QSS Server Developer’s Kit (QSDK) Reference Manual

June 1996

Version 2.0

1000 Marshall Street, Suite C, Redwood City, CA 94063
(415) 306-1600 Fax (415) 365-2706

Copyright

QSS Server Developer’s Kit Reference Manual
Version 2.0
June 1996

Trademarks

Printed in the United States of America.

Table of Contents

QSS Server Developer's Kit (QSDK)

Welcome to QSS Server Developer's Kit (QSDK) from ***Q****uintessential* ***S****chool* ***S****ystems.*

QSDK is a software developer's toolkit that enables you to easily develop client/server applications for your HP 3000 using TCP socket technology.

With the QSS Server Developer's Kit you (a software developer) can:

- Easily create network daemons and servers using the common programming languages found on the HP 3000 (COBOL, Pascal, etc).
- Deploy client/server applications without costly run-time or per-seat charges.
- Take your existing TurboImage applications and integrate them into a client/server architecture.
- Write servers which work seamlessly with client applications written in the popular RAD tools (VB, Delphi).
- Concentrate on developing your application without worrying about the details of network TCP programming.

Intended Audience

This guide is intended for software developers/managers who are developing/managing client/server applications for the HP 3000 using TCP sockets. This guide assumes you are familiar with HP 3000 system operations and software development terms.

This guide has been prepared using COBOL as the calling language.

What You'll Find In This Manual

This manual contains the following information, organized by chapter, to assist you in installing and using the QSS Server Developer's Kit (QSDK):

This page intentionally left blank.

Overview of QSDK

QSS Server Developer's Kit (QSDK) is a toolkit that enables software developers to create network applications for the HP 3000 that will communicate using the industry standard TCP protocol. Theses routines are contained in a native mode object module that can be linked with your HP 3000 applications. In addition to the library for the HP 3000, you get a Windows DLL and Visual Basic source for optional routines you can use when developing client applications for Microsoft Windows.

HP 3000 Series 9xx systems are network ready systems. They typically have a network card (NIC) and software that supports the industry standard TCP/IP protocol. On versions of the operating system (MPE/iX) 5.0 and later the TCP/IP software is bundled with the operating system. On systems prior to version 5.0 you purchase the TCP/IP software from HP.

On the HP 3000 there are two application programming interfaces (API) for software to communicate with the network using TCP/IP. The first is a HP proprietary TCP socket layer called NetIPC and the second is Berkeley Sockets, a standard application layer for TCP/IP socket programming. Application programs use these API routines to communicate with programs running on other systems over the network using TCP sockets.

You can develop server software (daemons/servers) for the HP 3000 with either socket layer API. The client programs that communicate with your server will not be able to detect the difference. Most network software written for the HP 3000 use NetIPC since it has been available for many years. Most recently, with the availability of Berkeley Sockets, some developers have begun to use the Berkeley Sockets API for their network socket layer. However, the Berkeley Socket API is geared for programming in 'c' and is not easily used from other languages.

QSDK is an abstraction of TCP socket programming which uses NetIPC. It provides you, the developer, with a set of powerful subroutines which allow you to access the features of TCP socket programming without needing to know the particulars of the NetIPC API or TCP programming. The daemons and servers you develop will be able to communicate with any client that can communicate using standard TCP sockets.

What's In QSDK?

The following components are included with QSDK:

- This reference manual
- NMOBJ file which contains all the toolkit routines.
- Microsoft Windows DLL which contains encryption/decryption routines for Windows based client software.
- One developer license for dsSocket, a Visual Basic custom control for TCP/IP programming using Windows Sockets.
- Sample source code in COBOL and Visual Basic 3.0 for sample client and server programs.
- Visual Basic functions for translating HP 3000 data formats (binary, COMP-3).

Installing QSDK - HP 3000

QSDK is installed on your HP 3000 from the tape included in your installation package. The QSDK installation is simply a matter of restoring the QSDK object file and sample programs from the tape. The optional DLL, Visual Basic functions and sample client applications are contained on the enclosed diskette.

Follow these instructions for installing QSDK on your HP 3000:

1. **Build the QSDK account**

   Place the enclosed tape in the tape drive and restore the QSDKBLD.PUB.SYS file.

   ```
   :HELLO MANAGER.SYS
   :FILE T;DEV=TAPE
   :FILE SYSLIST=$STDLIST
   :RESTORE *T;QSDKBLD.PUB.SYS;SHOW

   :STREAM QSDKBLD.PUB
   ```

   This job will log on as MANAGER.SYS and will build the QSDK account. No password is included in the JCL. You will be prompted for your MANAGER.SYS password(s). Wait for this job to complete.

2. **Restore QSDK Files**

   Put the tape back on-line, then type this command:

   ```
   :RESTORE *T;@.@.QSDK;KEEP;SHOW
   ```

Installing QSDK - PC

The QSDK disk contains a ZIP file that should be copied to your PC system harddrive and unzipped. The README.TXT file contains the latest information on the PC distribution files. See Chapter 6 for specifics on the PC files released with QSDK.

Using QSDK

QSDK is shipped as a NMOBJ file called QSDK.OBJECT.QSDK. You can link this file with your application object files to make NM program files. For example:

```
:LINK FROM=MYSERVER.OBJ,QSDK.OBJECT.QSDK;TO=MYSERVER.BIN;CAP=IA,BA
```

Super listener and listener processes require PH capability because they create son processes. Server and client processes do not need usually need PH.

There is no sofware limitation with QSDK to preclude using it in a NM XL file. However, the distribution license for QSDK does not allow you to freely distribute the object file to HP 3000 systems that do not have a license for QSDK.

QSDK is written in Pascal/XL and has been extensively tested on the HP 3000 using COBOL II/XL. The parameters are defined so you should be able to call the routines from any HP 3000 compatible language that supports calling by reference (all the parameters are defined to be call by reference).

The source code examples are yours to freely use in developing applications that use QSDK. However, you should not repackage them up and sell them as your own software. Call us if you want to do any private labelling of the samples.

You can determine the version of QSDK in the object file or linked into any NM program file by using the utility QSDKLV.PROGRAM.QSDK. Here is how you run it:

```
:FILE QSDKOBJ = <file>
:RUN QSDKLV.PROGRAM.QSDK
```

The version and license information will be displayed on your terminal.

<file> is any NMOBJ or NMPRG file that contains the QSDK library.

QSDK Reference

This chapter describes the QSDK API and how each routine is used. Example calling sequences are given for COBOL. The following tables list the available QSDK API routines for quick reference.

