'00'

C*** Bind socket to port
c                   if        bind(sock: bindto: len) < 0
c                   eval      peError = %str(strerror(errno))
c                   callp     close(sock)
c                   dealloc(E)              bindto
c                   return    -1
c                   endif

C*** Listen for a connection
c                   if        listen(sock: 1) < 0
c                   eval      peError = %str(strerror(errno))
c                   callp     close(sock)
c                   dealloc(E)              bindto
c                   return    -1
c                   endif

C*** Return newly set-up socket:
c                   dealloc(E)              bindto
c                   return    sock
```

```
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This copies data from one socket to another.
 *    parms:
 *         fromsock = socket to copy data from
 *         tosock = socket to copy data to
 *
 *    returns:  length of data copied, or -1 upon error
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P copysock        B
D copysock        PI            10I 0
D  fromsock                     10I 0 value
D  tosock                       10I 0 value
D buf             S           1024A
D rc              S             10I 0
c                   eval      rc = recv(fromsock: %addr(buf): 1024: 0)
c                   if        rc < 0
c                   return    -1
c                   endif
c                   if        send(tosock: %addr(buf): rc: 0) < rc
c                   return    -1
c                   endif
c                   return    rc
P                 E

 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This ends this program abnormally, and sends back an escape.
 *   message explaining the failure.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P die             B
D die             PI
D   peMsg                      256A   const

D SndPgmMsg       PR                  ExtPgm('QMHSNDPM')
D   MessageID                    7A   Const
D   QualMsgF                    20A   Const
D   MsgData                    256A   Const
D   MsgDtaLen                   10I 0 Const
D   MsgType                     10A   Const
D   CallStkEnt                  10A   Const
D   CallStkCnt                  10I 0 Const
D   MessageKey                   4A
D   ErrorCode                32766A   options(*varsize)

D dsEC            DS
D  dsECBytesP             1      4I 0 INZ(256)
D  dsECBytesA             5      8I 0 INZ(0)
D  dsECMsgID              9     15
D  dsECReserv            16     16
D  dsECMsgDta            17    256
```

```
D wwMsgLen        S             10I 0
D wwTheKey        S              4A

c                   eval      wwMsgLen = %len(%trimr(peMsg))
c                   if        wwMsgLen<1
c                   return
c                   endif

c                   callp     SndPgmMsg('CPF9897': 'QCPFMSG   *LIBL':
c                               peMsg: wwMsgLen: '*ESCAPE':
c                               '*PGMBDY': 1: wwTheKey: dsEC)

c                   return
P                 E

 /define ERRNO_LOAD_PROCEDURE
 /copy socktut/qrpglesrc,errno_h
```

Compile it by typing: `CRTBNDRPG PROXY1 SRCFILE(SOCKTUT/QRPGLESRC) DBGVIEW(*LIST)`

To run it:

1. On the AS/400, type: CALL PROXY1 PARM(4000 4001)
2. On a PC, type: `telnet as400 4000` Then leave that window open.
3. Start a 2nd telnet client by typing: `telnet as400 4001`
4. Whatever you type in the first telnet client should appear in the other one's window, and vice-versa. If the client that you're using is a 'line-at-a-time' client, you may have to press enter before the line shows up.
5. When you tell one of the telnet clients to disconnect, the AS/400 program will end, and the other telnet client should be disconnected.

   (To disconnect using the Windows telnet client, click on the "Connect" menu, and choose "Disconnect". For FreeBSD or Linux clients, press Ctrl-] and then type quit)

Lets try it with a web browser:

1. On the AS/400, type: `CALL PROXY1 PARM(4000 4001)`
2. On a PC, type: `telnet as400 4000`

   > **Tip:** If you're using the Windows Telnet client, and you can't read what you're typing, you might try clicking "Terminal", then "Preferences" and enabling "Local Echo".

3. In a web browser, set the `Location:` to the URL: `http://as400:4001/`
4. Back in the telnet client, you'll see the web browser send a request for a web page. It'll also send a bunch of extra information about itself, such as the type of browser you're using, what types of documents it accepts, etc.
5. In the telnet client, type something like:

```
<H1>Hello Web Browser!</H1>
<H3>Nice subtitle, eh?</H3>
```

6. Then tell the telnet client to disconnect. Take a look at what the web browser has displayed!

I'm sure you can see how this could be a very valuable tool for debugging programs and testing what your clients and servers are doing!

6.5. Blocking vs. non-blocking sockets

So far in this chapter, you've seen that select() can be used to detect when data is available to read from a socket. However, there are times when its useful to be able to call send(), recv(), connect(), accept(), etc without having to wait for the result.

For example, let's say that you're writing a web browser. You try to connect to a web server, but the server isn't responding. When a user presses (or clicks) a stop button, you want the connect() API to stop trying to connect.

With what you've learned so far, that can't be done. When you issue a call to connect(), your program doesn't regain control until either the connection is made, or an error occurs.

The solution to this problem is called "non-blocking sockets".

By default, TCP sockets are in "blocking" mode. For example, when you call recv() to read from a stream, control isn't returned to your program until at least one byte of data is read from the remote site. This process of waiting for data to appear is referred to as "blocking". The same is true for the write() API, the connect() API, etc. When you run them, the connection "blocks" until the operation is complete.

Its possible to set a descriptor so that it is placed in "non-blocking" mode. When placed in non-blocking mode, you never wait for an operation to complete. This is an invaluable tool if you need to switch between many different connected sockets, and want to ensure that none of them cause the program to "lock up."

If you call "recv()" in non-blocking mode, it will return any data that the system has in it's read buffer for that socket. But, it won't wait for that data. If the read buffer is empty, the system will return from recv() immediately saying `"Operation Would Block!"`.

The same is true of the send() API. When you call send(), it puts the data into a buffer, and as it's read by the remote site, it's removed from the buffer. If the buffer ever gets "full", the system will return the error 'Operation Would Block" the next time you try to write to it.

Non-blocking sockets have a similar effect on the accept() API. When you call accept(), and there isn't already a client connecting to you, it will return 'Operation Would Block', to tell you that it can't complete the accept() without waiting...

The connect() API is a little different. If you try to call connect() in non-blocking mode, and the API can't connect instantly, it will return the error code for 'Operation In Progress'. When you call connect() again, later, it may tell you 'Operation Already In Progress' to let you know that it's still trying to connect, or it may give you a successful return code, telling you that the connect has been made.

Going back to the "web browser" example, if you put the socket that was connecting to the web server into non-blocking mode, you could then call connect(), print a message saying "connecting to host www.floofy.com..." then maybe do something else, and them come back to connect() again. If connect() works the second time, you might print "Host contacted, waiting for reply..." and then start calling send() and recv(). If the connect() is still pending, you might check to see if the user has pressed a "abort" button, and if so, call close() to stop trying to connect.

Non-blocking sockets can also be used in conjunction with the select() API. In fact, if you reach a point where you actually WANT to wait for data on a socket that was previously marked as "non-blocking", you could simulate a blocking recv() just by calling select() first, followed by recv().

The "non-blocking" mode is set by changing one of the socket's "flags". The flags are a series of bits, each one representing a different capability of the socket. So, to turn on non-blocking mode requires three steps:

1. Call the fcntl() API to retrieve the socket descriptor's current flag settings into a local variable.
2. In our local variable, set the O_NONBLOCK (non-blocking) flag on. (being careful, of course, not to tamper with the other flags)
3. Call the fcntl() API to set the flags for the descriptor to the value in our local variable.

6.6. The fcntl() API call

The previous topic explained how non-blocking mode works. This topic describes the API call that can be used to turn non-blocking mode on for a given socket. This API is capable of setting many other options as well, though they are mostly options that relate to stream files, not sockets, and are not covered by this tutorial.

The fcntl (file control) API is used to retrieve or set the flags that are assocırated with a socket or stream file. The IBM manual page for the fcntl() API (when used on a socket) can be found here: http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/sfcntl.htm

The C-language prototype for the fcntl API looks like this:

```
int fcntl(int descriptor, int command, ...);
```

It starts out pretty simple. We know that the API is called fcntl, and we know that the return value is an integer. We can tell that the first two parameters are both integers, as well. But what the heck is '...'?

The '...' parameter is a special construct in C to denote that the number of parameters remaining is variable. It furthermore means that the types of each of the remaining parameters can also vary. In short, in addition to the two integers, we can pass any number of additional parameters to the fcntl() API, and they can be of any data type. Not very helpful, is it?

Reading more of the manual page, however, we see that some of the values for the 'command' parameter of fcntl() accept an integer as the 3rd parameter. Other values for the 'command' parameter don't require a 3rd parameter at all. Since we now know that the '...' can only symbolize either an integer or nothing, we can write the prototype in RPG like this:

```
D fcntl           PR            10I 0 ExtProc('fcntl')
D   SocketDesc                  10I 0 Value
D   Command                     10I 0 Value
D   Arg                         10I 0 Value Options(*NOPASS)
```

But, since fcntl can also be used for stream files (much like close() can) we will use some compiler directives to prevent it being defined twice, should we want to use the socket APIs in the same program as the IFS (stream file) APIs. So, we'll do this:

```
D/if not defined(FCNTL_PROTOTYPE)
D fcntl           PR            10I 0 ExtProc('fcntl')
D   SocketDesc                  10I 0 Value
D   Command                     10I 0 Value
D   Arg                         10I 0 Value Options(*NOPASS)
D/define FCNTL_PROTOTYPE
D/endif
```

And, of course, add that to the SOCKET_H header file that we've been working on.

In addition, we'll need constants to define the possible values for the 'command' parameter of fcntl(). For the sake of being able to turn on the "non-blocking" flag, we're only concerned with two constants. 'F_GETFL' and 'F_SETFL' which are used to get and set the status flags, respectively.

These can be added to our SOCKET_H header file by typing this:

```
** fcntl() commands
D F_GETFL         C                   CONST(6)
D F_SETFL         C                   CONST(7)
```

The O_NONBLOCK flag also needs a constant so that we don't have to try to remember which flag is used for blocking/non-blocking :) The page in the manual tells us that 'O_NONBLOCK', 'O_NDELAY' and 'FNDELAY' all do the same thing, so we'll just define them all, like this:

```
** fcntl() flags
D O_NONBLOCK      C                   CONST(128)
D O_NDELAY        C                   CONST(128)
D FNDELAY         C                   CONST(128)
```

So... all of that should now be in your SOCKET_H member. Of course, if you simply downloaded my copy of SOCKET_H, it was already there :)

To call fcntl() to put a socket in non-blocking mode, you'll do something like this:

```
***** retrieve flags
c                   eval      flags = fcntl(sock: F_GETFL)
c                   if        flags < 0
C* handle error "unable to retrieve descriptor status flags"
c                   endif

***** turn non non-blocking flag:
c                   eval      flags = flags + O_NONBLOCK

***** set flags
c                   if        fcntl(sock: F_SETFL: flags) < 0
C* handle error "unable to set descriptor status flags"
c                   endif
```

6.7. A multi-client example of our server

Well, all that discussion about blocking and non-blocking sockets has made me hungry. Wanna stop and get a bite to eat? No?

This chapter will put everything we've learned to the test. We're going to take the server program that we wrote in Chapter 5, and re-write it so that it can handle many simultaneous clients.

From the client's perspective, our server won't have changed much. It'll still ask for a name, and still say hello and goodbye. However the internal functioning of our program will be very different.

This program will use non-blocking sockets and the select() API to switch between a whole array of connected clients. By spending only a few milliseconds on each client, then switching to the next client and spending time there, then switching again, etc, we can handle many clients at once.

However, this creates a number of challenges for us:

- We have to have some way of keeping track of where our client has "left off", so that when we switch to another client and come back, we'll be able to pick things up and continue.
- We won't be able to sit and wait until a client has sent us a x'0A' to terminate a line of text, because all the other clients would have to wait as well. Instead, we'll have to take whatever the client sends us and throw it into a buffer. Scan that buffer for a x'0A', and only act upon it after the x'0A' has been received.
- To prevent this program from taking up all of the available CPU time on the system, we'll need to keep track of when we're "waiting" for data, and when we're "sending" data so that if we don't have anything to send, we can just sit and wait and not hog the CPU.
- In the midst of all of these things, we'll need to watch for new connections from new clients as well as disconnects from existing clients. We'll have to make sure that we only check for input on connected descriptors, and ignore the disconnected ones.
- One nice thing about the server we wrote in Chapter 5 is that it's very simple. We don't send a lot of data, so running out of space in the system's write buffer isn't really a concern. Consequently, we can just use the WrLine() routine that we wrote in Chapter 4, and not worry about the fact that we're using non-blocking sockets with it... However, when we do a more complex server, we'll have to take that into consideration as well...

So here it is... the fruit of all of our labors, thus far! Hope I've described everything well enough. I strongly suggest that you "play with it", i.e. try to add features or write new things that it does, because you'll get a much better idea of what you're doing that way.

File: SOCKTUT/QRPGLESRC, Member: SERVEREX3

```
     H DFTACTGRP(*NO) ACTGRP(*NEW)
     H BNDDIR('SOCKTUT/SOCKUTIL') BNDDIR('QC2LE')

      *** header files for calling service programs & APIs

     D/copy socktut/qrpglesrc,socket_h
     D/copy socktut/qrpglesrc,sockutil_h
     D/copy socktut/qrpglesrc,errno_h
```

```
 *** Prototypes for externally called programs:

D Translate       PR                  ExtPgm('QDCXLATE')
D    peLength                    5P 0 const
D    peBuffer                32766A   options(*varsize)
D    peTable                    10A   const

 *** Prototypes for local subprocedures:

D die             PR
D   peMsg                      256A   const

D NewListener     PR            10I 0
D   pePort                       5U 0 value
D   peError                    256A

D NewClient       PR            10I 0
D   peServ                      10I 0 value

D ReadClient      PR            10I 0
D   peClient                    10I 0 value

D HandleClient    PR            10I 0
D   peClient                    10I 0 value

D EndClient       PR            10I 0
D   peClient                    10I 0 value

D GetLine         PR            10I 0
D   peClient                    10I 0 value
D   peLine                     256A

 *** Configuration

D MAXCLIENTS      C                   CONST(100)

 *** Global Variables:

D Msg             S            256A
D to              S               *
D tolen           S             10I 0
D serv            S             10I 0
D max             S             10I 0
D rc              S             10I 0
D C               S             10I 0
D readset         S                   like(fdset)
D excpset         S                   like(fdset)
D endpgm          S              1N   inz(*off)

 *** Variables in the "client" data structure are kept
 *** seperate for each connected client socket.
```

```
D Client          DS                  Occurs(MAXCLIENTS)
D   sock                        10I 0
D   wait                         1N
D   rdbuf                      256A
D   rdbuflen                    10I 0
D   state                       10I 0
D   line                       256A


c                   eval      *inlr = *on

c                   exsr      Initialize

C*********************************************************
C*  Main execution loop:
C*
C*     1) Make 2 descriptor sets.  One to check which
C*         sockets have data to read, one to check which
C*         sockets have exceptional conditions pending.
C*         Each set will contain the listner socket, plus
C*         each connected client socket
C*
C*    2) Call select() to find out which descriptors need
C*         to be read from or have exceptions pending.
C*         We have a timeout value set here as well.  It's
C*         set to 1 minute if all sockets are waiting for
C*         user input, or 1/10 second if the sockets need
C*         us to write data to them.  (the 1/10 second is
C*         just to keep this program from gobbling too
C*         much CPU time)
C*
C*    3) Check to see if a user told us to shut down, or
C*         if the job/subsystem/system has requested us to
C*         end the program.
C*
C*    4) If the listener socket ("server socket") has data
C*         to read, it means someone is trying to connect
C*         to us, so call the NewClient procedure.
C*
C*    5) Check each socket for incoming data and load into
C*         the appropriate read buffer.
C*
C*    6) Do the next "task" that each socket needs.
C*         (could be sending a line of text, or waiting
C*         for input, or disconnecting, etc)
C*********************************************************
c                   dow       1 = 1

c                   exsr      MakeDescSets

c                   eval      rc = select(max+1: %addr(readset):
c                                *NULL: %addr(excpset): to)
```

```
c                   exsr      ChkShutDown

c                   if        rc > 0
c                   if        FD_ISSET(serv: readset)
c                   callp     NewClient(serv)
c                   endif
c                   exsr      CheckSockets
c                   endif

c                   exsr      DoClients

c                   enddo


C*===============================================================
C* Initialize some program vars & set up a server socket:
C*===============================================================
CSR   Initialize    begsr
C*------------------------
c                   do        MAXCLIENTS    C
c     C             occur     client
c                   eval      sock = -1
c                   callp     EndClient(C)
c                   enddo

c                   eval      tolen = %size(timeval)
c                   alloc     tolen         to
c                   eval      p_timeval = to