API Summary - Grouped by Typical Use

Super listener	QSDK_START_LISTENER
Listener	QSDK_CLOSE_SOCKET QSDK_CONN_ACCEPT QSDK_CONN_REJECT QSDK_CREATE_SERVER QSDK_CREATE_SOCKET QSDK_INIT_PROCESS QSDK_LISTEN QSDK_START_SERVER
Server	QSDK_CLOSE QSDL_INIT_PROCESS QSDK_OPEN QSDK_READ QSDK_READTRAN QSDK_WRITE
Miscellaneous	QSDK_CONNECT QSDK_ENCRYPT QSDK_ERRMSG QSDK_FILE_XFR QSDK_GET_SOCKET QSDK_GIVEUP_SOCKET QSDK_IPC_D2B QSDK_KEYCRIPT QSDK_PAUSE QSDK_SOCKET_SPECIAL QSDK_VERSION QSDK_WRITE_MSG

API Summary - Alphabetical

QSDK_CLOSE	Close a connection socket being used by this process.
QSDK_CLOSE_SOCKET	Close a listen socket.
QSDK_CONN_ACCEPT	Accept a deferred connection.
QSDK_CONN_REJECT	Reject a deferred connection.
QSDK_CONNECT	Connects to a remote system and TCP port number.
QSDK_CREATE_SERVER	Create (but not activate) a son process which will be used to communicate over a connection socket to a client.
QSDK_CREATE_SOCKET	Create a listen socket for the specified TCP port number.
QSDK_ENCRYPT	Encrypt text with a user defined key.
QSDK_ERRMSG	Return text descriptions for the error values in the error_status record.
QSDK_FILE_XFR	Opens and transfers a file from the HP 3000 to the client process.
QSDK_GET_SOCKET	Acquire a socket (listen/connection) that was released by another process.
QSDK_GIVEUP_SOCKET	Release a socket (listen/connection) so that another process can acquire it.
QSDK_INIT_PROCESS	Return run parameters and initialize the environment for a process.
QSDK_IPC_D2B	Converts an IP address in dot format to octet bye format.
QSDK_KEYCRYPT	Special encryption routine for encrypting a key for use with QSDK_ENCRYPT.
QSDK_LISTEN	Listen for a connection request.
QSDK_OPEN	Used by a server to get the connection socket which was released by the listener.
QSDK_PAUSE	Pause the calling process a specified number of seconds.
QSDK_READ	Read from a connection socket.

QSDK_READTRAN	Read a formatted transaction from a connection socket.
QSDK_READ_MSG	Read a record from the process message file.
QSDK_SOCKET_SPECIAL	Handle special requests for a listen/connection socket.
QSDK_START_LISTENER	Create and activate a son process which will be a listener.
QSDK_START_SERVER	Activate the server process which was created using QSDK_CREATE_SERVER and release the connection socket so the server process can acquire it (by calling QSDK_OPEN).
QSDK_VERSION	Return version and licensing information for QSDK.
QSDK_WRITE	Write to a connection socket.
QSDK_WRITE_MSG	Write a message to a process message file.

API - Specifications (parameter formats)

What follows are the syntax and parameter definitions for the routines contained within QSDK. All parameters are passed by reference. Integer values are specified as 32 bit values (full word) even when 16 bit values (half word) would suffice. The routines were built this way to avoid confusion over full and half word parameters, since it is easier to remember when they are always the same format. Most of the parameters are simple integers or record buffers. However, some parameters are complex record structures and are defined here to avoid having to define them in every routine.

ERROR-STATUS

An array of three (3) integers that is used to return error code and status information by various QSDK routines.

```
01 ERROR-STATUS.
   03 ERR-CODE                PIC S9(9) COMP.
   03 ERR-LL                  PIC S9(9) COMP.
   03 ERR-LL-ID               PIC S9(9) COMP.
```

ERR-CODE	QSDK returned error code. Zero is returned for successful completion. Any non-zero value indicates an error has occurred.
ERR-LL	The error code returned by the underlying system routine when an error occurs.
ERR-LL-ID	A value which defines the underlying system routine that has returned an error.

OPTIONAL-PARMS

A record structure that contains two sixteen (16) element arrays which are used for passing/returning optional information to QSDK routines.

```
01 OPTIONAL-PARMS.
   03 OPTIONAL-FLAG          PIC X OCCURS 16 TIMES.
   03 OPTIONAL-VAL           PIC S9(9) COMP OCCURS 16 TIMES.
```

OPTIONAL-FLAG	Used to pass single byte flags to QSDK routines for selecting optional subroutine behavior. When numeric values are needed or returned they are located in the OPTIONAL-VAL array using the same index as the OPTIONAL-FLAG parameter.
OPTIONAL-VAL	Used to pass/receive optional numeric data. The values are passed/returned in the same index as the corresponding OPTIONAL-FLAG.

This information will help you decode the documentation which describes how to use the OPTIONAL-PARMS parameter for each QSDK routine:

- Optional parameters are identified as “I” , “O”, or “I/O” to indicate input, output or input/output when described in the API syntax.
- Flag fields are identified as F-## where ## specifies the index (1-16).
- Value fields are identified as V-## where ## specifies the index (1-16).

LISTEN-RETURN

A record structure that is returned by a call to the QSDK_LISTEN routine. It is possible for this routine to return parameters for a new connection or a message received from another process. This record structure accommodates these two options.

```
01 LISTEN-RETURN.
   03 SD-SOCKET                PIC S9(9) COMP.
   03 CLIENT-PORT              PIC S9(9) COMP.
   03 MSG-LEN                  REDEFINES CLIENT-PORT
                               PIC S9(9) COMP.
   03 IP-OR-MSG.
      05 CLIENT-IP-BYTE        PIC X(4).
      05 CLIENT-IP-DOT         PIC X(16).
      05 FILLER                PIC X(236).
```

SD-SOCKET	If non-zero then this is a new connection and this contains the socket descriptor of the connection socket.
CLIENT-PORT	The tcp port number of the client who has been connected.
CLIENT-IP-BYTE	The ip address of the client who has connected in octet format.
CLIENT-IP-DOT	The ip address of the client who has connected in dot format. For example: 192.6.1.10.
MSG-LEN	The length of the received message. This is used when the value of sd-socket is 0.
IP-OR-MSG	The text of the received message. This is used when the value of sd-socket is 0.

API - Specifications (syntax)

QSDK_CLOSE

Closes the connection to the client and optionally purges the process message file associated with the calling process.

Syntax

```
CALL “QSDK_CLOSE” USING   SD-SOCKET,   OPTIONAL-PARMS, ERROR-STATUS.
```

Parameters

SD-SOCKET	**32-bit integer**	**(Input)**
	The socket descriptor which was returned by QSDK_OPEN. This is the connection socket being used to communicate with the client process.	
OPTIONAL-PARMS	**optional-parm record**	**(Input)**
	F-01 M=Purge process message file	
	V-01 FD of process message file	
ERROR-STATUS	**error-status**	**(Output)**
	Returned error status array.	

Discussion
This routine is used by a server process to close the connection to the client. When the socket is closed all pending read data is discarded and the socket is shutdown. Once this call is made you cannot communicate with the client process using the specified socket descriptor.

QSDK_CLOSE_SOCKET

Closes a listen socket. Typically used by a listener or super listener to close the listen socket created by QSDK_CREATE_SOCKET. Do not use this to close a connection socket, use QSDK_CLOSE instead.

Syntax

```
CALL “QSDK_CLOSE_SOCKET” USING     SD-LISTEN,   GRACEFUL-FLAG,
                                   ERR-CODE.
```

Parameters

SD-LISTEN	**32-bit integer**	**(Input)**
	The socket descriptor for the listen socket to close.	
GRACEFUL-FLAG	**32-bit integer**	**(Input)**
	A flag indicating whether to close the socket with a graceful (orderly) shutdown. A value of ‘1' indicates graceful (orderly) shutdown, which is preferred.	
ERR-CODE	**32-bit integer**	**(Output)**
	Returned error code. 0 indicates no error occurred.	

Discussion
This routine is used to close a listen socket. Listener processes should call this to close their listen socket. Server processes should call QSDK_CLOSE to close their connection socket.

QSDK_CONN_ACCEPT

Accepts a connection request that was established as ‘deferred’.

Syntax

```
CALL “QSDK_CONN_ACCEPT” USING    SD-SOCKET, ERROR-STATUS.
```

Parameters

SD-SOCKET	**32-bit integer**	**(Input)**
	The socket descriptor for the connection socket returned by QSDK_LISTEN.	
ERROR-STATUS	**error-status**	**(Output)**
	Returned error status array.	

Discussion

This routine is used to accept a connection that was setup as deferred. When you request a deferred connection you must accept it before you can read/write to it.

QSDK_CONN_REJECT

Rejects a deferred connection request.

Syntax

```
CALL “QSDK_CONN_REJECT” USING   SD-SOCKET, ERROR-STATUS.
```

Parameters

SD-SOCKET	**32-bit integer**	**(Input)**
	The socket descriptor for the connection socket returned by QSDK_LISTEN.	
ERROR-STATUS	**error-status**	**(Output)**
	Returned error status array.	

Discussion

This routine is used to reject a deferred connection request. If you reject a connection request the connection socket is no longer usable.

QSDK_CONNECT

Opens a connection socket to a remote system for a particular TCP port number. This is useful if you wish to develop client software that runs on your HP 3000.

Syntax

```
CALL “QSDK_CONNECT” USING    REMOTE-HOST, TCP-PORT,
                             OPTIONAL-PARMS, SD-SOCKET,
                             ERROR-STATUS.
```

Parameters

REMOTE-HOST **x(50)** **(Input)**
The name of the remote system you wish to communicate with. Leave this blank if this is your local HP 3000. This can be either a properly formatted host name or IP address in dot address format (192.6.1.1 for example).

TCP-PORT **32-bit integer** **(Input)**
The tcp port number which this socket is associated with. This is the port number for which the server executing on the HOST-NAME is using.

OPTIONAL-PARMS **x()** **(Input)**

I	F-01	Y = Set the socket timeout value. The default is 60 seconds.
I	V-01	The amount of time specified in tenths of a second for timeout. A value of 0 disables the socket timer.
I	F-02	Y = Enable no-wait I/O for this socket
	V-02	Not used.

SD-SOCKET **32-bit integer** **(Output)**
The socket descriptor for the connection socket created if the call is successful.

ERROR-STATUS **error-status** **(Output)**
Returned error status array.

Discussion

This routine is used open a connection socket to a server process running on your system (local) or another (remote) system. This is how you can write client programs which run on your HP 3000.

You can connect to servers running on the same HP 3000 system by passing blanks for the remote system name. You must have the LOOP network started with NETCONTROL for this to work properly. Your client IP address will be communicated as 127.0.0.1.

You must have configured a way for your HP 3000 to determine the IP address of the remote system. The following are your choices at the present time:

- Create directory entries for the remote locations in your NMMGR network directory.
- Establish the HOSTS.NET.SYS file which is a database (flat file) of remote system to IP address translations.
- Configure the MPE address resolver (MPE/iX 4.0 or later) which is the file RESLVCNF.NET.SYS. The resolver acts as a DNS client to whatever DNS server you have configured. Unfortunately at this time there are no DNS server products available that run on MPE/iX.

QSDK_CREATE_SERVER

Uses CreateProcess to create a son process which is a server. This process is not activated by this call. It is activated by QSDK_START_SERVER.

Syntax

```
CALL “QSDK_CREATE_SERVER” USING    SERVER-NAME, RUN-PARM,    INFO-LEN,
                                   INFO-STRING, OPTIONAL-PARMS, PIN,
                                   ERROR-STATUS.
```

Parameters

SERVER-NAME **X(50)** **(Input)**
Fully qualified program name of process to create.

RUN-PARM **32-bit integer** **(Input)**
Value to pass to the newly created process. Equivalent to the ‘;parm=’ parameter of the run command.

INFO-LEN **32-bit integer** **(Input)**
Length of the info string parameter.

INFO-STRING **x()** **(Input)**
Text string to pass to the newly created process. Equivalent to the ‘;info=’ parameter of the run command.

OPTIONAL-PARMS **optional-parm record** **(Input)**

I	F-01	M= Create process message file for server process
	V-01	Not used
I	F-02	Priority class for created server process (A,B,C,D,E)
	V-02	Not used

PIN **32-bit integer** **(Output)**
The pin of the newly created process.

ERROR-STATUS **error-status** **(Output)**
Returned error-status array.

Discussion

The created process is not activated by this routine. This is so that you can pre-create any number of servers from your listener and then when you get a connection request from a client you use QSDK_START_SERVER to connect everything up.

QSDK_CREATE_SOCKET

Creates (binds) a listen socket which can be used to listen for connection requests.

Syntax

```
CALL "QSDK_CREATE_SOCKET" USING    SOCKET-NAME, TCP-PORT,
                                   OPTIONAL-PARMS, SD-LISTEN,
                                   ERROR-STATUS.
```

Parameters

SOCKET-NAME **x(16)** **(Input)**
A unique name to give to this listen socket.

TCP-PORT **32-bit integer** **(Input)**
The tcp port number which this socket is associated with. This is the port number for which listen requests will be honored by this listen socket. This must be unique on your system.