C*********************************************************
C* Start listening to port 4000
C*********************************************************
c                   eval      serv = NewListener(4000: Msg)
c                   if        serv < 0
c                   callp     die(Msg)
c                   endif
C*------------------------
CSR                 endsr


C*===============================================================
C* This sets up the "readset" and "excpset" descriptor sets
C*  for use with select().   It also calculates the appropriate
C*  timeout value, and the maximum descriptor number to check
C*===============================================================
CSR   MakeDescSets  begsr
C*------------------------
c                   callp     FD_ZERO(readset)
c                   callp     FD_SET(serv: readset)

C* By default, set a 60 second timeout...  but if one or more
C* of the client’s is not in ’wait’ mode, change that timeout
C* to only 100,000 microseconds (1/10 second)
```

```
c                   eval      tv_sec = 60
c                   eval      tv_usec = 0
c                   eval      max = serv

c                   do        MAXCLIENTS    C
c     C             occur     client
c                   if        sock <> -1
c                   callp     FD_SET(sock: readset)
c                   if        not wait
c                   eval      tv_sec = 0
c                   eval      tv_usec = 100000
c                   endif
c                   if        sock > max
c                   eval      max = sock
c                   endif
c                   endif
c                   enddo

C* We can just copy excpset to readset... no point in going thru
C*  all of that again :)
c                   eval      excpset = readset
C*-----------------------
CSR                 endsr


C*===============================================================
C*  Check for a 'shutdown' condition.  If shutdown was requested
C*  tell all connected sockets, and then close them.
C*===============================================================
CSR   ChkShutDown   begsr
C*-----------------------
c                   shtdn                                        99
c                   if        *in99 = *on
c                   eval      endpgm = *On
c                   endif

 * Note that the 'endpgm' flag can also be set by the
 *   'HandleClient' subprocedure, not just the code above...

c                   if        endpgm = *on
c                   do        MAXCLIENTS    C
c     C             occur     client
c                   if        sock <> -1
c                   callp     WrLine(sock: 'Sorry!  We''re shutting ' +
c                              'down now!')
c                   callp     EndClient(C)
c                   endif
c                   enddo
c                   callp     close(serv)
c                   callp     Die('shut down requested...')
c                   return
c                   endif
C*-----------------------
```

```
CSR                 endsr


C*===============================================================
C*  This loads any data that has been sent by the various client
C*  sockets into their respective read buffers, and also checks
C*  for clients that may have disconnected:
C*===============================================================
CSR   CheckSockets  begsr
C*------------------------
c                   do        MAXCLIENTS    C

c     C             occur     client

c                   if        sock <> -1

c                   if        FD_ISSET(sock: readset)
c                   if        ReadClient(C) < 0
c                   callp     EndClient(C)
c                   callp     FD_CLR(sock: excpset)
c                   endif
c                   endif

c                   if        FD_ISSET(sock: excpset)
c                   callp     EndClient(C)
c                   endif

c                   endif

c                   enddo
C*------------------------
CSR                 endsr

C*===============================================================
C*  This finally gets down to "talking" to the client programs.
C*  It switches between each connected client, and then sends
C*  data or receives data as appropriate...
C*===============================================================
CSR   DoClients     begsr
C*------------------------
c                   do        MAXCLIENTS    C
c     C             occur     client
c                   if        sock <> -1
c                   callp     HandleClient(C)
c                   endif
c                   enddo
C*------------------------
CSR                 endsr


 *++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  Create a new TCP socket that's listening to a port
 *
```

```
 *       parms:
 *         pePort = port to listen to
 *        peError = Error message (returned)
 *
 *    returns: socket descriptor upon success, or -1 upon error
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P NewListener     B
D NewListener     PI            10I 0
D   pePort                       5U 0 value
D   peError                    256A

D sock            S             10I 0
D len             S             10I 0
D bindto          S               *
D on              S             10I 0 inz(1)
D linglen         S             10I 0
D ling            S               *
D flags           S             10I 0

C*** Create a socket
c                   eval      sock = socket(AF_INET:SOCK_STREAM:
c                                           IPPROTO_IP)
c                   if        sock < 0
c                   eval      peError = %str(strerror(errno))
c                   return    -1
c                   endif

C*** Tell socket that we want to be able to re-use the server
C***  port without waiting for the MSL timeout:
c                   callp     setsockopt(sock: SOL_SOCKET:
c                                SO_REUSEADDR: %addr(on): %size(on))

C*** create space for a linger structure
c                   eval      linglen = %size(linger)
c                   alloc     linglen       ling
c                   eval      p_linger = ling

C*** tell socket to only linger for 2 minutes, then discard:
c                   eval      l_onoff = 1
c                   eval      l_linger = 120
c                   callp     setsockopt(sock: SOL_SOCKET: SO_LINGER:
c                                ling: linglen)

C*** free up resources used by linger structure
c                   dealloc(E)              ling

C*** tell socket we don’t want blocking...
c                   eval      flags = fcntl(sock: F_GETFL)
c                   eval      flags = flags + O_NONBLOCK
c                   if        fcntl(sock: F_SETFL: flags) < 0
c                   eval      peError = %str(strerror(errno))
c                   return    -1
c                   endif
```

```
C*** Create a sockaddr_in structure
c                   eval      len = %size(sockaddr_in)
c                   alloc     len           bindto
c                   eval      p_sockaddr = bindto

c                   eval      sin_family = AF_INET
c                   eval      sin_addr = INADDR_ANY
c                   eval      sin_port = pePort
c                   eval      sin_zero = *ALLx'00'

C*** Bind socket to port
c                   if        bind(sock: bindto: len) < 0
c                   eval      peError = %str(strerror(errno))
c                   callp     close(sock)
c                   dealloc(E)              bindto
c                   return    -1
c                   endif

C*** Listen for a connection
c                   if        listen(sock: MAXCLIENTS) < 0
c                   eval      peError = %str(strerror(errno))
c                   callp     close(sock)
c                   dealloc(E)              bindto
c                   return    -1
c                   endif

C*** Return newly set-up socket:
c                   dealloc(E)              bindto
c                   return    sock
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This accepts a new client connection, and adds him to
 *   the 'client' data structure.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P NewClient       B
D NewClient       PI            10I 0
D   peServ                      10I 0 value

D X               S             10I 0
D S               S             10I 0
D cl              S             10I 0
D flags           S             10I 0
D ling            S               *
D connfrom        S               *
D len             S             10I 0
D Msg             S             52A

C*************************************************
C* See if there is an empty spot in the data
C* structure.
```

```
C*************************************************
c                   eval      cl = 0
c                   do        MAXCLIENTS    X
c     X             occur     Client
c                   if        sock = -1
c                   eval      cl = X
c                   leave
c                   endif
c                   enddo

C*************************************************
C*  Accept new connection
C*************************************************
c                   eval      len = %size(sockaddr_in)
c                   alloc     len           connfrom

c                   eval      S = accept(peServ: connfrom: len)
c                   if        S < 0
c                   return    -1
c                   endif

c                   dealloc(E)              connfrom

C*************************************************
C* Turn off blocking & limit lingering
C*************************************************
c                   eval      flags = fcntl(S: F_GETFL: 0)
c                   eval      flags = flags + O_NONBLOCK
c                   if        fcntl(S: F_SETFL: flags) < 0
c                   eval      Msg = %str(strerror(errno))
c                   dsply                   Msg
c                   return    -1
c                   endif

c                   eval      len = %size(linger)
c                   alloc     len           ling
c                   eval      p_linger = ling
c                   eval      l_onoff = 1
c                   eval      l_linger = 120
c                   callp     setsockopt(S: SOL_SOCKET: SO_LINGER:
c                                ling: len)
c                   dealloc(E)              ling

C*************************************************
C*  If we've already reached the maximum number
C*  of connections, let client know and then
C*  get rid of him
C*************************************************
c                   if        cl = 0
c                   callp     wrline(S: 'Maximum number of connect' +
c                              'tions has been reached!')
c                   callp     close(s)
c                   return    -1
```

```
c                   endif

C*************************************************
C*  Add client into the structure
C*************************************************
c     cl            occur     client
c                   eval      sock = S
c                   return    0
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  If there is data to be read from a Client's socket, add it
 *    to the client's buffer, here...
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P ReadClient      B
D ReadClient      PI            10I 0
D   peClient                    10I 0 value

D left            S             10I 0
D p_read          S               *
D err             S             10I 0
D len             S             10I 0

c     peClient      occur     client

c                   eval      left = %size(rdbuf) - rdbuflen
c                   eval      p_read = %addr(rdbuf) + rdbuflen

c                   eval      len = recv(sock: p_read: left: 0)
c                   if        len < 0
c                   eval      err = errno
c                   if        err = EWOULDBLOCK
c                   return    0
c                   else
c                   return    -1
c                   endif
c                   endif

c                   eval      rdbuflen = rdbuflen + len
c                   return    len
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This disconnects a client and cleans up his spot in the
 *    client data structure.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P EndClient       B
D EndClient       PI            10I 0
D   peClient                    10I 0 value
c     peClient      occur     client
c                   if        sock >= 0
```

```
c                   callp     close(sock)
c                   endif
c                   eval      sock = -1
c                   eval      wait = *off
c                   eval      rdbuf = *Blanks
c                   eval      rdbuflen = 0
c                   eval      state = 0
c                   eval      line = *blanks
c                   return    0
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  As we're switching between each different client, this
 *  routine is called to handle whatever the next 'step' is
 *  for a given client.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P HandleClient    B
D HandleClient    PI            10I 0
D   peClient                    10I 0 value

c     peClient      occur     client

c                   select
c                   when      state = 0
c                   callp     WrLine(sock: 'Please enter your name' +
c                                   ' now!')
c                   eval      state = 1

c                   when      state = 1
c                   eval      wait = *on
c                   if        GetLine(peClient: line) > 0
c                   eval      wait = *off
c                   eval      state = 2
c                   endif

c                   when      state = 2
c                   if        %trim(line) = 'quit'
c                   eval      endpgm = *on
c                   eval      state = 4
c                   else
c                   callp     WrLine(sock: 'Hello ' + %trim(line))
c                   eval      state = 3
c                   endif

c                   when      state = 3
c                   callp     WrLine(sock: 'Goodbye ' + %trim(line))
c                   eval      state = 4

c                   when      state = 4
c                   callp     EndClient(peClient)
c                   endsl
```

```
c                   return    0
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This removes one line of data from a client's read buffer
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P GetLine         B
D GetLine         PI             10I 0
D   peClient                     10I 0 value
D   peLine                      256A

D pos             S              10I 0

C*** Load correct client:
c     peClient      occur     client
c                   if        rdbuflen < 1
c                   return    0
c                   endif

C*** Look for an end-of-line character:
c                   eval      pos = %scan(x'0A': rdbuf)
c                   if        pos < 1 or pos > rdbuflen
c                   return    0
c                   endif

C*** Add line to peLine variable, and remove from rdbuf:
c                   eval      peLine = %subst(rdbuf:1:pos-1)

c                   if        pos < %size(rdbuf)
c                   eval      rdbuf = %subst(rdBuf:pos+1)
c                   else
c                   eval      rdbuf = *blanks
c                   endif

c                   eval      rdbuflen = rdbuflen - pos

C*** If CR character found, remove that too...
c                   eval      pos = pos - 1
c                   if        %subst(peLine:pos:1) = x'0D'
c                   eval      peLine = %subst(peLine:1:pos-1)
c                   eval      pos = pos - 1
c                   endif

C*** Convert to EBCDIC:
c                   if        pos > 0
c                   callp     Translate(pos: peLine: 'QTCPEBC')
c                   endif

C*** return length of line:
c                   return    pos
P                 E
```

```
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This ends this program abnormally, and sends back an escape.
 *   message explaining the failure.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P die             B
D die             PI
D   peMsg                      256A   const

D SndPgmMsg       PR                  ExtPgm('QMHSNDPM')
D   MessageID                    7A   Const
D   QualMsgF                    20A   Const
D   MsgData                    256A   Const
D   MsgDtaLen                   10I 0 Const
D   MsgType                     10A   Const
D   CallStkEnt                  10A   Const
D   CallStkCnt                  10I 0 Const
D   MessageKey                   4A
D   ErrorCode                32766A   options(*varsize)

D dsEC            DS
D  dsECBytesP             1      4I 0 INZ(256)
D  dsECBytesA             5      8I 0 INZ(0)
D  dsECMsgID              9     15
D  dsECReserv            16     16
D  dsECMsgDta            17    256

D wwMsgLen        S             10I 0
D wwTheKey        S              4A

c                   eval      wwMsgLen = %len(%trimr(peMsg))
c                   if        wwMsgLen<1
c                   return
c                   endif

c                   callp     SndPgmMsg('CPF9897': 'QCPFMSG   *LIBL':
c                               peMsg: wwMsgLen: '*ESCAPE':
c                               '*PGMBDY': 1: wwTheKey: dsEC)

c                   return
P                 E

 /define ERRNO_LOAD_PROCEDURE
 /copy socktut/qrpglesrc,errno_h
```

6.8. Adding some refinements to our approach

Not too bad for our first multi-client server program, is it? There's some room for improvement however. This topic will discuss what we could've done better, and what to do about it.

First of all, we're sending any outgoing data with the WrLine procedure. WrLine() is designed to be used with blocking sockets. As I mentioned before, this doesn't really cause problems in our sample program because there's only a small amount of data to send. The chances of us somehow filling up the system's TCP buffer is very remote, so we probably won't ever notice the problem.

But in a real application? One where files are being sent, or data is being typed by another user? We'd be sure to have problems! As soon as the TCP buffer is full, data would simply be cut off.

Speaking of buffers, the read buffer has a problem too. Right now, we add to the read buffer. That's not a problem, however, we only take data out of the read buffer when a x'0A' (end-of-line) character is found. That means that if a client should fill up his read buffer without having an end of line character, it'll simply STOP receiving data from that client!

The 'wait' indicator is a neat idea, but it wouldn't really be necessary if we handled writes correctly. If we handled them correctly, data would go into a write buffer, and be written as soon as writing to the socket was feasible. If we did that, there'd be little point to having a 'wait' indicator, we'd just always do the full 60 second timeout.

Finally, why are we setting the 'readset' for a client every single time we do a select()? We should only set descriptor on if we have space in our read buffer. That way, we don't have to go through the motions of trying to do a recv() when we know there isn't anyplace to put the data.

I suggest that you try to solve all of these problems yourself. It'll be a good challenge for you -- trying to implement a write buffer, and handle a full read buffer, etc.

If you get stuck, take a look at my solution to the problem, which is online at this address:
http://www.scottklement.com/rpg/socktut/qrpglesrc.serverex4

6.9. A Chat Application

In the last topic, you improved upon the 'example server' that we've been using since the start of the server portion of this tutorial. Yes, it's a nice, simple, example of writing a server.

But how practical is all of this knowledge you've picked up? Can you do anything truly useful with it?

Okay. Lets modify that last example, the 'simple example server' program and turn it into a simple 'chat room'.

Here's what we want to do:

1. Each message going from the server to the client should be prefixed by a unique 'message number'. This makes it much easier for a client program to understand the purpose for each message we send.
2. When a client program connects, the first message (message number 100) will ask the client for his name.
3. If the name is a good one, the server will respond with message number 101. "Login Accepted"
4. From this point on, each line that the client types will be sent to all of the other clients that are logged in. Each of these messages will be prefixed by the name of the person using the client which sent the message.

This will involve surprisingly few changes to the program that we created in the previous topic.

Here's our 'chat server':

```
File: SOCKTUT/QRPGLESRC, Member: SERVEREX5

     H DFTACTGRP(*NO) ACTGRP(*NEW)
     H BNDDIR('SOCKTUT/SOCKUTIL') BNDDIR('QC2LE')

     *** header files for calling service programs & APIs

     D/copy socktut/qrpglesrc,socket_h
     D/copy socktut/qrpglesrc,sockutil_h
     D/copy socktut/qrpglesrc,errno_h

     *** Prototypes for externally called programs:

     D Translate       PR                  ExtPgm('QDCXLATE')
     D    peLength                    5P 0 const
     D    peBuffer                32766A   options(*varsize)
     D    peTable                    10A   const

     *** Prototypes for local subprocedures:

     D die             PR
     D   peMsg                      256A   const

     D NewListener     PR            10I 0
     D   pePort                       5U 0 value
     D   peError                    256A

     D NewClient       PR            10I 0
     D   peServ                      10I 0 value

     D ReadClient      PR            10I 0
     D   peClient                    10I 0 value

     D WriteClient     PR            10I 0
     D   peClient                    10I 0 value

     D HandleClient    PR            10I 0
     D   peClient                    10I 0 value

     D EndClient       PR            10I 0
     D   peClient                    10I 0 value

     D GetLine         PR            10I 0
     D   peClient                    10I 0 value
     D   peLine                     256A

     D PutLine         PR            10I 0
     D   peClient                    10I 0 value
     D   peLine                     256A   const