OPTIONAL-PARMS **x()** **(Input)**

I	F-01	Y = Set the number of queued (waiting) connections. The default is 7.
I	V-01	The number of queued (waiting) connections.
I	F-02	Y = Set the socket timeout value. The default is 60 seconds.
I	V-02	The amount of time specified in tenths of a second for timeout. A value of 0 disables the socket timer.
I	F-03	Y = Enable no-wait I/O for this socket
	V-03	Not used.

SD-LISTEN **32-bit integer** **(Output)**
The socket descriptor for the listen socket created if the call is successful.

ERROR-STATUS **error-status** **(Output)**
Returned error status array.

Discussion

This routine is used to create a listen socket for receiving connection requests from client processes. A tcp port number is considered a unique resource so there can only be one process that has an open listen socket for a given tcp port number.

No-wait I/O is required if want to use these advanced options:

- Listens which return immediately if there is no connection request (IODONTWAIT).
- Listen for a connection and/or a message file read at the same time.

QSDK_ENCRYPT

Encrypts/decrypts a text string using a supplied key. The encryption is done in place, overwriting the original text with the encrypted/decrypted result.

Syntax

```
CALL “QSDK_ENCRYPT” USING    TXT-LEN, TXT-BUF, KEY-LEN, KEY-TXT.
```

Parameters

TXT-LEN	**32-bit integer**	**(Input)**
	The length of the text string to encrypt/decrypt.	
TXT-BUF	**x()**	**(Input/Output)**
	The text string to encrypt/decrypt.	
KEY-LEN	**32-bit integer**	**(Input)**
	The length of the key text string.	
KEY-TXT	**x()**	**(Input)**
	The key text string to use in the encrypt/decrypt operation.	

Discussion
This routine is used to encrypt/decrypt a text string using a supplied key.

QSDK_ERRMSG

Retrieves the text descriptions for the values returned in ERROR-STATUS.

Syntax

```
CALL “QSDK_ERRMSG” USING     ERROR-STATUS, QSDK-ERR-MSG, LL-ERR-MSG,
                             LL-ID-NAME.
```

Parameters

ERROR-STATUS	**error-status**	**(Input)**
	The error status array.	
QSDK-ERR-MSG	**x(80)**	**(Output)**
	The error message corresponding to the first word of the error status array (ERR-CODE).	
LL-ERR-MSG	**x(80)**	**(Output)**
	The error message corresponding to the second word of the error status array (ERR-LL).	
LL-ID-NAME	**x(80)**	**(Output)**
	The name of the low level routine which generated the error.	

QSDK_FILE_XFR

Opens a file and writes it to the specified socket using MR/NOBUF file processing. Supports fixed and variable length records.

Syntax

```
CALL “QSDK_FILE_XFR” USING   SD-SOCKET, FILE-NAME, MAX-SIZE,
                             OPTIONAL-PARMS, ERROR-STATUS.
```

Parameters

SD-SOCKET **32-bit integer** **(Input)**
The socket descriptor for the connection socket connected to your client.

FILE-NAME **x(50)** **(Input)**
The name of the file to transfer. The name must be formatted as required by HPFOPEN which means the first and last character are delimiters and should be the same. For example: &myfile& is a correctly formatted name.

MAX-SIZE **32-bit integer** **(Input)**
The maximum size for write operations to the socket. This is not used as an absolute maximum as records are sent in their entirety and don’t split blocks. The actual bytes sent my exceed this value. 4096 is the largest value you can specify.

OPTIONAL-PARMS **x()** **(Input)**

I	F-01	R= Record mode. A separate socket write operation is performed for each record. B= Block mode. A block of records are send in one socket write operation.
I	V-01	Not used.
I	F-02	Y= Enable handshaking with the client.
	V-02	Number of seconds to wait for client acknowledgements.

ERROR-STATUS **error-status** **(Output)**
Returned error status array.

Discussion

This routine is used send a file from the HP 3000 to the client process. The data stream is specifically formatted for easy parsing. The file is opened MR/NOBUF and read in 8K (8192) byte chunks for fast operation. Fixed and variable length MPE ascii/binary files can be processed with this routine. The handshaking protocol is as follows:

1. QSDK_FILE_XFR sends a five (5) byte message “READY” when it is ready to begin the file transfer. The client should wait and not begin sending until receiving this message.

2. QSDK_FILE_XFR waits for a two byte message of “OK” prior to sending any data records.

For example:

Server (QSDK_FILE_XFR)	**Client**
Sends “READY”	
	Receives “READY”
	Sends “OK”
Receives “OK”	
Sends data	
	Receives data
	Sends “OK”

The format of transmitted data is as follows:

Len	Data	Len	Data	Len	Data

Len is a 16 bit integer containing the number of data bytes. This is stored as two bytes in the data stream. **Data** is the data and is **Len** number of bytes in size. The file is terminated by a **Len** of zero. Even though QSDK_FILE_XFR sends complete records in each transmission you cannot guarantee they will be received the same way. TCP transmissions are guaranteed to arrive in the same sequence as sent, but not in the same sizes. You must code for this on the receiving end. See the sample QTRANC for sample logic which correctly decodes the transmission.

QSDK_GET_SOCKET

Retrieves a listen or connection socket by the referenced name. This routine is used by QSDK_OPEN. It is available for use when custom listener/server applications need to be created.

Syntax

```
CALL “QSDK_GET_SOCKET” USING    SOCKET-NAME, SD-SOCKET, ERR-CODE.
```

Parameters

SOCKET-NAME	**x(16)**	**(Input)**
	The name of the socket which you wish to acquire.	
SD-SOCKET	**32-bit integer**	**(Output)**
	The socket descriptor for the socket acquired by this call.	
ERR-CODE	**32-bit integer**	**(Output)**
	Returned error code. 0 indicates no error occurred.	

Discussion

This routine is used to acquire a listen or connection socket that has been given up by another process. This is a special purpose routine that can be used to create custom applications that pass the socket from one process to another.

Socket names must be unique so its a good idea to always embed the PIN of a known process into the name to avoid conflicts.

QSDK_GIVEUP_SOCKET

Relinquishes a listen or connection socket by the referenced name. This routine is used by QSDK_START_SERVER. It is available for use when custom listener/server applications need to be created.

Syntax

```
CALL “QSDK_GIVEUP_SOCKET” USING    SD-SOCKET, SOCKET-NAME, ERR-CODE.
```

Parameters

SD-SOCKET	**32-bit integer**	**(Input)**
	The socket descriptor for the socket (listen or connection) which should be returned to MPE.	
SOCKET-NAME	**x(16)**	**(Input)**
	The name which should be given to the socket by which another process can acquire it (using QSDK_GET_SOCKET).	
ERR-CODE	**32-bit integer**	**(Output)**
	Returned error code. 0 indicates no error occurred.	

Discussion

This routine is used to relinquish ownership of a socket. By giving the socket a name that is known to another process you can use this and QSDK_GET_SOCKET to pass sockets from one process to another.

Socket names must be unique so its a good idea to always embed the PIN of a known process into the name to avoid conflicts.

QSDK_INIT_PROCESS

This routine returns the ";info=" and ";parm=" parameters used when this process was created.

Syntax

```
CALL "QSDK_INIT_PROCESS" USING      INFO-LEN, OPTIONAL-PARMS,
                                    INFO-STRING,  RUN-PARM, MY-PIN,
                                    DAD-PIN.
```

Parameters

INFO-LEN — **32-bit integer** **(Input)**
Length of the info string parameter.

OPTIONAL-PARMS — **optional-parm record** **(Input/Output)**

I	F-01	M= Open the process message file already created for this process.
O	V-01	Returned FD of the process message file. If the file was not found then this will be zero.
I	F-02	M= Open father's process message file.
O	V-02	Returned FD of your father process message file. If the file was not found then this will be zero.

INFO-STRING — **x()** **(Output)**
The ";info=" string used when this process was created. This buffer must be as large as INFO-LEN.

RUN-PARM — **32-bit integer** **(Output)**
Value used in the ";parm=" parameter when this process was created.

MY-PIN — **32-bit integer** **(Output)**
Your PIN.

DAD-PIN — **32-bit integer** **(Output)**
The PIN of your father. If you are a server then your father is the listener and if you are a listener then your father is either a super listener or your session/job main process (JSMAIN).

Discussion

You can use this in lieu of calling the system intrinsics (JOBINFO) to get the ";info=" and ";parm=" parameter values. In addition, this routine can open the process message files which you can use to communicate with other processes.

QSDK_IPC_D2B

Converts an IP address from dot notation (192.6.1.1) to byte and decimal format. Can also be used to validate the format of an IP address.

Syntax

```
CALL “QSDK_IPC_D2B” USING    IP-ADDR, IP-BYTE-ADDR, IP-DEC-ADDR,
                             ERR-CODE.
```

Parameters

IP-ADDR — **x(16)** **(Input)**
The IP address in display format. For example: 192.6.1.1

IP-BYTE-ADDR — **x(4)** **(Output)**
The IP address in byte octed format. This is the format used by NetIPC routines and is a useful format for doing IP address range checking.

IP-DEC-ADDR — **16-bit integer X 4** **(Output)**
A table of 4 16-bit integers which contains the octet value in a format that can be easily manipulated with COBOL.

ERROR-CODE — **32-bit integer** **(Output)**
Error code. 0 = no error detected. The possible errors are:

1	=	IP-ADDR contains invalid character values.
2	=	IP-ADDR is formatted incorrectly.
3	=	IP-ADDR contains invalid characters and is formatted incorrectly.
-1	=	One or more octet values is invalid. The invalid octet value is identified by the corresponding IP-DEC-ADDR value being set to -1.

QSDK_KEYCRYPT

This routine is used to encrypt a key using a common seed (client IP address).

Syntax

```
CALL “QSDK_KEYCRYPT” USING   KEY-LEN,  KEY-TXT, IP-ADDR,
                             ENCRYPTED-KEY.
```

Parameters

KEY-LEN — **32-bit integer** **(Input)**
The length of the key text string.

KEY-TXT — **x()** **(Input)**
The key text string to encrypt.

IP-ADDR — **x(16)** **(Input)**
The IP address of the client in dot notation (as in 192.6.1.10).

ENCRYPTED-KEY — **x()** **(Output)**
The encrypted key text string.

Discussion

The QSS.DLL contains a corresponding routine which can be used to encrypt a key string. This routine can then be used to decrypt the string so it can be used in decrypting messages. By using a known seed (client IP address) both client and server can communicate encrypted keys without having to ‘trust” each other with hard coded keys.

QSDK_LISTEN

Listens for an incoming connection from a client for the specified socket.

Syntax

```
CALL "QSDK_LISTEN" USING     SD-LISTEN,  OPTIONAL-PARMS,
                             LISTEN-RESULT, ERROR-STATUS.
```

Parameters

SD-LISTEN **32-bit integer** **(Input)**
Socket descriptor for the listen socket.

OPTIONAL-PARMS **optional-parms** **(Input)**

I	F-01	Y= Override maximum send/rcv size.
I	V-01	Maximum number of bytes for send/rcv operations.
I	F-02	Y= Ignore timeout errors.
I	V-02	The number of timeouts to ignore. A value of 0 means ignore all timeouts.
I	F-02	W= Use iowait (blocking nowait I/O)
	V-02	Not used
I	F-02	D= Use iodontwait (non-blocking nowait I/O)
	V-02	Not used
I	F-02	M= Process a read from the process message file or a connection request using blocking nowait I/O
I	V-02	FD of the process message file to read
I	F-03	Y= Setup the connection as 'deferred'
	V-03	Not used.

LISTEN-RESULT **listen-result record** **(Output)**
Depending upon the results of the routine this will be identifying information for client making the connection or a message from another process. The definition of this structure is defined at the beginning of this chapter.

ERROR-STATUS **error-status** **(Output)**
Returned error status array.

Discussion
This routine is used to listen for a connection request and/or for a message from another process. This operation is usually a blocking operating in that control is not returned to the calling process unless a connection is received, a message is read or a network error occurs.

Some additional issues regarding the use of QSDK_LISTEN:

- F-02 values are mutually exclusive, you can only choose one of the options.
- The F-02 'W' option is not normally used and was put in place for testing only. It does work, even if there is no reason to use it since you can get the same result by using F-02 'Y' with a timeout value of zero (0).
- The F-02 'D' option is used to do a no-wait listen for a connection request. This can be used in more sophisticated listeners to allow a higher degree of parallelism.
- Unless otherwise configured, NetIPC sockets have a built-in timeout of 60 seconds. You can use the F-02 'Y' option to logically ignore these timeouts so you don't have to code for them in your listener.
- In order to use option F-02 'W', 'D', and 'M' you need to enable no-wait I/O for the listen socket when it is created (QSDK_CREATE_SOCKET) or enable no-wait I/O using QSDK_SOCKET_SPECIAL.

QSDK_OPEN

Retrieves the connection socket which has been released by the listener through the use of the QSDK_START_SERVER routine.

Syntax

```
CALL “QSDK_OPEN” USING    NAME-BASE, OPTIONAL-PARMS, SD-SOCKET,
                          MY-PIN,  ERROR-STATUS.
```

Parameters

NAME-BASE **x(8)** **(Input)**
The base name used by the listener when releasing the connection socket. If this is left blank then “SOCKET’ is used.