     *** Configuration
```

```
D MAXCLIENTS      C                   CONST(100)

*** Global Variables:

D Msg             S            256A
D to              S               *
D tolen           S             10I 0
D serv            S             10I 0
D max             S             10I 0
D rc              S             10I 0
D C               S             10I 0
D readset         S                   like(fdset)
D excpset         S                   like(fdset)
D writeset        S                   like(fdset)
D endpgm          S              1N   inz(*off)

*** Variables in the "client" data structure are kept
*** seperate for each connected client socket.

D Client          DS                  Occurs(MAXCLIENTS)
D   sock                        10I 0
D   rdbuf                      256A
D   rdbuflen                    10I 0
D   state                       10I 0
D   wrbuf                     2048A
D   wrbuflen                    10I 0
D   name                        20A


c                   eval      *inlr = *on

c                   exsr      Initialize

C**********************************************************
C*  Main execution loop:
C*
C*     1) Make read/write/exception descriptor sets.
C*         and figure out the timeout value for select()
C*
C*    2) Call select() to find out which descriptors need
C*         data to be written or read, and also to find
C*         any exceptions to handle.
C*
C*    3) Check to see if a user told us to shut down, or
C*         if the job/subsystem/system has requested us to
C*         end the program.
C*
C*    4) If the listener socket ("server socket") has data
C*         to read, it means someone is trying to connect
C*         to us, so call the NewClient procedure.
C*
C*    5) Check each socket for incoming data and load into
```

```
C*         the appropriate read buffer.
C*
C*    6) Check each client for outgoing data and write into
C*         the appropriate socket.
C*
C*    7) Do the next "task" that each socket needs.
C*         (could be sending a line of text, or waiting
C*         for input, or disconnecting, etc)
C*********************************************************
c                   dow       1 = 1

c                   exsr      MakeDescSets

c                   eval      rc = select(max+1: %addr(readset):
c                                %addr(writeset): %addr(excpset): to)

c                   exsr      ChkShutDown

c                   if        rc > 0
c                   if        FD_ISSET(serv: readset)
c                   callp     NewClient(serv)
c                   endif
c                   exsr      CheckSockets
c                   endif

c                   exsr      DoClients

c                   enddo


C*===============================================================
C* Initialize some program vars & set up a server socket:
C*===============================================================
CSR   Initialize    begsr
C*------------------------
c                   do        MAXCLIENTS    C
c     C             occur     client
c                   eval      sock = -1
c                   callp     EndClient(C)
c                   enddo

c                   eval      tolen = %size(timeval)
c                   alloc     tolen         to
c                   eval      p_timeval = to

C*********************************************************
C* Start listening to port 4000
C*********************************************************
c                   eval      serv = NewListener(4000: Msg)
c                   if        serv < 0
c                   callp     die(Msg)
c                   endif
C*------------------------
```

```
CSR                 endsr


C*===============================================================
C*  This makes the descriptor sets:
C*     readset -- includes the 'server' (listener) socket plus
C*          any clients that still have space in their read buffer
C*    writeset -- includes any clients that have data in their
C*          write buffer.
C*     excpset -- includes all socket descriptors, since all of
C*          them should be checked for exceptional conditions
C*===============================================================
CSR   MakeDescSets  begsr
C*------------------------
c                   callp     FD_ZERO(writeset)
c                   callp     FD_ZERO(readset)
c                   callp     FD_ZERO(excpset)

c                   callp     FD_SET(serv: readset)
c                   callp     FD_SET(serv: excpset)

C* the 60 second timeout is just so that we can check for
C*  system shutdown periodically.
c                   eval      tv_sec = 60
c                   eval      tv_usec = 0
c                   eval      max = serv

c                   do        MAXCLIENTS    C
c     C             occur     client
c                   if        sock <> -1
c                   callp     FD_SET(sock: excpset)
c                   if        rdbuflen < %size(rdbuf)
c                   callp     FD_SET(sock: readset)
c                   endif
c                   if        wrbuflen > 0
c                   callp     FD_SET(sock: writeset)
c                   endif
c                   if        sock > max
c                   eval      max = sock
c                   endif
c                   endif
c                   enddo
C*------------------------
CSR                 endsr


C*===============================================================
C*  Check for a 'shutdown' condition.  If shutdown was requested
C*  tell all connected sockets, and then close them.
C*===============================================================
CSR   ChkShutDown   begsr
C*------------------------
c                   shtdn                                        99
```

```
c                   if        *in99 = *on
c                   eval      endpgm = *On
c                   endif

 * Note that the 'endpgm' flag can also be set by the
 *   'HandleClient' subprocedure, not just the code above...

c                   if        endpgm = *on
c                   do        MAXCLIENTS    C
c     C             occur     client
c                   if        sock <> -1
c                   callp     WrLine(sock: '902 Sorry!  We''re ' +
c                              'shutting down now!')
c                   callp     EndClient(C)
c                   endif
c                   enddo
c                   callp     close(serv)
c                   callp     Die('shut down requested...')
c                   return
c                   endif
C*-------------------------
CSR                 endsr


C*===============================================================
C*  This reads any data that's waiting to be read from each
C*   socket, and writes any data that's waiting to be written.
C*
C*  Also disconnects any socket that returns an error, or has
C*  and exceptional condition pending.
C*===============================================================
CSR   CheckSockets  begsr
C*-------------------------
c                   do        MAXCLIENTS    C

c     C             occur     client

c                   if        sock <> -1

c                   if        FD_ISSET(sock: readset)
c                   if        ReadClient(C) < 0
c                   callp     EndClient(C)
c                   callp     FD_CLR(sock: excpset)
c                   callp     FD_CLR(sock: writeset)
c                   endif
c                   endif

c                   if        FD_ISSET(sock: writeset)
c                   if        WriteClient(C) < 0
c                   callp     EndClient(C)
c                   callp     FD_CLR(sock: excpset)
c                   callp     FD_CLR(sock: writeset)
c                   endif
```

```
c                   endif

c                   if        FD_ISSET(sock: excpset)
c                   callp     EndClient(C)
c                   endif

c                   endif

c                   enddo
C*------------------------
CSR                 endsr


C*===============================================================
C*  This finally gets down to "talking" to the client programs.
C*  It switches between each connected client, and then sends
C*  data or receives data as appropriate...
C*===============================================================
CSR   DoClients     begsr
C*------------------------
c                   do        MAXCLIENTS    C
c     C             occur     client
c                   if        sock <> -1
c                   callp     HandleClient(C)
c                   endif
c                   enddo
C*------------------------
CSR                 endsr


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  Create a new TCP socket that's listening to a port
 *
 *       parms:
 *         pePort = port to listen to
 *        peError = Error message (returned)
 *
 *    returns: socket descriptor upon success, or -1 upon error
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P NewListener     B
D NewListener     PI            10I 0
D   pePort                       5U 0 value
D   peError                    256A

D sock            S             10I 0
D len             S             10I 0
D bindto          S               *
D on              S             10I 0 inz(1)
D linglen         S             10I 0
D ling            S               *
D flags           S             10I 0

C*** Create a socket
```

```
c                   eval      sock = socket(AF_INET:SOCK_STREAM:
c                                           IPPROTO_IP)
c                   if        sock < 0
c                   eval      peError = %str(strerror(errno))
c                   return    -1
c                   endif

C*** Tell socket that we want to be able to re-use the server
C***  port without waiting for the MSL timeout:
c                   callp     setsockopt(sock: SOL_SOCKET:
c                                SO_REUSEADDR: %addr(on): %size(on))

C*** create space for a linger structure
c                   eval      linglen = %size(linger)
c                   alloc     linglen       ling
c                   eval      p_linger = ling

C*** tell socket to only linger for 2 minutes, then discard:
c                   eval      l_onoff = 1
c                   eval      l_linger = 120
c                   callp     setsockopt(sock: SOL_SOCKET: SO_LINGER:
c                                ling: linglen)

C*** free up resources used by linger structure
c                   dealloc(E)              ling

C*** tell socket we don't want blocking...
c                   eval      flags = fcntl(sock: F_GETFL)
c                   eval      flags = flags + O_NONBLOCK
c                   if        fcntl(sock: F_SETFL: flags) < 0
c                   eval      peError = %str(strerror(errno))
c                   return    -1
c                   endif

C*** Create a sockaddr_in structure
c                   eval      len = %size(sockaddr_in)
c                   alloc     len           bindto
c                   eval      p_sockaddr = bindto

c                   eval      sin_family = AF_INET
c                   eval      sin_addr = INADDR_ANY
c                   eval      sin_port = pePort
c                   eval      sin_zero = *ALLx'00'

C*** Bind socket to port
c                   if        bind(sock: bindto: len) < 0
c                   eval      peError = %str(strerror(errno))
c                   callp     close(sock)
c                   dealloc(E)              bindto
c                   return    -1
c                   endif

C*** Listen for a connection
```

```
c                   if        listen(sock: MAXCLIENTS) < 0
c                   eval      peError = %str(strerror(errno))
c                   callp     close(sock)
c                   dealloc(E)              bindto
c                   return    -1
c                   endif

C*** Return newly set-up socket:
c                   dealloc(E)              bindto
c                   return    sock
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This accepts a new client connection, and adds him to
 *   the 'client' data structure.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P NewClient       B
D NewClient       PI            10I 0
D   peServ                      10I 0 value

D X               S             10I 0
D S               S             10I 0
D cl              S             10I 0
D flags           S             10I 0
D ling            S               *
D connfrom        S               *
D len             S             10I 0
D Msg             S             52A

C*************************************************
C* See if there is an empty spot in the data
C* structure.
C*************************************************
c                   eval      cl = 0
c                   do        MAXCLIENTS    X
c     X             occur     Client
c                   if        sock = -1
c                   eval      cl = X
c                   leave
c                   endif
c                   enddo

C*************************************************
C*  Accept new connection
C*************************************************
c                   eval      len = %size(sockaddr_in)
c                   alloc     len           connfrom

c                   eval      S = accept(peServ: connfrom: len)
c                   if        S < 0
c                   return    -1
c                   endif
```

```
c                   dealloc(E)              connfrom

C**************************************************
C* Turn off blocking & limit lingering
C**************************************************
c                   eval      flags = fcntl(S: F_GETFL: 0)
c                   eval      flags = flags + O_NONBLOCK
c                   if        fcntl(S: F_SETFL: flags) < 0
c                   eval      Msg = %str(strerror(errno))
c                   dsply                   Msg
c                   return    -1
c                   endif

c                   eval      len = %size(linger)
c                   alloc     len           ling
c                   eval      p_linger = ling
c                   eval      l_onoff = 1
c                   eval      l_linger = 120
c                   callp     setsockopt(S: SOL_SOCKET: SO_LINGER:
c                                ling: len)
c                   dealloc(E)              ling

C**************************************************
C*  If we've already reached the maximum number
C*  of connections, let client know and then
C*  get rid of him
C**************************************************
c                   if        cl = 0
c                   callp     wrline(S: '901 Maximum number of ' +
c                              'connections has been reached!')
c                   callp     close(s)
c                   return    -1
c                   endif

C**************************************************
C*  Add client into the structure
C**************************************************
c     cl            occur     client
c                   eval      sock = S
c                   return    0
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  If there is data to be read from a Client's socket, add it
 *    to the client's buffer, here...
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P ReadClient      B
D ReadClient      PI            10I 0
D   peClient                    10I 0 value

D left            S             10I 0
```

```
D p_read          S               *
D err             S             10I 0
D len             S             10I 0

c     peClient      occur     client

c                   eval      left = %size(rdbuf) - rdbuflen
c                   eval      p_read = %addr(rdbuf) + rdbuflen

c                   eval      len = recv(sock: p_read: left: 0)
c                   if        len < 0
c                   eval      err = errno
c                   if        err = EWOULDBLOCK
c                   return    0
c                   else
c                   return    -1
c                   endif
c                   endif

c                   eval      rdbuflen = rdbuflen + len
c                   return    len
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  If a client's socket is ready to write data to, and theres
 *    data to write, go ahead and write it...
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P WriteClient     B
D WriteClient     PI            10I 0
D   peClient                    10I 0 value

D len             S             10I 0

c     peClient      occur     client
c                   if        wrbuflen < 1
c                   return    0
c                   endif

c                   eval      len = send(sock:%addr(wrbuf):wrbuflen:0)

c                   if        len > 0
c                   eval      wrbuf = %subst(wrbuf: len+1)
c                   eval      wrbuflen = wrbuflen - len
c                   endif

c                   return    len
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This disconnects a client and cleans up his spot in the
 *    client data structure.
```

```
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P EndClient       B
D EndClient       PI            10I 0
D   peClient                    10I 0 value
c     peClient      occur     client
c                   if        sock >= 0
c                   callp     close(sock)
c                   endif
c                   eval      sock = -1
c                   eval      rdbuf = *Blanks
c                   eval      rdbuflen = 0
c                   eval      state = 0
c                   eval      wrbuflen = 0
c                   eval      wrbuf = *Blanks
c                   return    0
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  As we're switching between each different client, this
 *  routine is called to handle whatever the next 'step' is
 *  for a given client.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P HandleClient    B
D HandleClient    PI            10I 0
D   peClient                    10I 0 value

D X               S             10I 0
D from            S             24A
D msg             S            256A

c     peClient      occur     client

c                   select
c                   when      state = 0
c                   callp     PutLine(peClient: '100 Please enter ' +
c                                   'your name now!')
c                   eval      state = 1

c                   when      state = 1
c                   if        GetLine(peClient: msg) > 0

c                   if        %trim(msg) = 'quit'
c                   eval      endpgm = *on
c                   callp     PutLine(peClient: ' ')
c                   eval      state = 3
c                   else
c                   eval      name = %trim(msg)
c                   callp     PutLine(peClient: '101 Login Accepted.')
c                   eval      state = 2
c                   endif

c                   endif
```

```
c                   when      state = 2
c                   if        GetLine(peClient: msg) > 0
c                   eval      from = '200 ' + name

C*         copy message to each client:
c                   do        MAXCLIENTS    X
c     X             occur     client
c                   if        sock <> -1 and state > 1
c                   callp     PutLine(X: %trimr(from) + ': ' + msg)
c                   endif
c                   enddo

c     peClient      occur     client
c                   endif


c                   when      state = 3
c                   callp     EndClient(peClient)
c                   endsl

c                   return    0
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This removes one line of data from a client's read buffer
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P GetLine         B
D GetLine         PI            10I 0
D   peClient                    10I 0 value
D   peLine                     256A

D pos             S             10I 0

C*** Load correct client:
c     peClient      occur     client
c                   if        rdbuflen < 1
c                   return    0
c                   endif

C*** Look for an end-of-line character:
c                   eval      pos = %scan(x'0A': rdbuf)

C*** If buffer is completely full, take the whole thing
C***   even if there is no end-of-line char.
c                   if        pos<1 and rdbuflen>=%size(rdbuf)
c                   eval      peLine = rdbuf
c                   eval      pos = rdbuflen
c                   eval      rdbuf = *blanks
c                   eval      rdbuflen = 0
c                   callp     Translate(pos: peLine: 'QTCPEBC')
c                   return    pos
```

```
c                   endif

C*** otherwise, only return something if end of line existed
c                   if        pos < 1 or pos > rdbuflen
c                   return    0
c                   endif

C*** Add line to peLine variable, and remove from rdbuf:
c                   eval      peLine = %subst(rdbuf:1:pos-1)

c                   if        pos < %size(rdbuf)
c                   eval      rdbuf = %subst(rdBuf:pos+1)
c                   else
c                   eval      rdbuf = *blanks
c                   endif

c                   eval      rdbuflen = rdbuflen - pos

C*** If CR character found, remove that too...
c                   eval      pos = pos - 1
c                   if        %subst(peLine:pos:1) = x'0D'
c                   eval      peLine = %subst(peLine:1:pos-1)
c                   eval      pos = pos - 1
c                   endif

C*** Convert to EBCDIC:
c                   if        pos > 0
c                   callp     Translate(pos: peLine: 'QTCPEBC')
c                   endif

C*** return length of line:
c                   return    pos
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  Add a line of text onto the end of a client's write buffer
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P PutLine         B
D PutLine         PI            10I 0
D   peClient                    10I 0 value
D   peLine                     256A   const

D wkLine          S            258A
D saveme          S             10I 0
D len             S             10I 0

c                   occur     client        saveme
c     peClient      occur     client

C* Add CRLF & calculate length & translate to ASCII
c                   eval      wkLine = %trimr(peLine) + x'0D25'
c                   eval      len = %len(%trimr(wkLine))
```

```
c                   callp     Translate(len: wkLine: 'QTCPASC')

C* make sure we don't overflow buffer
c                   if        (wrbuflen+len) > %size(wrbuf)
c                   eval      len = %size(wrbuf) - wrbuflen
c                   endif
c                   if        len < 1
c     saveme        occur     client
c                   return    0
c                   endif