OPTIONAL-PARMS **optional-parm record** **(Input)**

I	F-01	Y= Configure a timeout value for the connection socket.
I	V-01	Timeout value in tenths of a second. Use zero (0) to disable timeouts for the connection socket.
I	F-02	Y= Configure nowait I/O for the connection socket.
	V-02	Not used.

SD-SOCKET **32-bit integer** **(Output)**
The socket descriptor for the connection socket which can be used to communicate with the client process.

ERROR-STATUS **error-status** **(Output)**
Returned error status array.

Discussion

This routine is used by a server to acquire its connection socket that can be used to communicate with the client process.

QSDK_PAUSE

Pauses the calling process for a specified number of seconds.

Syntax

```
CALL “QSDK_PAUSE” USING   PAUSE-SECONDS.
```

Parameters

PAUSE-SECONDS **32-bit integer** **(Input)**
The number of seconds to pause.

Discussion

This routine was added for languages like COBOL that cannot call the PAUSE intrinsic easily because of a lack of support for real data types.

QSDK_READ

Reads from the connection socket or the process message file.

Syntax

```
CALL “QSDK_READ” USING    SD-SOCKET, READ-LEN, OPTIONAL-PARMS,
                          READ-ACTUAL, READ-BUFFER, ERROR-STATUS.
```

Parameters

SD-SOCKET **32-bit integer** **(Input)**
The socket descriptor for the connection socket for which you wish to read.

READ-LEN **32-bit integer** **(Input)**
The maximum amount of text to read.

OPTIONAL-PARMS **optional-parm record** **(Input/Output)**

I	F-01	Y=Ignore timeouts that occur while waiting for data to read.
I	V-01	The number of timeouts to ignore. Use 0 to ignore all timeouts.
I	F-01	D=Use iodontwait for the read.
	V-01	Not used.
I	F-01	M=Read from the socket or the process message file using nowait I/O.
I	V-01	FD of the process message file.
O	F-02	M=Read was from the message file.
O	V-02	Number of bytes read in the message file. Will be 0 if the read was from SD-SOCKET.

READ-ACTUAL **32-bit integer** **(Output)**
The actual number of bytes read. Will be equal to or less than READ-LEN. Note that for message file reads this value is repeated in V-02.

READ-BUFFER **x()** **(Output)**
Buffer area where read text is placed. This must be as large as READ-LEN or memory corruption can occur.

ERROR-STATUS **error-status** **(Output)**
Returned error status array.

Discussion
This routine is used by your server to read a message from the client. You can also use this to read messages from other process.

Some additional issues concerning the use of QSDK_READ:

- F-01 values are mutually exclusive, only choose one of the options.
- Using F-01 ‘D’ and ‘M’ requires nowait I/O on the connection socket. See QSDK_OPEN or QSDK_SOCKET_SPECIAL to see how to enable nowait I/O on the connection socket.
- The F-02 values are output only and will be used only when you use the F-01 ‘M’ option.
- Unless otherwise configured, NetIPC sockets have a built-in timeout of 60 seconds. You can use the F-1 ‘Y’ option to logically ignore these timeouts so you don’t have to code for them in your server.

QSDK_READTRAN

Reads from the connection socket a formatted message and buffers up any extraneous text until the next call to QSDK_READTRAN.

Syntax

```
CALL “QSDK_READTRAN” USING   SD-SOCKET, TRAN-DEF, OPTIONAL-PARMS,
                             READ-X-LEN, READ-X-BUFFER,
                             READ_LEN, READ-BUFFER, ERROR-STATUS.
```

Parameters

SD-SOCKET — **32-bit integer** **(Input)**
The socket descriptor for the connection socket for which you wish to read.

TRAN-DEF — **x(8)** **(Input)**
Sentinel characters which define the start and end of the transaction. Bytes 1-4 are the leading sentinel characters and bytes 5-8 are the trailing sentinel characters.

OPTIONAL-PARMS — **optional-parm record** **(Input)**

I	F-01	Y=Ignore timeouts that occur while waiting for data to read.
I	V-01	The number of timeouts to ignore. Use 0 to ignore all timeouts.
I	F-02	Y=Strip TRAN-DEF characters when filling READ-BUFFER.
I	V-02	Not-used.
I	F-03	Y=Use specified maximum read size. Default is 2048 and maximum is 2048.
I	V-03	Maximum read size in bytes.

READ-X-LEN — **32-bit integer** **(Input/Output)**
Number of remaining bytes from previous read.

READ-X-BUFFER — **x()** **(Input/Output)**
Buffer area where data read that has not been returned as a complete transaction is stored.

READ-ACTUAL — **32-bit integer** **(Output)**
The actual number of bytes in the transaction. The data is stored in READ-BUFFER.

READ-BUFFER — **x()** **(Output)**
Buffer area where the transaction data is placed.

ERROR-STATUS — **error-status** **(Output)**
Returned error status array.

Discussion

This routine is used by a server to read a formatted message from the client. Refer to the discussion of QSDK_READ for additional information pertaining to reading from sockets and timeouts.

This routine is designed to allow you to issue a read request and not return until a complete transaction has been read over the network. You use the TRAN-DEF field to define how a transaction is identified. You must be careful to pick characters that will not occur in your data stream or unpredictable results can occur.

Depending on system/network load and the way in which the client sends data this routine may read data which contains more than one defined transaction. In this case the size (in bytes) of the extra data is stored in READ-X-LEN and the data itself is stored in READ-X-BUFFER.

Before calling this routine for the first time you should make sure that READ-X-LEN is set to zero and that READ-X-BUFFER is initialized to spaces.

To avoid memory corruption make sure that READ-X-BUFFER is large enough to handle any expected overruns.

QSDK_READ_MSG

Reads from the process message file.

Syntax

```
CALL “QSDK_READ_MSG” USING   FD-MESSAGE, READ-WAIT, READ-LEN,
                             READ-BUFFER, ERROR-STATUS.
```

Parameters

FD-MESSAGE	**32-bit integer**	**(Input)**
	The fd of the process message file. This should have been opened through a call to QSDK_INIT_PROCESS.	
READ-WAIT	**32-bit integer**	**(Input)**
	The number of seconds to wait for a timed read. Setting this to zero (0) causes an untimed read which will not be satisfied until there is data available to satisfy the read.	
READ-LEN	**32-bit integer**	**(Output)**
	The number of bytes read.	
READ-BUFFER	**x()**	**(Output)**
	Buffer area where read text is placed. Process message files return a maximum of 256 bytes.	
ERROR-STATUS	**error-status**	**(Output)**
	Returned error status array.	

Discussion

You can use this routine to read messages written to your process message from other processes (usually the listener if you are a server and the super listener if you are a listener).

QSDK_SOCKET_SPECIAL

Performs special operations on sockets whether they are listen or connection.

Syntax

```
CALL “QSDK_SOCKET_SPECIAL” USING   SD-SOCKET, IN-BUFFER,
                                   OPTIONAL-PARMS, OUT-BUFFER,
                                   ERROR-STATUS.
```

Parameters

SD-SOCKET	**32-bit integer**	**(Input)**
	The socket descriptor for the listen or connection socket for which you wish operate upon.	
IN-BUFFER	**x()**	**(Input)**
	To pass text data to QSDK_SOCKET_SPECIAL. What data and its format is dependent on the OPTIONAL-PARMS values. Currently the only value is the trace file name.	
OPTIONAL-PARMS	**optional-parm record**	**(Input/Output)**

I	F-01	Y=Configure the socket timeout value.
I	V-01	The new timeout value in tenths of seconds. Use 0 to disable timeouts for this socket.
I	F-02	Y=Enable nowait I/O. N=Disable nowait I/O.
	V-02	Not used.
I	F-03	Y=Abort outstanding nowait I/O.
I	V-03	Not used.
I	F-04	T=Enable tracing. F=Enable full tracing. Q=Quite tracing.
I/O	V-04	The length of the trace file name in bytes. Setting this to 0 will have a trace file name assigned for you and its length returned here.

I	F-05	Y=Set trace file size.
I	V-05	The size of the trace file in number of records.
I	F-06	Y=Set data length to trace.
I	V-06	The size of data to trace in bytes.
I	F-07	Y=Return local node name.
O	V-07	The returned length in bytes of the local node name.

OUT-BUFFER **x()** **(Output)**
Buffer area where the local node name and possible the assigned trace file name is returned. If both names are returned the trace file is returned first. Use the V-04 and V-6 values to parse this returned value.

ERROR-STATUS **error-status** **(Output)**
Returned error status array.

Discussion
Use this routine to dynamically configure the sockets you are using and to enable tracing when you are attempting to do troubleshooting on your socket transactions.

QSDK_START_LISTENER

Uses CreateProcess to create and activate a son process which will operate as a listener.

Syntax

```
CALL “QSDK_START_LISTENER” USING   LISTENER-NAME, RUN-PARM,
                                   INFO-LEN, INFO-STRING,
                                   OPTIONAL-PARMS, PIN,
                                   ERROR-STATUS.
```

Parameters

LISTENER-NAME **X(50)** **(Input)**
Fully qualified program name of process to create.

RUN-PARM **32-bit integer** **(Input)**
Value to pass to the newly created process. Equivalent to the ‘;parm=’ parameter of the run command.

INFO-LEN **32-bit integer** **(Input)**
Length of the info string parameter.

INFO-STRING **x()** **(Input)**
Text string to pass to the newly created process. Equivalent to the ‘;info=’ parameter of the run command.

OPTIONAL-PARMS	**optional-parm record**	**(Input)**

I	F-01 V-01	M=Create process message file for server process Not used
I	F-02 V-02	Priority class for created server process (A,B,C,D,E) Not used

PIN	**32-bit integer** The pin of the newly created process.	**(Output)**
ERROR-STATUS	**error-status** Returned error-status array.	**(Output)**

Discussion
This routine is used to create a listener process from a super listener process. Different from QSDK_CREATE_SERVER this routine activates the son process so it begins executing immediately.

QSDK_START_SERVER

Activates a server process after relinquishing the connection socket so the server can acquire it from calling QSDK_OPEN. This is usually called from a listener process.

Syntax

```
CALL “QSDK_START_SERVER” USING     SD-SOCKET, SERVER-PIN,
                                   NAME-BASE,  ERROR-STATUS.
```

Parameters

SD-SOCKET	**32-bit integer** The socket descriptor for the connection socket which will be relinquished to the server process.	**(Input)**
SERVER-PIN	**32-bit integer** The pin of the server to activate. Positive pin values don’t suspend the caller, negative pin values suspend the caller.	**(Input)**
NAME-BASE	**x(8)** The base name used to name the connection socket. This name must be known by the server process acquiring the socket. If this is left blank then “SOCKET’ is used.	**(Input)**
ERROR-STATUS	**error-status** Returned error status array.	**(Output)**

Discussion
This routine is typically called by a listener process to activate the server and pass the connection socket.

The default operation is to activate the process without suspending the calling process because the calling is usually a listener which should be not suspended (or it would be difficult to listen for additional connections!).

To override the default operation and force the calling process to be suspended you should pass the negative of the server pin (pin = -pin).

QSDK_WRITE

Writes to the connection socket.

Syntax

```
CALL “QSDK_WRITE” USING   SD-SOCKET, WRITE-LEN, WRITE-BUFFER,
                          OPTIONAL-PARMS, ERROR-STATUS.
```

Parameters

SD-SOCKET	**32-bit integer** The socket descriptor for the connection socket for which you wish to read.	**(Input)**
WRITE-LEN	**32-bit integer** The number of bytes to write.	**(Input)**
WRITE-BUFFER	**x()** Buffer area where the text to write to the socket is stored.	**(Input)**
OPTIONAL-PARMS	**optional-parm record** No optional parameters are in use at this time.	**(Input)**
ERROR-STATUS	**error-status** Returned error status array.	**(Output)**

Discussion

This routine is used by your server to write a message to the client process.

QSDK_VERSION

Returns version and license holder information.

Syntax

```
CALL “QSDK_VERSION” USING    VERSION-TEXT.
```

Parameters

VERSION-TEXT **x(66)** **(Output)**

A formatted text string with this breakout:

1-18	Version string
19-58	License holder string
59-66	Expiration date in YYYYMMDD format. Fully licensed, non-expiring QSDK modules have a zero expiration date.

QSDK_WRITE_MSG

Writes a message to a process message file belonging to another process.

Syntax

```
CALL “QSDK_WRITE_MSG” USING     FD-MESSAGE, PIN, WRITE-LEN,
                                WRITE-BUFFER, ERROR-STATUS.
```

Parameters

FD-MESSAGE	**32-bit integer**	**(Input)**
	The FD of the process message file. This is usually your father process whose message file FD is returned by the call to QSDK_INIT_PROCESS. If you do not know the FD of the message file, but have the pin of the process you should set this to zero and pass the PIN in the PIN field. The FD will be returned in this field if the message could be opened.	
PIN	**32-bit integer**	**(Input)**
	The pin of the process who you wish to communicate with by writing to its process message file. This field is only used if you pass a zero FD-MESSAGE.	
WRITE-LEN	**32-bit integer**	**(Input)**
	The number of bytes to write read.	
WRITE-BUFFER	**x()**	**(Input)**
	Buffer area where the data to write is located. Process message files can contain a maximum of 256 bytes.	
ERROR-STATUS	**error-status**	**(Output)**
	Returned error status array.	