C* add data onto end of buffer
c                   eval      %subst(wrbuf:wrbuflen+1) =
c                                  %subst(wkLine:1:len)
c                   eval      wrbuflen = wrbuflen + len
c     saveme        occur     client
c                   return    len
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This ends this program abnormally, and sends back an escape.
 *   message explaining the failure.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P die             B
D die             PI
D   peMsg                      256A   const

D SndPgmMsg       PR                  ExtPgm('QMHSNDPM')
D   MessageID                    7A   Const
D   QualMsgF                    20A   Const
D   MsgData                    256A   Const
D   MsgDtaLen                   10I 0 Const
D   MsgType                     10A   Const
D   CallStkEnt                  10A   Const
D   CallStkCnt                  10I 0 Const
D   MessageKey                   4A
D   ErrorCode                32766A   options(*varsize)

D dsEC            DS
D  dsECBytesP             1      4I 0 INZ(256)
D  dsECBytesA             5      8I 0 INZ(0)
D  dsECMsgID              9     15
D  dsECReserv            16     16
D  dsECMsgDta            17    256

D wwMsgLen        S             10I 0
D wwTheKey        S              4A

c                   eval      wwMsgLen = %len(%trimr(peMsg))
c                   if        wwMsgLen<1
c                   return
c                   endif
```

```
c                   callp     SndPgmMsg('CPF9897': 'QCPFMSG   *LIBL':
c                               peMsg: wwMsgLen: '*ESCAPE':
c                               '*PGMBDY': 1: wwTheKey: dsEC)

c                   return
P                 E

 /define ERRNO_LOAD_PROCEDURE
 /copy socktut/qrpglesrc,errno_h
```

Chapter 7. Handling many sockets by spawning jobs

Written by Scott Klement.

7.1. Overview of job spawning approach

The last chapter explained in detail how to write server programs that could handle many simultaneous clients, all being served by one program in one job.

As I explained in the introduction to that chapter, that type of coding has it's advantages. It's main advantages are that the various connected clients can share data very easily through the server, and that it uses less system resources than some of the other methods of handling multiple clients.

It has some big disadvantages too! The coding can get very complicated! If something that one client is doing should crash the program, it'll crash all of the other clients as well! Everything in that server must run with the same user profile, so all client's have the same authority to system settings.

This chapter will discuss the opposite method. This method will use the SBMJOB command to "spawn" a new job for every client that connects. It will then become the new job's responsibility to send & receive data from that client.

Since the operating system will handle the multitasking for us, and the operating system is undoubtedly better at it than our programs are, this will give better performance for very network-intensive applications.

It will however, make communications between the various connected clients extremely difficult. You'd have a very hard time writing a 'chat room' using this method -- even if you did get it to work, it would almost certainly be less efficient.

The 'job spawning approach' requires two (or more) programs in order to do it's basic duty. The first program is the 'listener' program. It listens and accepts new client connections, then submits a job to handle each client. The second program is the 'server instance' program, it handles all of the socket communications for a single client.

Here's the basic pseudocode of the 'listener program':

1. A socket is created that listens for connections.
2. accept() is called to receive a new connection.
3. a new job is submitted to handle this new connection.
4. We call the givedescriptor() API to tell the operating system that its okay for the new job to take the client's socket descriptor from us.
5. We close() the client's descriptor, we're done with it.
6. Go back to step #2.

Here's the basic pseudocode of the 'server instance' program:

1. Communicate our job information back to the listener program so that it knows who to give a descriptor to.
2. Call the takedescriptor() API to take the client's socket descriptor from the listener program.

3. send() and recv() data to/from the client application.
4. close the socket
5. end.

7.2. The takedescriptor() and givedescriptor() API calls

The givedescriptor() and takedescriptor() API calls are vital for implementing a server using the 'job spawning approach.' They allow a listener program to pass a descriptor to a server instance program.

At first glance, it sounds as if this should be easy. After all, a socket descriptor is just a number, right? And passing a number from one program to another isn't that hard, is it?

Well, unfortunately it's not that easy. First of all, there's security. Understandably, you don't want one job to be able to read data from a socket that another job has open! If that were allowed, I could write a simple loop to try every descriptor on the system and peek into the affairs of every network job on the system! Yikes!

Secondly, each open descriptor is an attribute of the job it was opened in. Job A and Job B can both have a descriptor #5. And they can be different things. The memory that the system uses to keep track of each descriptor is allocated to each individual job. If it weren't, then jobs could interfere with each other -- and possibly crash one another. While this may be acceptable in Windows, it certainly is not in OS/400!

So, in order to pass a descriptor from one job to another, you MUST use the givedescriptor() and takedescriptor() APIs.

The givedescriptor() API is documented in the IBM manual page at this location:
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/gvsoc.htm

This manual lists the C language prototype for the givedescriptor() API as looking like this:

```
int givedescriptor(int descriptor, char *target_job);
```

Pretty straight forward, right? It's called 'givedescriptor', and it returns an integer. It has two parameters, the first one is an integer, and the second one is a pointer.

To make the same prototype in RPG, we add this to our SOCKET_H member:

```
D givedescriptor  PR            10I 0 extproc('givedescriptor')
D   SockDesc                    10I 0 VALUE
D   target_job                    *   VALUE
```

The "usage notes" tell us that the "target_job" parameter is the 'internal job identifier' of the job that we want to give the descriptor to. And that we can look up this job identifier using the QUSRJOBI API.

The IBM manual page for the takedescriptor() API is located here:
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/tksoc.htm

And it tells us that the C language prototype for the takedescriptor() API looks like this:

```
int takedescriptor(char *source_job);
```

So, the API is called 'takedescriptor'. It returns an integer, and accepts one parameter, a pointer. This makes the RPG prototype look like this:

```
D takedescriptor  PR            10I 0 extproc('takedescriptor')
D   source_job                    *   VALUE
```

So add that prototype to your SOCKET_H member as well.

The usage notes for the takedescriptor() API tell us that the source_job is also the 'internal job identifier' which we can retrieve using the QUSRJOBI API. It also tells us that if we set the source_job parameter to NULL, it'll take any descriptor that the system tries to pass to it.

You can call these APIs in RPG by doing something like this:

In the "listener" program:

```
c                   if        givedescriptor(sock: target_int_id) < 0
c*** handle failure
c                   endif
```

And in the "server instance" program:

```
c                   eval      sock = takedescriptor(*NULL)
```

The tricky part, of course, is getting the 'target_int_id', which we shall discuss in the next section.

7.3. Retrieving the internal job id

The takedescriptor() and getdescriptor() APIs require that you know the internal job ID of the job that you wish to pass a descriptor to. This topic discusses the QUSRJOBI (Retrieve Job Information) API which we will use to determine the Internal Job Identifier.

The QUSRJOBI API is an 'Work Management API' (most of the other APIs we've worked with have been 'Unix-Type APIs', which makes it somewhat different.

The IBM manual page for QUSRJOBI is located here:
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/qusrjobi.htm

The prototype listed on this page isn't in C format, but rather is in a somewhat 'language-neutral' format. It looks like this:

Parm#	Description	Usage	Data Type
1	Receiver variable	Output	Char (*)
2	Length of receiver variable	Input	Binary (4)
3	Format name	Input	Char (8)
4	Qualified job name	Input	Char (26)
5	Internal job identifier	Input	Char (16)
Optional parameter:			

Parm#	Description	Usage	Data Type
6	Error code	I/O	Char (*)

QUSRJOBI is actually a program on the system that we need to call (as opposed to a procedure in a service program) The 2nd column of the parameter group listing above tells us whether the parameter is used for "Output" from the API, "Input" to the API, or "I/O" for both input and output.

The Last column explains the data type of the variable used, and that data type's length. A (*) under length indicates that the length can vary, otherwise, the length is the number of bytes to pass.

Since this is a "common gotcha" for people who are calling APIs, I'll say it again. The length is the *number of bytes to pass* . When it says "Binary(4)", this is NOT the RPG '4B 0' data type. In RPG, '4B 0' means a 4 digit binary number. But, a 4-digit binary number is only 2 bytes long! Instead, a '9B 0' should be passed, or much better yet, a '10I 0' should be passed. In general, the RPG 'I' data type works better for anything called 'binary' than the legacy 'B' data type.

So, the prototype for the QUSRJOBI API will look like this:

```
D RtvJobInf       PR                  ExtPgm('QUSRJOBI')
D   RcvVar                  32766A   options(*varsize)
D   RcvVarLen                  10I 0 CONST
D   Format                      8A   CONST
D   JobName                    26A   CONST
D   IntJobID                   16A   CONST
D   ErrorCode               32766A   options(*varsize)
```

> **Note:** Each parameter that's to be 'Input' only to the API is marked as 'CONST'. This is always a good idea, as it protects your program from accidental changes, helps to document the usage of the API, and allows you to substitute expressions for the parameters.

The parameters that can vary in size we used the "options(*varsize)" keyword with. This allows us to call the API using different sized variables for each call.

You'll note that QUSRJOBI return's all of it's data in the 'RcvVar' parm, so we need an idea of what format the data will be in. In fact, most APIs can return data in more than one format, and this format is determined by the 'FORMAT' parameter.

Reading more of the IBM manual page tells us that there are many different formats. Since we only need one that returns the internal job id, we can use any of the formats that it allows. The shortest, and best performing format is called 'JOBI0100', so why don't we use that?

The manual tells us that the format of the returned data for QUSRJOBI looks like this:

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Number of bytes returned
4	4	BINARY(4)	Number of bytes available
8	8	CHAR(10)	Job name

Offset			
Dec	**Hex**	**Type**	**Field**
18	12	CHAR(10)	User name
28	1C	CHAR(6)	Job number
34	22	CHAR(16)	Internal job identifier
50	32	CHAR(10)	Job status
60	3C	CHAR(1)	Job type
61	3D	CHAR(1)	Job subtype
62	3E	CHAR(2)	Reserved
64	40	BINARY(4)	Run priority
68	44	BINARY(4)	Time slice
72	48	BINARY(4)	Default wait
76	4C	CHAR(10)	Purge

So, we'll make a data structure to use as our receiver variable. That'll help us by separating all of the fields listed above when we get our returned data from the API.

Here's what this data structure should look like:

```
D dsJobI0100      DS
D   JobI_ByteRtn                10I 0
D   JobI_ByteAvl                10I 0
D   JobI_JobName                10A
D   JobI_UserID                 10A
D   JobI_JobNbr                  6A
D   JobI_IntJob                 16A
D   JobI_Status                 10A
D   JobI_Type                    1A
D   JobI_SbType                  1A
D   JobI_Reserv1                 2A
D   JobI_RunPty                 10I 0
D   JobI_TimeSlc                10I 0
D   JobI_DftWait                10I 0
D   JobI_Purge                  10A
```

And, finally, the 'Error Code' data structure. This is used in a lot of different APIs, and is a convienient way to get returned error info.

It looks like this:

```
D dsEC            DS
D  dsECBytesP           1      4I 0 INZ(256)
D  dsECBytesA           5      8I 0 INZ(0)
D  dsECMsgID            9     15
D  dsECReserv          16     16
D  dsECMsgDta          17    256
```

Since this prototype & data struct definition don't (directly) have anything to do with sockets, I decided to make a new header file for this API, called 'JOBINFO_H'

If you want to download my copy of JOBINFO_H, you can get it here:
http://www.scottklement.com/rpg/socktut/qrpglesrc.jobinfo_h
(http://www.scottklement.com/rpg/socktut/qrpglesrc.jobinfo_h)

You call the QUSRJOBI API like this:

```
C                   callp     RtvJobInf(dsJOBI0100: %size(dsJOBI0100):
C                               'JOBI0100': '*': *BLANKS: dsEC)
c                   if        dsECBytesA > 0
C** Error occurred.   Error message number is in dsECMsgID
C                   endif
```

7.4. Communicating the job information

The QUSRJOBI API can be used to give us the Internal Job ID that the givedescriptor() API requires. But, before we can call the QUSRJOBI API, we need to know the JobName/Userid/JobNbr of the server instance job.

This leads to a problem. If we haven't submitted the job yet, how can we possibly know it's job number? And even after we've submitted it, how do we find out the number?

The easiest way is to submit the server instance job, and then have it look up it's own job information. Once it has that information, it should communicate it back to the listener program, so that givedescriptor() can be called.

The easiest way for the server instance to communicate back to the listener program is by using a data queue.

The data queue API's are described in the Object APIs manual. You can find them online at this URL:
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/obj2.htm

In our situation, we are interested in calling two of these APIs, they are the 'Receive From Data Queue' API, and the 'Send to a Data Queue' API. (QRCVDTAQ an QSNDDTAQ, respectively)

Here are the parameters listed in the manual for the QSNDDTAQ API:

Required Parameter Group:			
1	Data queue name	Input	Char(10)
2	Library name	Input	Char(10)
3	Length of data	Input	Packed(5,0)
4	Data	Input	Char(*)
Optional Parameter Group 1:			
5	Length of key data	Input	Packed(3,0)
6	Key data	Input	Char(*)
Optional Parameter Group 2:			
7	Asynchronous request	Input	Char(10)

The RPG prototype for QSNDDTAQ looks like this:

```
D SndDtaQ         PR                  ExtPgm('QSNDDTAQ')
D  dtaqname                     10A   const
D  dtaqlib                      10A   const
D  dtaqlen                       5P 0 const
D  data                      32766A   const options(*varsize)
D  keylen                        3P 0 const options(*nopass)
D  keydata                   32766A   const options(*varsize: *nopass)
D  asyncreq                     10A   const options(*nopass)
```

Here are the parameters listed in the manual for the QRCVDTAQ API:

Required Parameter Group:			
1	Data queue name	Input	Char(10)
2	Library name	Input	Char(10)
3	Length of data	Output	Packed(5,0)
4	Data	Output	Char(*)
5	Wait time	Input	Packed(5,0)
Optional Parameter Group 1:			
6	Key order	Input	Char(2)
7	Length of key data	Input	Packed(3,0)
8	Key data	I/O	Char(*)
9	Length of sender information	Input	Packed(3,0)
10	Sender information	Output	Char(*)
Optional Parameter Group 2:			
11	Remove message	Input	Char(10)
12	Size of data receiver	Input	Packed(5,0)
13	Error code	I/O	Char(*)

So, an RPG prototype for the QRCVDTAQ API looks like this:

```
D RcvDtaQ         PR                  ExtPgm('QRCVDTAQ')
D   DtaqName                    10A   const
D   DtaqLib                     10A   const
D   DtaqLen                      5P 0
D   Data                     32766A   options(*varsize)
D   WaitTime                     5P 0 const
D   KeyOrder                     2A   const options(*nopass)
D   KeyLen                       3P 0 const options(*nopass)
D   KeyData                  32766A   options(*varsize: *nopass)
D   SenderLen                    3P 0 const options(*nopass)
D   SenderInfo               32766A   options(*varsize: *nopass)
D   RmvMsg                      10A   const options(*nopass)
D   RcvVarSize                   5P 0 const options(*nopass)
D   ErrorCode                32766A   options(*varsize: *nopass)
```

Each time one of our server instances starts, it will call QUSRJOBI to look up it's internal job ID. It will then send it's internal job ID, along with it's job name, userid, and job number to the data queue by calling QSNDDTAQ like this:

```
D dsJobInfo       DS
D   MsgType                1     10A
D   JobName               11     20A
D   JobUser               21     30A
D   JobNbr                31     36A
D   InternalID            65     80A

c                   callp     SndDtaQ('SVREX6DQ': 'SOCKTUT': 80:
c                                   dsJobInfo)
```

Note: We're making the data queue length be 80, and including a 'MsgType' field at the start in case we ever want to make this data queue compatible with those used by display files, icf files, etc.

After the Listener program has submitted its next server instance pgm, it will read an entry off of the data queue to find out the job info it needs to call givedescriptor(). It will call the QRCVDTAQ API like this:

```
c                   callp     RcvDtaQ('SVREX6DQ': 'SOCKTUT': Len:
c                               dsJobInfo: 60)
c                   if        Len < 80
c*** timeout occurred.
c                   else
C*** dsJobInfo is populated
c                   endif
```

Please add the prototypes shown in this topic, as well as the 'dsJobInfo' data structure to the JOBINFO_H member that you created earlier in this chapter.

Or, if you prefer, you can download my copy of JOBINFO_H here:
http://www.scottklement.com/rpg/socktut/qrpglesrc.jobinfo_h

We will use this JOBINFO_H member for all of our 'job spawning approach' examples.

7.5. Our server as a multi-job server

So, we've learned what the "job spawning approach" is, and we've learned about the givedescriptor() and takedescriptor() APIs, how to get the internal job id, and how to communicate that id back to the listener job.

Great. Time to put this knowledge to use!

Our "hello/goodbye" server that we originally wrote in Chapter 5 can now be made into a job spawning server. This will involve the two programs that we've discussed in this chapter, the "listener" and the "server instance" programs.

I've decided to name them "svrex6l" (server example 6 -- listener) and "svrex6i" (server example 6 -- instance). And here they are:

```
File SOCKTUT/QRPGLESRC, Member SVREX6L:
```

```
H DFTACTGRP(*NO) ACTGRP(*NEW)
H BNDDIR('SOCKTUT/SOCKUTIL') BNDDIR('QC2LE')

 *** header files for calling service programs & APIs

D/copy socktut/qrpglesrc,socket_h
D/copy socktut/qrpglesrc,sockutil_h
D/copy socktut/qrpglesrc,errno_h
D/copy socktut/qrpglesrc,jobinfo_h

 *** prototypes for external calls

D Cmd             PR                  ExtPgm('QCMDEXC')
D   command                    200A   const
D   length                      15P 5 const

 *** Prototypes for local subprocedures:

D die             PR
D   peMsg                      256A   const

D NewListener     PR            10I 0
D   pePort                       5U 0 value
D   peError                    256A

D KillEmAll       PR

 *** local variables & constants

D MAXCLIENTS      C                   CONST(256)

D svr             S             10I 0
D cli             S             10I 0
D msg             S            256A
D err             S             10I 0
D calen           S             10I 0
D clientaddr      S               *
D jilen           S              5P 0

c                   eval      *inlr = *on

C*************************************************
C* Clean up any previous instances of the dtaq
C*************************************************
c                   callp(e)  Cmd('DLTDTAQ SOCKTUT/SVREX6DQ': 200)
c                   callp(e)  Cmd('CRTDTAQ DTAQ(SOCKTUT/SVREX6DQ) ' +
c                                       ' MAXLEN(80) TEXT("Data ' +
c                                       ' queue for SVREX6L")': 200)
c                   if        %error
c                   callp     Die('Unable to create data queue!')
c                   return
c                   endif
```

```
C*************************************************
C* Start listening for connections on port 4000
C*************************************************
c                   eval      svr = NewListener(4000: msg)
c                   if        svr < 0
c                   callp     die(msg)
c                   return
c                   endif

C*************************************************
C* create a space to put client addr struct into
C*************************************************
c                   eval      calen = %size(sockaddr_in)
c                   alloc     calen         clientaddr

c                   dow       1 = 1

C***********************************
C* Get a new server instance ready
C***********************************
c                   callp(e)  Cmd('SBMJOB CMD(CALL PGM(SVREX6I))' +
c                                       ' JOB(SERVERINST) ' +
c                                       ' JOBQ(QSYSNOMAX) ' +
c                                       ' JOBD(QDFTJOBD) ' +
c                                       ' RTGDTA(QCMDB)': 200)
c                   if        %error
c                   callp     close(svr)
c                   callp     KillEmAll
c                   callp     Die('Unable to submit a new job to ' +
c                             'process clients!')
c                   return
c                   endif

C***********************************
C* Accept a new client conn
C***********************************
c                   eval      cli = accept(svr: clientaddr: calen)
c                   if        cli < 0
c                   eval      err = errno
c                   callp     close(svr)
c                   callp     KillEmAll
c                   callp     die('accept(): ' + %str(strerror(err)))
c                   return
c                   endif

c                   if        calen <> %size(sockaddr_in)
c                   callp     close(cli)
c                   eval      calen = %size(sockaddr_in)
c                   iter
c                   endif

C***********************************
C* get the internal job id of a
```

```
C*  server instance to handle client
C************************************
c                   eval      jilen = %size(dsJobInfo)
c                   callp     RcvDtaQ('SVREX6DQ': 'SOCKTUT': jilen:
c                                     dsJobInfo: 60)
c                   if        jilen < 80
c                   callp     close(cli)
c                   callp     KillEmAll
c                   callp     close(svr)
c                   callp     die('No response from server instance!')
c                   return
c                   endif

C************************************
C* Pass descriptor to svr instance
C************************************
c                   if        givedescriptor(cli: %addr(InternalID))<0
c                   eval      err = errno
c                   callp     close(cli)
c                   callp     KillEmAll
c                   callp     close(svr)
c                   callp     Die('givedescriptor(): ' +
c                                 %str(strerror(err)))
c                   Return
c                   endif

c                   callp     close(cli)
c                   enddo


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This ends any server instances that have been started, but
 *   have not been connected with clients.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P KillEmAll       B
D KillEmAll       PI
c                   dou       jilen < 80

c                   eval      jilen = %size(dsJobInfo)
c                   callp     RcvDtaQ('SVREX6DQ': 'SOCKTUT': jilen:
c                                     dsJobInfo: 1)

c                   if        jilen >= 80

c                   callp(E)  Cmd('ENDJOB JOB(' + %trim(JobNbr) +
c                                  '/' + %trim(JobUser) + '/' +
c                                  %trim(jobName) + ') OPTION(*IMMED)'+
c                                  ' LOGLMT(0)': 200)

C                   endif

c                   enddo
P                 E
```

```
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  Create a new TCP socket that's listening to a port
 *
 *       parms:
 *         pePort = port to listen to
 *        peError = Error message (returned)
 *
 *    returns: socket descriptor upon success, or -1 upon error
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P NewListener     B
D NewListener     PI            10I 0
D   pePort                       5U 0 value
D   peError                    256A

D sock            S             10I 0
D len             S             10I 0
D bindto          S               *
D on              S             10I 0 inz(1)
D linglen         S             10I 0
D ling            S               *

C*** Create a socket
c                   eval      sock = socket(AF_INET:SOCK_STREAM:
c                                           IPPROTO_IP)
c                   if        sock < 0
c                   eval      peError = %str(strerror(errno))
c                   return    -1
c                   endif

C*** Tell socket that we want to be able to re-use the server
C***  port without waiting for the MSL timeout:
c                   callp     setsockopt(sock: SOL_SOCKET:
c                                SO_REUSEADDR: %addr(on): %size(on))

C*** create space for a linger structure
c                   eval      linglen = %size(linger)
c                   alloc     linglen       ling
c                   eval      p_linger = ling

C*** tell socket to only linger for 2 minutes, then discard:
c                   eval      l_onoff = 1
c                   eval      l_linger = 120
c                   callp     setsockopt(sock: SOL_SOCKET: SO_LINGER:
c                                ling: linglen)

C*** free up resources used by linger structure
c                   dealloc(E)              ling

C*** Create a sockaddr_in structure
c                   eval      len = %size(sockaddr_in)
c                   alloc     len           bindto
```

```
c                   eval      p_sockaddr = bindto

c                   eval      sin_family = AF_INET
c                   eval      sin_addr = INADDR_ANY
c                   eval      sin_port = pePort
c                   eval      sin_zero = *ALLx'00'

C*** Bind socket to port
c                   if        bind(sock: bindto: len) < 0
c                   eval      peError = %str(strerror(errno))
c                   callp     close(sock)
c                   dealloc(E)              bindto
c                   return    -1
c                   endif

C*** Listen for a connection
c                   if        listen(sock: MAXCLIENTS) < 0
c                   eval      peError = %str(strerror(errno))
c                   callp     close(sock)
c                   dealloc(E)              bindto
c                   return    -1
c                   endif

C*** Return newly set-up socket:
c                   dealloc(E)              bindto
c                   return    sock
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This ends this program abnormally, and sends back an escape.
 *   message explaining the failure.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P die             B
D die             PI
D   peMsg                      256A   const

D SndPgmMsg       PR                  ExtPgm('QMHSNDPM')
D   MessageID                    7A   Const
D   QualMsgF                    20A   Const
D   MsgData                    256A   Const
D   MsgDtaLen                   10I 0 Const
D   MsgType                     10A   Const
D   CallStkEnt                  10A   Const
D   CallStkCnt                  10I 0 Const
D   MessageKey                   4A
D   ErrorCode                32766A   options(*varsize)

D dsEC            DS
D  dsECBytesP             1      4I 0 INZ(256)
D  dsECBytesA             5      8I 0 INZ(0)
D  dsECMsgID              9     15
D  dsECReserv            16     16
```

```
D  dsECMsgDta            17    256

D wwMsgLen        S             10I 0
D wwTheKey        S              4A

c                   eval      wwMsgLen = %len(%trimr(peMsg))
c                   if        wwMsgLen<1
c                   return
c                   endif

c                   callp     SndPgmMsg('CPF9897': 'QCPFMSG   *LIBL':
c                               peMsg: wwMsgLen: '*ESCAPE':
c                               '*PGMBDY': 1: wwTheKey: dsEC)

c                   return
P                 E

 /define ERRNO_LOAD_PROCEDURE
 /copy socktut/qrpglesrc,errno_h
```

File SOCKTUT/QRPGLESRC, member SVREX6I:

```
H DFTACTGRP(*NO) ACTGRP(*NEW)
H BNDDIR('SOCKTUT/SOCKUTIL') BNDDIR('QC2LE')

 *** header files for calling service programs & APIs

D/copy socktut/qrpglesrc,socket_h
D/copy socktut/qrpglesrc,sockutil_h
D/copy socktut/qrpglesrc,errno_h
D/copy socktut/qrpglesrc,jobinfo_h

 *** Prototypes for local subprocedures:

D die             PR
D   peMsg                      256A   const

D GetClient       PR            10I 0

D cli             S             10I 0
D name            S             80A

c                   eval      *inlr = *on

c                   eval      cli = GetClient
c                   if        cli < 0
c                   callp     Die('Failure retrieving client socket '+
c                              'descriptor.')
c                   return
c                   endif

c                   callp     WrLine(cli: 'Please enter your ' +
```

```
c                                 'name now!')

c                   if        RdLine(cli: %addr(name): 80: *On) < 0
c                   callp     close(cli)
c                   callp     Die('RdLine() failed!')
c                   return
c                   endif

c                   callp     WrLine(cli: 'Hello ' + %trim(name) + '!')
c                   callp     WrLine(cli: 'Goodbye ' +%trim(name)+ '!')

c                   callp     close(cli)
c                   return


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  Get the new client from the listener application
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P GetClient       B
D GetClient       PI            10I 0

D jilen           S              5P 0
D sock            S             10I 0

c                   callp     RtvJobInf(dsJobI0100: %size(dsJobI0100):
c                               'JOBI0100': '*': *BLANKS: dsEC)
c                   if        dsECBytesA > 0
c                   return    -1
c                   endif

c                   eval      JobName = JobI_JobName
c                   eval      JobUser = JobI_UserID
c                   eval      JobNbr = JobI_JobNbr
c                   eval      InternalID = JobI_IntJob

c                   eval      jilen = %size(dsJobInfo)

c                   callp     SndDtaq('SVREX6DQ': 'SOCKTUT': jilen:
c                                dsJobInfo)

c                   eval      sock = TakeDescriptor(*NULL)
c                   return    sock
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This ends this program abnormally, and sends back an escape.
 *   message explaining the failure.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P die             B
D die             PI
D   peMsg                      256A   const
```

```
D SndPgmMsg       PR                  ExtPgm('QMHSNDPM')
D   MessageID                     7A   Const
D   QualMsgF                     20A   Const
D   MsgData                     256A   Const
D   MsgDtaLen                    10I 0 Const
D   MsgType                      10A   Const
D   CallStkEnt                   10A   Const
D   CallStkCnt                   10I 0 Const
D   MessageKey                    4A
D   ErrorCode                 32766A   options(*varsize)

D dsEC            DS
D  dsECBytesP             1      4I 0 INZ(256)
D  dsECBytesA             5      8I 0 INZ(0)
D  dsECMsgID              9     15
D  dsECReserv            16     16
D  dsECMsgDta            17    256

D wwMsgLen        S             10I 0
D wwTheKey        S              4A

c                   eval      wwMsgLen = %len(%trimr(peMsg))
c                   if        wwMsgLen<1
c                   return
c                   endif

c                   callp     SndPgmMsg('CPF9897': 'QCPFMSG   *LIBL':
c                               peMsg: wwMsgLen: '*ESCAPE':
c                               '*PGMBDY': 1: wwTheKey: dsEC)

c                   return
P                 E

 /define ERRNO_LOAD_PROCEDURE
 /copy socktut/qrpglesrc,errno_h
```

7.6. Working towards our next example

If you haven't yet tried the first "job spawning" example, try it now.

Compile it by typing:

`CRTBNDRPG SVREX6L SRCFILE(SOCKTUT/QRPGLESRC) DBGVIEW(*LIST)`

`CRTBNDRPG SVREX6I SRCFILE(SOCKTUT/QRPGLESRC) DBGVIEW(*LIST)`

And then start it like so:

`SBMJOB CMD(CALL SVREX6L) JOBQ(QSYSNOMAX) JOB(LISTENER)`

Test it from your PC by doing a 'telnet as400 4000', as we've discussed in the past few chapters. Look! Nice!

When you're done playing with it, type 'NETSTAT *CNN' and find the reference to 'Local Port' 4000, that's in 'Listen' state. Use the '4' option to end the server.

Looking back at the sourcecode for SVREX6L, you'll see that when SVREX6L receives an error on the accept() command, it goes and does an 'ENDJOB' command to the server instance. So, in fact, when you entered the 4 in NETSTAT, SVREX6L ended all of the programs involved. Not too bad, huh?

However, if you tried to end the job using other methods, the listener may end without stopping the server instances. We should really be checking the SHTDN op-code, and ending all of the instances whenever the ENDJOB, ENDSBS, ENDSYS or PWRDWNSYS commands are run.

To do this, we need to call select() prior to accept(). Calling select() will allow us to detect when a client connects, but still provide us with a timeout value, so we can check for the system shutting down periodically.

Add these to the bottom of your D-specs:

```
D rc              S             10I 0
D tolen           S             10I 0
D timeout         S               *
D readset         S                   like(fdset)
D excpset         S                   like(fdset)
```

Then, add this code right after the 'alloc calen' line:

```
c                   eval      tolen = %size(timeval)
c                   alloc     tolen         timeout
```

Then, right before calling accept(), add the code to check for shutdown. It should look something like this:

```
c                   dou       rc > 0

c                   callp     FD_ZERO(readset)
c                   callp     FD_ZERO(excpset)
c                   callp     FD_SET(svr: readset)
c                   callp     FD_SET(svr: excpset)
c                   eval      p_timeval = timeout
c                   eval      tv_sec = 20
c                   eval      tv_usec = 0

c                   eval      rc = select(svr+1: %addr(readset):
c                                   *NULL: %addr(excpset): timeout)

c                   shtdn                                        99
c                   if        *in99 = *on
c                   callp     close(svr)
c                   callp     KillEmAll
c                   callp     die('shutdown requested!')
c                   return
c                   endif

c                   enddo
```

Now, when you end the job with *CNTRLD, it'll end all of the server instances as well. We're making progress!

We also didn't implement the 'quit' command that we've used in chapters 5 & 6 to end this server from the client side. If you're looking for a new challenge, you might want to try to implement this with the new, "job spawning" approach. However, we won't do that in this tutorial. (Being realistic, it's unusual that you want the client to be able to end your server!)

Another thing that you may note about this new server is that if you start up many clients rapidly, it doesn't respond as quickly as the examples from chapter 6 did. (Though, if you have a faster AS/400, this may be hard to detect!)

The reason that it's slower is that it takes a bit of time for the system to create a new job, make it active, and have it communicate back to the listener job. The easiest way to improve the performance of this is to "pre-submit" several different server instances, which can be waiting and ready when a client connects.

To implement this, I'm going to add a new named constant, right after the 'MAXCLIENTS' constant. It'll look like this:

```
D PRESTART        C                   CONST(5)
```

Then, after we call the 'NewListener' procedure, we'll insert this code:

```
c                   do        PRESTART
c                   callp(e)  Cmd('SBMJOB CMD(CALL PGM(SVREX6I))' +
c                                       ' JOB(SERVERINST) ' +
c                                       ' JOBQ(QSYSNOMAX) ' +
c                                       ' JOBD(QDFTJOBD) ' +
c                                       ' RTGDTA(QCMDB)': 200)
c                   if        %error
c                   callp     close(svr)
c                   callp     KillEmAll
c                   callp     Die('Unable to submit a new job to ' +
c                             'process clients!')
c                   return
c                   endif
c                   enddo
```

When each of these jobs starts, it'll put it's job info onto the data queue. When the listener program goes to read the data queue, it'll get them in the order they were added. This should cut the delay between the time it takes to accept each new client down to 1/5th the time.

Since the 'KillEmAll' procedure (nice name, huh?) will end any server instance that is currently waiting for a client to service, it'll happily end all of the pre-submitted server instances when NETSTAT, ENDJOB, ENDSBS, ENDSYS or PWRDWNSYS is used to try to shut us down.

Now, we'll begin working towards a new example server program. This program will ask the client for 3 different things: a user-id, a password, and a program to call. It will validate the user-id and password, and then use that user's authority to call the program.

It will pass the program 2 parameters, the socket descriptor that it can use to talk to the client program and the user-id of the validated user.

When the called program has completed, the server instance will close the socket and then end.

The next few topics will explain this in more detail.

7.7. Validating a user-id and password

The Get Profile Handle API (QSYGETPH) will be used to check a user-id and password.

The QSYGETPH API is listed under "Security APIs" in the IBM manuals, and you'll find it at this location: http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/QSYGETPH.htm

The manual page tells us that the parameters to the QSYGETPH API look like this:

Required Parameter Group:			
1	User ID	Input	Char(10)
2	Password	Input	Char(10)
3	Profile handle	Output	Char(12)
Optional parameter:			
4	Error code	I/O	Char(*)

So, our prototype in RPG will look like this:

```
D GetProfile      PR                  ExtPgm('QSYGETPH')
D   UserID                      10A   const
D   Password                    10A   const
D   Handle                      12A
D   ErrorCode                32766A   options(*varsize)
```

Please add that to your JOBINFO_H member.

Or, if you prefer, you could download my copy of JOBINFO_H, available here: http://www.scottklement.com/rpg/socktut/qrpglesrc.jobinfo_h

When we call this API, we will pass it a user-id and password, and it'll either return a "profile handle" in the Handle parameter, or it'll give us an error in the ErrorCode parameter.

We'll use this profile handle for the API that we discuss in the next topic.

To call the GetProfile API, we'll do something like this:

```
c                   callp     GetProfile(UserID: Passwd: Handle:
c                                dsEC)
c                   if        dsECBytesA > 0
C** handle "invalid user profile" error
c                   endif
```

7.8. Running with the user's authority

Once we know that a user-id and password as usable, and we've received a "profile handle" for them, we can use that profile handle to make the job "run under that user profile".

We do that by calling the Set Profile (QWTSETP) API. After this API has been called successfully, everything that new that we try to do will use the authority of the new user-id.

The Set Profile (QWTSETP) API is documented in the IBM manual at this location:
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/QWTSETP.htm

The manual tells us that the API accepts these parameters:

Required Parameter Group:			
1	Profile handle	Input	Char(12)
Optional parameter:			
2	Error code	I/O	Char(*)

The RPG prototype for this API will look like this:

```
D SetProfile      PR                  ExtPgm('QWTSETP')
D   Handle                      12A   const
D   ErrorCode                32766A   options(*varsize: *nopass)
```

And we'll call it like this:

```
c                   callp     GetProfile(UserID: Passwd: Handle:
c                                dsEC)
c                   if        dsECBytesA > 0
C** handle "invalid user profile" error
c                   endif

c                   callp     SetProfile(Handle: dsEC)
c                   if        dsECBytesA > 0
C** handle "could not set profile" error
C                   endif
```

Please add this prototype to your JOBINFO_H member, as well.

Or, if you prefer, you could download my copy of JOBINFO_H, available here:
http://www.scottklement.com/rpg/socktut/qrpglesrc.jobinfo_h

7.9. A "generic server" example

Are you excited about trying another example program? Are you? Are you?

As I mentioned before, this example program differs from the last one, in that it asks for a userid & password, then validates them, them changes it's 'effective user profile' to the user & password that you've signed in as.

Once you're signed in, it asks for a program name, and then it calls that program, passing the socket descriptor and user-id as passwords.

This design is very practical, because by using this server program, you can easily write many different client/server applications without needing to write a separate listener & server instance program for each.

This one involves 3 different programs. The Listener program, which hasn't changed much since our last example -- the only real difference is that the phrase 'SVREX6' has been changed to 'SVREX76' throughout the member. The server instance program, which now validates userid & password, and calls a program. And the ’program to call’, for which I provide one sample program.

In the next topic, we’ll talk about how to run this program, as well as giving a few samples of what you can do with this server.

So... here it is!

```
File: SOCKTUT/QRPGLESRC, Member: SVREX7L

     H DFTACTGRP(*NO) ACTGRP(*NEW)
     H BNDDIR('SOCKTUT/SOCKUTIL') BNDDIR('QC2LE')

      *** header files for calling service programs & APIs

     D/copy socktut/qrpglesrc,socket_h
     D/copy socktut/qrpglesrc,sockutil_h
     D/copy socktut/qrpglesrc,errno_h
     D/copy socktut/qrpglesrc,jobinfo_h

      *** prototypes for external calls

     D Cmd             PR                  ExtPgm('QCMDEXC')
     D   command                    200A   const
     D   length                      15P 5 const

      *** Prototypes for local subprocedures:

     D die             PR
     D   peMsg                      256A   const

     D NewListener     PR            10I 0
     D   pePort                       5U 0 value
     D   peError                    256A

     D KillEmAll       PR

      *** local variables & constants

     D MAXCLIENTS      C                   CONST(256)
     D PRESTART        C                   CONST(5)

     D svr             S             10I 0
     D cli             S             10I 0
     D msg             S            256A
     D err             S             10I 0
     D calen           S             10I 0
     D clientaddr      S               *
     D jilen           S              5P 0
     D rc              S             10I 0
     D tolen           S             10I 0
     D timeout         S               *
     D readset         S                   like(fdset)
     D excpset         S                   like(fdset)
```

```
c                   eval      *inlr = *on

C*************************************************
C* Clean up any previous instances of the dtaq
C*************************************************
c                   callp(e)  Cmd('DLTDTAQ SOCKTUT/SVREX7DQ': 200)
c                   callp(e)  Cmd('CRTDTAQ DTAQ(SOCKTUT/SVREX7DQ) ' +
c                                ' MAXLEN(80) TEXT("Data ' +
c                                ' queue for SVREX7L")': 200)
c                   if        %error
c                   callp     Die('Unable to create data queue!')
c                   return
c                   endif

C*************************************************
C* Start listening for connections on port 4000
C*************************************************
c                   eval      svr = NewListener(4000: msg)
c                   if        svr < 0
c                   callp     die(msg)
c                   return
c                   endif

C*************************************************
C*  Pre-start some server instances
C*************************************************
c                   do        PRESTART
c                   callp(e)  Cmd('SBMJOB CMD(CALL PGM(SVREX7I))' +
c                               ' JOB(SERVERINST) ' +
c                               ' JOBQ(QSYSNOMAX) ' +
c                               ' JOBD(QDFTJOBD) ' +
c                               ' RTGDTA(QCMDB)': 200)
c                   if        %error
c                   callp     close(svr)
c                   callp     KillEmAll
c                   callp     Die('Unable to submit a new job to ' +
c                             'process clients!')
c                   return
c                   endif
c                   enddo

C*************************************************
C* create a space to put client addr struct into
C*************************************************
c                   eval      calen = %size(sockaddr_in)
c                   alloc     calen         clientaddr

c                   eval      tolen = %size(timeval)
c                   alloc     tolen         timeout

c                   dow       1 = 1
```

```
C*************************************
C* Get a new server instance ready
C*************************************
c                   callp(e)  Cmd('SBMJOB CMD(CALL PGM(SVREX7I))' +
c                                       ' JOB(SERVERINST) ' +
c                                       ' JOBQ(QSYSNOMAX) ' +
c                                       ' JOBD(QDFTJOBD) ' +
c                                       ' RTGDTA(QCMDB)': 200)
c                   if        %error
c                   callp     close(svr)
c                   callp     KillEmAll
c                   callp     Die('Unable to submit a new job to ' +
c                             'process clients!')
c                   return
c                   endif

C*************************************
C* Check every 30 seconds for a
C*  system shutdown, until a client
C*  connects.
C*************************************
c                   dou       rc > 0

c                   callp     FD_ZERO(readset)
c                   callp     FD_ZERO(excpset)
c                   callp     FD_SET(svr: readset)
c                   callp     FD_SET(svr: excpset)
c                   eval      p_timeval = timeout
c                   eval      tv_sec = 20
c                   eval      tv_usec = 0

c                   eval      rc = select(svr+1: %addr(readset):
c                                     *NULL: %addr(excpset): timeout)

c                   shtdn                                            99
c                   if        *in99 = *on
c                   callp     close(svr)
c                   callp     KillEmAll
c                   callp     die('shutdown requested!')
c                   return
c                   endif

c                   enddo

C*************************************
C* Accept a new client conn
C*************************************
c                   eval      cli = accept(svr: clientaddr: calen)
c                   if        cli < 0
c                   eval      err = errno
c                   callp     close(svr)
c                   callp     KillEmAll
c                   callp     die('accept(): ' + %str(strerror(err)))
```

```
c                   return
c                   endif

c                   if        calen <> %size(sockaddr_in)
c                   callp     close(cli)
c                   eval      calen = %size(sockaddr_in)
c                   iter
c                   endif

C*************************************
C* get the internal job id of a
C*  server instance to handle client
C*************************************
c                   eval      jilen = %size(dsJobInfo)
c                   callp     RcvDtaQ('SVREX7DQ': 'SOCKTUT': jilen:
c                                     dsJobInfo: 60)
c                   if        jilen < 80
c                   callp     close(cli)
c                   callp     KillEmAll
c                   callp     close(svr)
c                   callp     die('No response from server instance!')
c                   return
c                   endif

C*************************************
C* Pass descriptor to svr instance
C*************************************
c                   if        givedescriptor(cli: %addr(InternalID))<0
c                   eval      err = errno
c                   callp     close(cli)
c                   callp     KillEmAll
c                   callp     close(svr)
c                   callp     Die('givedescriptor(): ' +
c                                 %str(strerror(err)))
c                   Return
c                   endif

c                   callp     close(cli)
c                   enddo


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This ends any server instances that have been started, but
 *   have not been connected with clients.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P KillEmAll       B
D KillEmAll       PI
c                   dou       jilen < 80

c                   eval      jilen = %size(dsJobInfo)
c                   callp     RcvDtaQ('SVREX7DQ': 'SOCKTUT': jilen:
c                                     dsJobInfo: 1)
```

```
c                   if        jilen >= 80

c                   callp(E)  Cmd('ENDJOB JOB(' + %trim(JobNbr) +
c                                  '/' + %trim(JobUser) + '/' +
c                                  %trim(jobName) + ') OPTION(*IMMED)'+
c                                  ' LOGLMT(0)': 200)

C                   endif

c                   enddo
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  Create a new TCP socket that's listening to a port
 *
 *       parms:
 *         pePort = port to listen to
 *        peError = Error message (returned)
 *
 *    returns: socket descriptor upon success, or -1 upon error
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P NewListener     B
D NewListener     PI            10I 0
D   pePort                       5U 0 value
D   peError                    256A

D sock            S             10I 0
D len             S             10I 0
D bindto          S               *
D on              S             10I 0 inz(1)
D linglen         S             10I 0
D ling            S               *

C*** Create a socket
c                   eval      sock = socket(AF_INET:SOCK_STREAM:
c                                           IPPROTO_IP)
c                   if        sock < 0
c                   eval      peError = %str(strerror(errno))
c                   return    -1
c                   endif

C*** Tell socket that we want to be able to re-use the server
C***  port without waiting for the MSL timeout:
c                   callp     setsockopt(sock: SOL_SOCKET:
c                                SO_REUSEADDR: %addr(on): %size(on))

C*** create space for a linger structure
c                   eval      linglen = %size(linger)
c                   alloc     linglen       ling
c                   eval      p_linger = ling

C*** tell socket to only linger for 2 minutes, then discard:
```

```
c                   eval      l_onoff = 1
c                   eval      l_linger = 120
c                   callp     setsockopt(sock: SOL_SOCKET: SO_LINGER:
c                                ling: linglen)

C*** free up resources used by linger structure
c                   dealloc(E)              ling

C*** Create a sockaddr_in structure
c                   eval      len = %size(sockaddr_in)
c                   alloc     len           bindto
c                   eval      p_sockaddr = bindto

c                   eval      sin_family = AF_INET
c                   eval      sin_addr = INADDR_ANY
c                   eval      sin_port = pePort
c                   eval      sin_zero = *ALLx'00'

C*** Bind socket to port
c                   if        bind(sock: bindto: len) < 0
c                   eval      peError = %str(strerror(errno))
c                   callp     close(sock)
c                   dealloc(E)              bindto
c                   return    -1
c                   endif

C*** Listen for a connection
c                   if        listen(sock: MAXCLIENTS) < 0
c                   eval      peError = %str(strerror(errno))
c                   callp     close(sock)
c                   dealloc(E)              bindto
c                   return    -1
c                   endif

C*** Return newly set-up socket:
c                   dealloc(E)              bindto
c                   return    sock
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This ends this program abnormally, and sends back an escape.
 *   message explaining the failure.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P die             B
D die             PI
D   peMsg                      256A   const

D SndPgmMsg       PR                  ExtPgm('QMHSNDPM')
D   MessageID                    7A   Const
D   QualMsgF                    20A   Const
D   MsgData                    256A   Const
D   MsgDtaLen                   10I 0 Const
```

```
D   MsgType                     10A   Const
D   CallStkEnt                  10A   Const
D   CallStkCnt                  10I 0 Const
D   MessageKey                   4A
D   ErrorCode                32766A   options(*varsize)

D dsEC            DS
D  dsECBytesP             1      4I 0 INZ(256)
D  dsECBytesA             5      8I 0 INZ(0)
D  dsECMsgID              9     15
D  dsECReserv            16     16
D  dsECMsgDta            17    256

D wwMsgLen        S             10I 0
D wwTheKey        S              4A

c                   eval      wwMsgLen = %len(%trimr(peMsg))
c                   if        wwMsgLen<1
c                   return
c                   endif

c                   callp     SndPgmMsg('CPF9897': 'QCPFMSG   *LIBL':
c                               peMsg: wwMsgLen: '*ESCAPE':
c                               '*PGMBDY': 1: wwTheKey: dsEC)

c                   return
P                 E

 /define ERRNO_LOAD_PROCEDURE
 /copy socktut/qrpglesrc,errno_h
```

File: SOCKTUT/QRPGLESRC, Member: SVREX7I

```
H DFTACTGRP(*NO) ACTGRP(*NEW)
H BNDDIR('SOCKTUT/SOCKUTIL') BNDDIR('QC2LE')

 *** header files for calling service programs & APIs

D/copy socktut/qrpglesrc,socket_h
D/copy socktut/qrpglesrc,sockutil_h
D/copy socktut/qrpglesrc,errno_h
D/copy socktut/qrpglesrc,jobinfo_h

 *** Prototypes for local subprocedures:

D die             PR
D   peMsg                      256A   const

D GetClient       PR            10I 0

D SignIn          PR            10I 0
D   sock                        10I 0 value
```

```
D   userid                      10A

D cli             S             10I 0
D rc              S             10I 0
D usrprf          S             10A
D pgmname         S             21A

D lower           C                   'abcdefghijklmnopqrstuvwxyz'
D upper           C                   'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

c                   eval      *inlr = *on

C*********************************************************
C* Get socket descriptor from 'listener' program
C*********************************************************
c                   eval      cli = GetClient
c                   if        cli < 0
c                   callp     Die('Failure retrieving client socket '+
c                              'descriptor.')
c                   return
c                   endif

C*********************************************************
C* Ask user to sign in, and set user profile.
C*********************************************************
c                   eval      rc = SignIn(cli: usrprf)
c                   select
c                   when      rc < 0
c                   callp     Die('Client disconnected during sign-in')
c                   callp     close(cli)
c                   return
c                   when      rc = 0
c                   callp     Die('Authorization failure!')
c                   callp     close(cli)
c                   return
c                   endsl

C*********************************************************
C*  Ask for the program to be called
C*********************************************************
c                   callp     WrLine(cli: '102 Please enter the ' +
c                               'program you''d like to call')

c                   if        RdLine(cli: %addr(pgmname): 21: *On) < 0
c                   callp     Die('Error calling RdLine()')
c                   callp     close(cli)
c                   return
c                   endif

c     lower:upper   xlate     pgmname       pgmname

C*********************************************************
C* Call the program, passing the socket desc & profile
```

```
C*   as the parameters.
C*********************************************************
c                   call(e)   PgmName
c                   parm                    cli
c                   parm                    usrprf

c                   if        not %error
c                   callp     WrLine(cli: '103 Call succeeded.')
c                   else
c                   callp     WrLine(cli: '902 Call failed.')
c                   endif

C*********************************************************
C* End.
C*********************************************************
c                   callp     close(cli)
c                   return


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  Sign a user-id into the system
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P SignIn          B
D SignIn          PI            10I 0
D   sock                        10I 0 value
D   userid                      10A

D passwd          S             10A
D handle          S             12A

c                   dou       userid <> *blanks

c                   callp     WrLine(sock: '100 Please enter your ' +
c                               'user-id now!')

c                   if        RdLine(sock: %addr(userid): 10: *On) < 0
c                   return    -1
c                   endif

c     lower:upper   xlate     userid        userid

c                   callp     WrLine(sock: '101 Please enter your ' +
c                               'password now!')

c                   if        RdLine(sock: %addr(passwd): 10: *On) < 0
c                   return    -1
c                   endif

c     lower:upper   xlate     passwd        passwd

c                   callp     GetProfile(userid: passwd: handle: dsEC)
c                   if        dsECBytesA > 0
c                   callp     WrLine(sock: '900 Incorrect userid ' +
```

```
c                                  'or password! ('+%trim(dsECMsgID)+')')
c                    eval      userid = *blanks
c                    endif

c                    enddo

c                    callp     SetProfile(handle: dsEC)
c                    if        dsECBytesA > 0
c                    callp     WrLine(sock: '901 Unable to set ' +
c                              'profile!  ('+%trim(dsECMsgID)+')')
c                    return    0
c                    endif

c                    return    1
P                  E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  Get the new client from the listener application
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P GetClient        B
D GetClient        PI            10I 0

D jilen            S              5P 0
D sock             S             10I 0

c                    callp     RtvJobInf(dsJobI0100: %size(dsJobI0100):
c                                'JOBI0100': '*': *BLANKS: dsEC)
c                    if        dsECBytesA > 0
c                    return    -1
c                    endif

c                    eval      JobName = JobI_JobName
c                    eval      JobUser = JobI_UserID
c                    eval      JobNbr = JobI_JobNbr
c                    eval      InternalID = JobI_IntJob

c                    eval      jilen = %size(dsJobInfo)

c                    callp     SndDtaq('SVREX7DQ': 'SOCKTUT': jilen:
c                                 dsJobInfo)

c                    eval      sock = TakeDescriptor(*NULL)
c                    return    sock
P                  E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This ends this program abnormally, and sends back an escape.
 *   message explaining the failure.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P die              B
D die              PI
```

```
     D   peMsg                      256A   const

     D SndPgmMsg       PR                  ExtPgm('QMHSNDPM')
     D   MessageID                    7A   Const
     D   QualMsgF                    20A   Const
     D   MsgData                    256A   Const
     D   MsgDtaLen                   10I 0 Const
     D   MsgType                     10A   Const
     D   CallStkEnt                  10A   Const
     D   CallStkCnt                  10I 0 Const
     D   MessageKey                   4A
     D   ErrorCode                32766A   options(*varsize)

     D dsEC            DS
     D  dsECBytesP             1      4I 0 INZ(256)
     D  dsECBytesA             5      8I 0 INZ(0)
     D  dsECMsgID              9     15
     D  dsECReserv            16     16
     D  dsECMsgDta            17    256

     D wwMsgLen        S             10I 0
     D wwTheKey        S              4A

     c                   eval      wwMsgLen = %len(%trimr(peMsg))
     c                   if        wwMsgLen<1
     c                   return
     c                   endif

     c                   callp     SndPgmMsg('CPF9897': 'QCPFMSG   *LIBL':
     c                               peMsg: wwMsgLen: '*ESCAPE':
     c                               '*PGMBDY': 1: wwTheKey: dsEC)

     c                   return
     P                 E

      /define ERRNO_LOAD_PROCEDURE
      /copy socktut/qrpglesrc,errno_h
```

File: SOCKTUT/QRPGLESRC, Member: TESTPGM

```
     H DFTACTGRP(*NO) ACTGRP(*NEW)
     H BNDDIR('SOCKTUT/SOCKUTIL') BNDDIR('QC2LE')

      *** header files for calling service programs & APIs

     D/copy socktut/qrpglesrc,socket_h
     D/copy socktut/qrpglesrc,sockutil_h

     D sock            S             10I 0
     D user            S             10A

     c     *entry        plist
```

```
     c                   parm                    sock
     c                   parm                    user

     c                   callp     WrLine(sock: 'Hello ' + %trim(user))
     c                   callp     WrLine(sock: 'Goodbye ' + %trim(user))

     c                   eval      *inlr = *on
```

7.10. Trying the "generic server" out

If you haven't already done so, compile the "generic server" example like this:

```
CRTBNDRPG SVREX7L SRCFILE(SOCKTUT/QRPGLESRC) DBGVIEW(*LIST)
```

```
CRTBNDRPG SVREX7I SRCFILE(SOCKTUT/QRPGLESRC) DBGVIEW(*LIST)
```

```
CRTBNDRPG TESTPGM SRCFILE(SOCKTUT/QRPGLESRC) DBGVIEW(*LIST)
```

Then start it up, like so:

```
SBMJOB CMD(CALL SVREX7L) JOBQ(QSYSNOMAX) JOB(LISTENER)
```

Try it out by telnetting from your PC:

```
telnet as400 4000
```

It'll ask for your user-id, and password. After you've typed both, it'll set your user profile to the one that you've typed it.

> **Note:** Unless SVREX7L was submitted by a user with *ALLOBJ authority, it will only be able to set the user profile to the same profile that it's running under. This is a security safeguard on the AS/400 that prevents programmers from writing programs that would give them more access than they are allowed to have. The solution, of course, would be to run SVREX7L from a job with *ALLOBJ authority, so that it can handle sign-ons for any user.

Once it has set the user profile, it'll ask you for a program to call. You can qualify your program name with a library if you like. To try my demonstration program, you could type SOCKTUT/TESTPGM

You'll notice that each response the server program sends to the telnet client is prefixed by a 3-digit number. This is to make it easy to write a client program that interfaces with the server. The first 3 digits are always a number, so if the client wanted to display the human-readable portion, it could do a simple substring starting in position 5 to remove the message number. The message number can be used to discern exactly what the server is expecting. Plus, any number starting with a '1' is a positive response, any number starting with a '9' is a negative (error) response.

The TESTPGM program will say 'hello' & 'goodbye' just as our other simple server programs have done. Not particularly practical, but it's always good to start simple.

Here's another example of a program you could call. This one asks for the name of a source file, and a member. It then sends back the member's contents. Each source line starts with an extra '.', which a client program would normally remove. This is useful, as it allows us to tell the difference between a line of source code and a server's response message.

File: SOCKTUT/QRPGLESRC, Member: GETSRC

```
H DFTACTGRP(*NO) ACTGRP(*NEW)
H BNDDIR('SOCKTUT/SOCKUTIL') BNDDIR('QC2LE')

FSOURCE    IF   F  252        DISK    USROPN INFDS(dsSrc)

 *** header files for calling service programs & APIs

D/copy socktut/qrpglesrc,socket_h
D/copy socktut/qrpglesrc,sockutil_h

D Cmd             PR                  ExtPgm('QCMDEXC')
D   command                    200A   const
D   length                      15P 5 const

D dsSrc           DS
D   dsSrcRLen           125    126I 0

D lower           C                   'abcdefghijklmnopqrstuvwxyz'
D upper           C                   'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

D sock            S             10I 0
D user            S             10A
D file            S             21A
D mbr             S             10A
D reclen          S              5I 0

ISOURCE    NS
I                                13  252  SrcDta

c     *entry        plist
c                   parm                    sock
c                   parm                    user

c                   eval      *inlr = *on

c                   callp     WrLine(sock: '110 Name of source file?')
c                   if        RdLine(sock: %addr(file): 21: *On) < 0
c                   return
c                   endif

c                   callp     WrLine(sock: '111 Name of member?')
c                   if        RdLine(sock: %addr(mbr): 21: *On) < 0
c                   return
c                   endif

c     lower:upper   xlate     file          file
c     lower:upper   xlate     mbr           mbr

c                   callp(e)  cmd('OVRDBF FILE(SOURCE) ' +
c                                       'TOFILE('+%trim(file)+') ' +
c                                       'MBR(' +%trim(mbr)+ ')': 200)
c                   if        %error
c                   callp     WrLine(sock: '910 Error calling OVRDBF')
```

```
     c                   return
     c                   endif

     c                   open(e)   SOURCE
     c                   if        %error
     c                   callp     WrLine(sock: '911 Unable to open file!')
     c                   callp     Cmd('DLTOVR FILE(SOURCE)': 200)
     c                   return
     c                   endif

     c                   eval      reclen = dsSrcRLen - 12

     c                   read      SOURCE
     c                   dow       not %eof(SOURCE)
     c                   if        WrLine(sock:
     c                                    '.' + %subst(SrcDta:1:reclen)) < 0
     c                   leave
     c                   endif
     c                   read      SOURCE
     c                   enddo

     c                   callp     WrLine(sock: '112 Download successful!')

     c                   close     SOURCE
     c                   callp     Cmd('DLTOVR FILE(SOURCE)': 200)
     c                   return
```

Compile it by typing:

```
CRTBNDRPG GETSRC SRCFILE(SOCKTUT/QRPGLESRC) DBGVIEW(*LIST)
```

Now use your PC to telnet to the AS/400's port 4000 again. Type in your user name and pasword. When it asks for the program, type socktut/getsrc. For source file, you might type something like SOCKTUT/QRPGLESRC for source member, maybe type TESTPGM

See how easy it is, using this example server program to write your own TCP servers? Just think, by writing your own program (perhaps somewhat similar to GETSRC) and by writing a GUI client program, you could write a client/server application using native RPG database access!

Whew. I think we've covered server programs well enough for now. Time to try something new...

Chapter 8. The User Datagram Protocol

Written by Scott Klement.

8.1. Overview of using UDP

Like TCP, the User Datagram Protocol (UDP) is a protocol that runs on top of IP. Where TCP is designed to allow you to communicate using "streams" of data, UDP is designed to let you communicate using "datagrams."

As you may recall from the introduction at the start of this tutorial, all IP information is transmitted in "datagrams". So why is there a UDP? What is the difference between an "IP datagram" and a "UDP datagram"?

What's the difference? What's the difference? Not much.

If you really want to know (you were quite insistent, after all) here's the difference:

```
struct udphdr {            /* UDP header        */

    u_short     uh_sport;  /* source port       */
    u_short     uh_dport;  /* destination port  */
    short       uh_ulen;   /* UDP length        */
    u_short     uh_sum;    /* UDP checksum      */

};
```

That's it. Those 8 bytes are the only difference between UDP and IP datagrams. ("short" is 5I 0 in RPG, and "u_short" is 5U 0 in RPG... so these are all "short integers") When these 4 items are placed in the first 8 bytes of an IP datagram, it becomes a UDP datagram!

What these four integers provide is a "source port", "destination port", and some safeguards to ensure that the data is the same length and content on the receiving side as it was on the sending side.

These "source" and "destination" ports are really very significant. They allow different applications to call bind() and therefore these applications can be addressed individually. They're a method of differentiating between different applications on the same computer.

In other words, if these ports weren't specified, only one program on the system could work with them -- the operating system itself -- because there would be no way to differentiate each program's datagrams.

And this is why it's called the "User" datagram protocol. It allows "user" applications to send and receive datagrams, as opposed to system processes.

Unlike TCP, UDP does not ensure that data was received at the other end of the connection. When many datagrams are sent, UDP does not ensure that they are received in the same order that they were sent like TCP does.

So why would you ever use UDP?

Well, first of all, it's faster than TCP. The "safeguards" and "datagram sequencing" functions of TCP do add some overhead to the protocol. UDP does not have that overhead, and therefore UDP is a bit faster.

Secondly, you don't always WANT the data that gets "lost". An example of this might be a radio broadcast over the internet. If one datagram in the middle of a stream of audio data gets lost, it might be better to just skip the lost data

-- much like receiving static on your traditional radio, rather than lose time in the broadcast. Real Audio is an application that uses this feature.

Third, you don't always need to send more than a single datagram of data. If the entire conversation consists of only a few bytes of data, there's no need to undergo the additional overhead of negotiating a TCP connection. Just send all the data in one datagram, and have the other side manually send back an acknowledgement.

So, although there are many advantages to using TCP, there are some situations where UDP makes more sense. Nearly all of the internet application protocols are run over TCP. But there are still several that use UDP. Some of the better known ones are:

- Domain Name System ("DNS") (the protocol that translates names, such as 'www.myhost.com' to IP addresses)
- Network Time Protocol ("NTP") (a protocol for synchronizing your system clock over a TCP/IP network)
- Simple Network Management Protocol ("SNMP") (a protocol for managing your network devices)
- Traceroute (a program used to see how packets are being routed across the internet)

By comparison to TCP, very few applications actually use UDP. However, it is still an important protcol that deserves a look in our tutorial!

8.2. Creating a socket for use with UDP

Ya know, there's nothing like a good cheeseburger.

Like TCP, UDP programming requires the use of a socket. You'll need to call the socket() API to get one.

Before we can do that, we'll need to add a new constant to our SOCKET_H member:

```
D SOCK_DGRAM      C                   CONST(2)
```

This tells the socket() API that you want a datagram socket instead of a TCP socket.

Now, we can call the socket() API like this:

```
C                   eval      s = socket(AF_INET:SOCK_DGRAM:IPPROTO_IP)
C                   if        s < 0
C**  error occurred, check errno
C                   endif
```

As you can see, we still tell the socket() API that it's going to use the "INET" (internet) address family. And that it's going to use the "IP" (Internet Protocol) In TCP, we send our data in a stream, but in UDP, we'll use datagrams, thus we tell it "SOCK_DGRAM" instead of "SOCK_STREAM".

Makes a lot more sense than that cheeseburger remark, doesn't it?

8.3. The sendto() API call

Each datagram that you send on a UDP socket is sent "by itself". It's not a part of a larger conversation. There is no 'connection' to make, since there isn't a continuous stream of data. It's "one message at a time."

Therefore, a sending program must send a destination address and port with each and every datagram that gets sent. The send() API doesn't give us a place to put an address or port, so a new API is necessary, sendto().

The sendto() API is found in the IBM manuals at this location:
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/sendto.htm

It tells us that the C-language prototype for sendto() looks like this:

```
int sendto(int socket_descriptor,
            char *buffer,
            int buffer_length,
            int flags,
            struct sockaddr *destination_address,
            int address_length);
```

Nothing terribly different or exciting about this prototype. A procedure called 'sendto', with 6 parameters. An integer, a pointer, an integer, another integer, a pointer and finally an integer. So, lets prototype this in RPG:

```
D sendto          PR            10I 0 ExtProc('sendto')
D   sock_desc                   10I 0 Value
D   buffer                        *   Value
D   buffer_len                  10I 0 Value
D   flags                       10I 0 Value
D   destaddr                      *   Value
D   addrlen                     10I 0 Value
```

The first four parms are identical to those used in the send() API. The only difference is that we've added a sockaddr structure to contain the address & port number of the destination. And, like every other address structure we've done, we'll need to supply a length for it.

Please, add this little gem to your SOCKET_H member.

The sendto() API is called like this:

```
c                   eval      len = %size(sockaddr_in)
c                   alloc     len           toaddr

c                   eval      p_sockaddr = toaddr
c                   eval      sin_family = AF_INET
c                   eval      sin_addr = SOME_IP
c                   eval      sin_port = SOME_PORT
c                   eval      sin_zero = *ALLx'00'

c                   if        sendto(s: %addr(buf): buflen: 0:
c                                  toaddr: len) < 0
C*** error! check errno!
c                   endif
```

8.4. The recvfrom() API call

Just as the sendto() API adds an address and port to the send() API, so does the recvfrom() API add an address and port to the recv() API.

Sometimes in a TCP socket, you don't bother to keep track of where the connection comes from. You don't need to. You don't care. With TCP, the API takes care of making sure your responses get where they need to go.

This is not true with UDP. Each datagram stands alone. You need to read in the address that a datagram came from so that you know who to respond to -- that is, if your application requires responses at all.

So, when you supply an address and length in the recvfrom API, it's to receive the address and port that the datagram originated with. You can use it to reply!

The IBM manual page for the recvfrom() API is found right about here:
http://publib.boulder.ibm.com/pubs/html/as400/v4r5/ic2924/info/apis/recvfr.htm

And it tells us that the C-language prototype for the recvfrom() API looks like this:

```
int recvfrom(int socket_descriptor,
              char *buffer,
              int buffer_length,
              int flags,
              struct sockaddr *from_address,
              int *address_length);
```

Yes, it's just like recv(), except that they've added a socket address structure, and an address length. You'll see that the address length is passed by pointer -- this is because the length depends on what the sending program sends us, just like it did with the accept() API.

The RPG version of recvfrom() looks like this:

```
D recvfrom        PR            10I 0 ExtProc('recvfrom')
D   sock_desc                   10I 0 Value
D   buffer                        *   Value
D   buffer_len                  10I 0 Value
D   flags                       10I 0 Value
D   fromaddr                      *   Value
D   addrlength                  10I 0
```

You'll note that we're passing the address length parameter 'by address' instead of passing a pointer to an integer. This looks exactly the same to the API that receives the parameter, but allows the compiler to do some better validity checking.

Add this to your SOCKET_H member, please.

Here's how we call recvfrom:

```
C                   eval      datalen = recvfrom(s: %addr(buf):
c                                %size(buf): 0: fromaddr: fromlen)
c                   if        datalen < 0
c*** Error occurred, check errno!
c                   endif
```

8.5. Sample UDP server

Time to put try our new found UDP knowledge out with a sample UDP server program. This program will receive datagrams containing a user-id and some text, and then will send that user a message using the SNDMSG command. Once the message has been sent, it will send back a datagram that acknowledges that the message was sent correctly.

So, the pseudocode for our server looks like this:

1. Create a UDP socket
2. Bind the socket to a port. Maybe port 4000, that's worked pretty well so far, yes?
3. Wait for any datagrams to be received. (we can use select() to give us a timeout, so we can check for system shut down)
4. convert the datagram to EBCDIC.
5. substring out a userid & message
6. Call the SNDMSG command to send the message
7. Send back an acknowledgement packet
8. GO back to step 3.

> **Note:** Unlikely TCP, we CANNOT use a simple telnet client to test our server program. We'll have to write a client program to play with our server -- which we'll do in the next topic.

Here's our server:

```
File SOCKTUT/QRPGLESRC, Member: UDPSERVER

     H DFTACTGRP(*NO) ACTGRP(*NEW)
     H BNDDIR('QC2LE') BNDDIR('SOCKTUT/SOCKUTIL')

     D/copy socktut/qrpglesrc,socket_h
     D/copy socktut/qrpglesrc,sockutil_h
     D/copy socktut/qrpglesrc,errno_h

     D Cmd             PR                  ExtPgm('QCMDEXC')
     D   command                    200A   const
     D   length                      15P 5 const

     D translate       PR                  ExtPgm('QDCXLATE')
     D   length                       5P 0 const
     D   data                     32766A   options(*varsize)
     D   table                       10A   const

     D die             PR
     D   peMsg                      256A   const

     D sock            S             10I 0
     D err             S             10I 0
     D bindto          S               *
```

```
D len             S             10I 0
D dtalen          S             10I 0
D msgfrom         S               *
D fromlen         S             10I 0
D timeout         S               *
D tolen           S             10I 0
D readset         S                     like(fdset)
D excpset         S                     like(fdset)
D user            S             10A
D msg             S            150A
D buf             S            256A
D rc              S             10I 0

c                   eval      *inlr = *on

C* Reserve space for the address that we receive a
C*  datagram from.
c                   eval      fromlen = %size(sockaddr_in)
c                   alloc     fromlen       msgfrom

C* Create a timeval structure so we can check for shutdown
C*   every 25 seconds:
c                   eval      tolen = %size(timeval)
c                   alloc     tolen         timeout

c                   eval      p_timeval = timeout
c                   eval      tv_sec = 25
c                   eval      tv_usec = 0

C* Create a socket to do our UDP stuff:
c                   eval      sock = socket(AF_INET: SOCK_DGRAM:
c                                       IPPROTO_IP)
c                   if        sock < 0
c                   callp     die('socket(): '+%str(strerror(errno)))
c                   return
c                   endif

C* Create a sockaddr struct to tell the
C*  bind() API which port to use.
c                   eval      len = %size(sockaddr_in)
c                   alloc     len           bindto

c                   eval      p_sockaddr = bindto
c                   eval      sin_family = AF_INET
c                   eval      sin_addr = INADDR_ANY
c                   eval      sin_port = 4000
c                   eval      sin_zero = *ALLx'00'

C* bind to the port.
c                   if        bind(sock: bindto: len) < 0
c                   eval      err = errno
c                   callp     close(sock)
c                   callp     die('bind(): '+%str(strerror(err)))
```

```
c                   return
c                   endif

c                   dow        1 = 1

C* Use select to determine when data is found
c                   callp     fd_zero(readset)
c                   callp     fd_zero(excpset)
c                   callp     fd_set(sock: readset)
c                   callp     fd_set(sock: excpset)

c                   eval      rc = select(sock+1: %addr(readset):
c                                  *NULL: %addr(excpset): timeout)

C* If shutdown is requested, end program.
c                   shtdn                                        99
c                   if        *in99 = *on
c                   callp     close(sock)
c                   return
c                   endif

C* If select timed out, go back to select()...
c                   if        rc  = 0
c                   iter
c                   endif

C* Receive a datagram:
c                   eval      dtalen = recvfrom(sock: %addr(buf):
c                                  %size(buf): 0: msgfrom: fromlen)

C* Check for errors
c                   if        dtalen < 0
c                   eval      err = errno
c                   callp     close(sock)
c                   callp     die('recvfrom(): '+%str(strerror(err)))
c                   return
c                   endif

C* Skip any invalid messages
c                   if        dtalen < 11
c                   iter
c                   endif

C* Skip any messages from invalid addresses
c                   if        fromlen <> %size(sockaddr_in)
c                   eval      fromlen = %size(sockaddr_in)
c                   iter
c                   endif

C* Translate to EBCDIC
c                   callp     Translate(dtalen: buf: 'QTCPEBC')

c* send message to user
```

```
c                   eval      user = %subst(buf:1: 10)
c                   eval      dtalen = dtalen - 10
c                   eval      msg = %subst(buf:11:dtalen)

c                   callp(e)  Cmd('SNDMSG MSG("' + %trimr(msg) +
c                                '") TOUSR(' + %trim(user) + ')': 200)

c* make an acknowledgement
c                   if        %error
c                   eval      buf = 'failed'
c                   eval      dtalen = 6
c                   else
c                   eval      buf = 'success'
c                   eval      dtalen = 7
c                   endif

c* convert acknowledgement to ASCII
c                   callp     Translate(dtalen: buf: 'QTCPASC')

c* send acknowledgement to ASCII
c                   if        sendto(sock: %addr(buf): dtalen: 0:
c                                   msgfrom: fromlen) < 0
c                   eval      err = errno
c                   callp     close(sock)
c                   callp     die('sendto(): '+%str(strerror(err)))
c                   return
c                   endif

c                   enddo


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This ends this program abnormally, and sends back an escape.
 *   message explaining the failure.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P die             B
D die             PI
D   peMsg                      256A   const

D SndPgmMsg       PR                  ExtPgm('QMHSNDPM')
D   MessageID                    7A   Const
D   QualMsgF                    20A   Const
D   MsgData                    256A   Const
D   MsgDtaLen                   10I 0 Const
D   MsgType                     10A   Const
D   CallStkEnt                  10A   Const
D   CallStkCnt                  10I 0 Const
D   MessageKey                   4A
D   ErrorCode                32766A   options(*varsize)

D dsEC            DS
D  dsECBytesP             1      4I 0 INZ(256)
D  dsECBytesA             5      8I 0 INZ(0)
```

```
D  dsECMsgID               9     15
D  dsECReserv             16     16
D  dsECMsgDta             17    256

D wwMsgLen        S              10I 0
D wwTheKey        S               4A

c                   eval      wwMsgLen = %len(%trimr(peMsg))
c                   if        wwMsgLen<1
c                   return
c                   endif

c                   callp     SndPgmMsg('CPF9897': 'QCPFMSG   *LIBL':
c                               peMsg: wwMsgLen: '*ESCAPE':
c                               '*PGMBDY': 1: wwTheKey: dsEC)

c                   return
P                 E

 /define ERRNO_LOAD_PROCEDURE
 /copy socktut/qrpglesrc,errno_h
```

8.6. Sample UDP client

Here is a sample client program that goes with our UDP server. This also, incidentally, demonstrates the ability of the socket library to enable communications between two different programs on the same computer.

Here's pseudocode for our client:

1. We need 3 pieces of data from the command line. The host to send datagrams to, a user-id, and a message. (Since the CALL command has problems with variables over 32 bytes long, we'll have to create a *cmd object as a front-end)
2. Attempt to treat the hostname as a dotted-decimal IP address, and convert it into a 32-bit address.
3. If it's not a dotted-decimal IP address, attempt to look the host up using DNS by calling the gethostbyname() API.
4. Create a new socket. Use SOCK_DGRAM to tell the API that we want a UDP socket.
5. Create a socket address structure, and assign it the address that we calculated in step 2-3. Also assign port 4000.
6. Format the user-id and message into a buffer. Translate that buffer to ASCII.
7. Send the buffer as a UDP datagram to the address that we assigned in step 5.
8. Wait for an acknowledgement datagram.
9. Convert the acknowledgement data back to EBCDIC.
10. Send back the results as a "completion message".

Here's the source code for the *cmd object that we'll use as a front-end:

File: SOCKTUT/QCMDSRC, Member: UDPCLIENT

```
             CMD        PROMPT('Sample UDP client')

             PARM       KWD(RMTSYS) TYPE(*CHAR) LEN(128) MIN(1) +
                          PROMPT('Remote System')

             PARM       KWD(USERID) TYPE(*CHAR) LEN(10) MIN(1) +
                          PROMPT('UserID to Send Message To')

             PARM       KWD(MSG) TYPE(*CHAR) LEN(150) MIN(1) +
                          PROMPT('Message to Send')
```

Here's the RPG source code for our client:

File: SOCKTUT/QRPGLESRC, Member: UDPCLIENT

```
     H DFTACTGRP(*NO) ACTGRP(*NEW)
     H BNDDIR('QC2LE') BNDDIR('SOCKTUT/SOCKUTIL')

     D/copy socktut/qrpglesrc,socket_h
     D/copy socktut/qrpglesrc,sockutil_h
     D/copy socktut/qrpglesrc,errno_h

     D translate       PR                  ExtPgm('QDCXLATE')
     D   length                       5P 0 const
     D   data                     32766A   options(*varsize)
     D   table                       10A   const

     D compmsg         PR
     D   peMsg                      256A   const

     D die             PR
     D   peMsg                      256A   const

     D sock            S             10I 0
     D err             S             10I 0
     D len             S             10I 0
     D bindto          S               *
     D addr            S             10U 0
     D buf             S            256A
     D buflen          S             10I 0
     D host            S            128A
     D user            S             10A
     D msg             S            150A
     D destlen         S             10I 0
     D destaddr        S               *

     c     *entry        plist
     c                   parm                    host
     c                   parm                    user
     c                   parm                    msg
```

```
c                   eval      *inlr = *on

C* Get the 32-bit network IP address for the host
C*  that was supplied by the user:
c                   eval      addr = inet_addr(%trim(host))
c                   if        addr = INADDR_NONE
c                   eval      p_hostent = gethostbyname(%trim(host))
c                   if        p_hostent = *NULL
c                   callp     die('Unable to find that host!')
c                   return
c                   endif
c                   eval      addr = h_addr
c                   endif

C* Create a UDP socket:
c                   eval      sock = socket(AF_INET: SOCK_DGRAM:
c                                       IPPROTO_IP)
c                   if        sock < 0
c                   callp     die('socket(): '+%str(strerror(errno)))
c                   return
c                   endif

C* Create a socket address struct with destination info
C                   eval      destlen = %size(sockaddr_in)
c                   alloc     destlen       destaddr
c                   eval      p_sockaddr = destaddr

c                   eval      sin_family = AF_INET
c                   eval      sin_addr = addr
c                   eval      sin_port = 4000
c                   eval      sin_zero = *ALLx'00'

C* Create a buffer with the userid & password and
C*  translate it to ASCII
c                   eval      buf = user
c                   eval      %subst(buf:11) = msg
c                   eval      buflen = %len(%trimr(buf))
c                   callp     translate(buflen: buf: 'QTCPASC')

C* Send the datagram
c                   if        sendto(sock: %addr(buf): buflen: 0:
c                               destaddr: destlen) < 0
c                   eval      err = errno
c                   callp     close(sock)
c                   callp     die('sendto(): '+%str(strerror(err)))
c                   return
c                   endif

C* Wait for an ack
c                   eval      len = recvfrom(sock: %addr(buf): 256: 0:
c                               destaddr: destlen)
c                   if        len < 6
```

```
c                   callp     close(sock)
c                   callp     die('error receiving ack!')
c                   return
c                   endif

C* Report status & end
c                   callp     Translate(len: buf: 'QTCPEBC')
c                   callp     compmsg('Message sent.  Server ' +
c                              'responded with: ' + %subst(buf:1:len))

c                   callp     close(sock)
c                   return


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This sends a 'completion message', showing a successful
 *   termination to the program.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P compmsg         B
D compmsg         PI
D   peMsg                      256A   const

D SndPgmMsg       PR                  ExtPgm('QMHSNDPM')
D   MessageID                    7A   Const
D   QualMsgF                    20A   Const
D   MsgData                    256A   Const
D   MsgDtaLen                   10I 0 Const
D   MsgType                     10A   Const
D   CallStkEnt                  10A   Const
D   CallStkCnt                  10I 0 Const
D   MessageKey                   4A
D   ErrorCode                32766A   options(*varsize)

D dsEC            DS
D  dsECBytesP             1      4I 0 INZ(256)
D  dsECBytesA             5      8I 0 INZ(0)
D  dsECMsgID              9     15
D  dsECReserv            16     16
D  dsECMsgDta            17    256

D wwMsgLen        S             10I 0
D wwTheKey        S              4A

c                   eval      wwMsgLen = %len(%trimr(peMsg))
c                   if        wwMsgLen<1
c                   return
c                   endif

c                   callp     SndPgmMsg('CPF9897': 'QCPFMSG   *LIBL':
c                               peMsg: wwMsgLen: '*COMP':
c                               '*PGMBDY': 1: wwTheKey: dsEC)

c                   return
```

```
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This ends this program abnormally, and sends back an escape.
 *   message explaining the failure.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P die             B
D die             PI
D   peMsg                      256A   const

D SndPgmMsg       PR                  ExtPgm('QMHSNDPM')
D   MessageID                    7A   Const
D   QualMsgF                    20A   Const
D   MsgData                    256A   Const
D   MsgDtaLen                   10I 0 Const
D   MsgType                     10A   Const
D   CallStkEnt                  10A   Const
D   CallStkCnt                  10I 0 Const
D   MessageKey                   4A
D   ErrorCode                32766A   options(*varsize)

D dsEC            DS
D  dsECBytesP             1      4I 0 INZ(256)
D  dsECBytesA             5      8I 0 INZ(0)
D  dsECMsgID              9     15
D  dsECReserv            16     16
D  dsECMsgDta            17    256

D wwMsgLen        S             10I 0
D wwTheKey        S              4A

c                   eval      wwMsgLen = %len(%trimr(peMsg))
c                   if        wwMsgLen<1
c                   return
c                   endif

c                   callp     SndPgmMsg('CPF9897': 'QCPFMSG   *LIBL':
c                               peMsg: wwMsgLen: '*ESCAPE':
c                               '*PGMBDY': 1: wwTheKey: dsEC)

c                   return
P                 E

 /define ERRNO_LOAD_PROCEDURE
 /copy socktut/qrpglesrc,errno_h
```

8.7. Trying it out

In order to try out our sample UDP programs, we’ll first have to compile them. Let’s build the server first:

```
CRTBNDRPG UDPSERVER SRCFILE(SOCKTUT/QRPGLESRC) DBGVIEW(*LIST)
```

Now the client:

```
CRTBNDRPG UDPCLIENT SRCFILE(SOCKTUT/QRPGLESRC) DBGVIEW(*LIST)
```

```
CRTCMD CMD(UDPCLIENT) PGM(SOCKTUT/UDPCLIENT) SRCFILE(SOCKTUT/QCMDSRC)
```

Start the server by typing this:

```
SBMJOB CMD(CALL UDPSERVER) JOBQ(QSYSNOMAX) JOB(UDP)
```

And then you can try out the client, like this:

```
UDPCLIENT RMTSYS('127.0.0.1') USERID(klemscot) MSG('this is a fine test!')
```

At first glance, this program doesn’t seem very practical. After all, you could’ve done the same thing with the SNDMSG command. However, if you now go to another AS/400, and put our newly built UDPCLIENT on that machine, you can use it to send messages over the network! or even over the internet!

Likewise, if you wrote a message client (similar to UDPCLIENT) to run on your PC, you could send messages from there.

Of course, this is still not very practical :) It is, however, a simple example of how to use UDP datagrams. If you have ideas for a better UDP program, I certainly encourage you to go ahead and experiment!

This concludes my socket tutorial (for now, anyway) I hope you found it enlightening.

If you’ve found any mistakes in the tutorial, or anything that you feel is not explained as well as it could be, please send me an E-mail. I’d like this tutorial to be the best that it can be -- and your suggestions will help to make that happen!

Colophon

This tutorial was written by Scott Klement as a free service to the AS400 / iSeries RPG community.

It was originally written as a plain ASCII text file, written using the **vi** text editor on a FreeBSD system. Before releasing it to the public, however, I re-formatted it into HTML to match the the theme of the rest of my web page.

After receiving many helpful comments from members of the RPG community, I decided that it would be better to re-format it into SGML, so that it could be transformed into PDF, RTF and HTML formats for both on-line viewing and easy printing. After some research, I decided to use the DocBook DTD. This allows me to transform it into different presentation formats using **Jade**, an open source DSSSL engine. Norm Walsh's DSSSL stylesheets were used with an additional customization layer to provide the presentation instructions for Jade.

All of the software used to re-format this document and transform it into different formats is open-source technology, and having used it for the first time creating this document, I highly recommend it!