Discussion
You can use this routine to write messages to another processes message file, even if you don’t already have the file opened.

Error Messages

Here is a list of the error codes (ERR-CODE) that may be returned by the various QSDK routines.

Error	Error Description
0	Successful completion (no error)
1	An invalid parameter has been passed
2	The requested TCP port number is already in use
3	There is no network on this system or the network has not been started
4	There are no more sockets available (check NMMGR configuration)
5	The network is being shutdown
6	Could not name the socket
7	The socket must a type ‘listen’ for this operation
8	The socket timer has expired (the timer has ‘popped’)
9	The remote system has aborted the connection
10	The remote system has closed the connection
11	An error has occurred writing to the socket
12	Could not relinquish the socket
13	Could not activate the process
14	An error occurred closing the socket
15	Could not get the connection socket
16	An invalid timeout value was specified
17	No-wait operation on a socket not configured for no-wait I/O

18	Cannot accept the deferred connection
19	Cannot reject the deferred connection
20	No connection, or error attempting to connect to remote server
21	Fatal error starting listener process
22	Fatal error creating server process
23	Cannot create process message file
24	Cannot open process message file
25	IOWAIT on process message file has failed
26	PIN and FD cannot both be zero
27	The process message file is full
28	An error has occurred writing to the process message file
29	The supplied remote host name is invalid
30	Remote host not found
31	Remote host is unreachable (failure during QSDK_CONNECT)
32	Could not open the file you asked to transfer (QSDK_FILE_XFR)
33	Did not receive an acknowledgement within the specified time
34	Improperly formatted IP address
98	The QSDK demo copy has expired
99	Unknown error

This page intentionally left blank.

QSDK and Socket Primer

This chapter contains definitions and descriptions of the various concepts and topics which you will encounter while working with QSDK. This is not meant to be an exhaustive treatise on sockets programming for the HP 30000, for which there is little public documentation. The majority of information published on socket programming is targeted for UNIX or Windows platforms. However, due to the fact that TCP is an Internet standard there is a lot of similarity between platforms.

A Little Background

The entire reason for using TCP socket programming is so that you can get two processes running on different systems to communicate with each other using a standard that is dependable and reliable. When you use TCP sockets for this communication you are using a network (LAN/WAN) since TCP is a protocol that is used to transfer information over a network. And thanks to the clever technical folks there is now the ability to run TCP over serial connections so you can do this same communication using modems (SLIP/PPP).

When you have two programs talking to each other like this you have a general case of client/server computing. One program acts as the client, asking for information and the other program, acting as the server, responds with the requested information.

A Brief Socket Overview

A socket is a communication channel that can be used for two processes to talk to each other. This communication is private between the client and server process such that you can view a socket as a file or pipe that connects the two processes together. A socket is actually a layer of abstraction applied to TCP which is a standard protocol for moving packets of information around networks. You have two choices on the HP 3000 for socket programming interfaces. NetIPC, the HP proprietary format or BSD Sockets, a more universal standard. Because both these socket layers are using TCP as the underlying protocol they appear identical to other processes attempting to communicate with them. QSDK is an abstraction of NetIPC so it embodies those concepts in lieu of BSD Sockets concepts. All references to sockets use the NetIPC terminology which is more in line with MPE terminology than the BSD concepts which are more UNIX like.

Glossary of Basic Terms

Client Process
The unique instance of a program which is communicating with a server process. Client processes request the connection to the server process using the TCP port number and host system address (by name or IP address).

Connection Socket
This is a unique socket which is used for the client and server process to communicate with each other. Once a connection between client and server is established a connection socket is established that both processes can use.

Host System
The system where the server process is executing. This can be equated to the HP 3000 system where your listener and server programs are executing.

Host System Address
A client process that wants to connect to a server running on a specific system must identify this system by name or IP address. Host system names can be resolved by either DNS (or other similar name resolution like WINS) or by using a HOSTS file which contains a name to IP address resolution table.

Listen Socket
This is a special class of socket that is used to facilitate the connection between two processes. A listen socket is bound (or created) for a particular TCP port number and is used to coordinate connection requests from all clients wishing to connect to a server. On your system you can have only one listen socket for any given TCP port number.

Listener Process
A program which is waiting for a connection request for a particular socket (and therefore TCP port number) by calling QSDK_LISTEN. A successful call to QSDK_LISTEN returns a connection socket descriptor which can then be used to communicate with the client process. QSDK includes a standard listener program called QLISTEN which calls QSDK_START_SERVER to startup the server process and pass along this socket descriptor. After starting up the server program then QLISTEN loops back and listens for the next client process connection.

Server Process

The unique instance of a program which will be communicating with a client process using QSDK routines. Servers are typically created as son processes to a listener (QLISTEN for example) and there is one server process for each client communicating over the TCP socket interface. Server processes call QSDK_READ and QSDK_WRITE to read and write the socket for communication with the client process.

TCP Port

Each socket must have an address which uniquely identifies it. This identification is handled through the concept of a port number which is a value from 1-32767. Low order port numbers are reserved for Internet standard services and application software should use the higher port numbers (30000+ for example). In essence a TCP port number identifies the service that is being offered by a server.

Concepts and Anecdotal Comments

This section contains a description of different concepts which you should become familiar with to make full use of the QSDK for your socket programming. This information is particular to QSDK, NetIPC and the HP 3000.

Deferred Connection

You can listen for a connection (QSDK_LISTEN) in such a way that the connection is not completed until you accept it. This is a handy way to introduce security into your listener program. After a connection request is satisfied you are returned the IP address of the client who has connected. You could check that IP address against a list of valid address and reject the connection request. See the routines QSDK_CONN_ACCEPT and QSDK_CONN_REJECT to see how to accept or reject a deferred connection.

Process Message File

QSDK contains the logic for you to create and read/write message files which are uniquely identified to each process in the process tree of your session or job. These files are temporary and have a name of MSG<PIN> where PIN is the PIN of the process. Each process message file is opened by one reader, the process who has the same PIN number and any number of writers, the other processes. These files are variable length with a maximum of 256 byte records. The QSDK routines which manage these files only allow 254 bytes of data because the first 2 bytes are a record length. You can use this feature to develop communication between your listener and server processes.

Queued Connections

On a busy system there might be a large number of simultaneous client connection requests that queue up before the listener process can handle them. By default NetIPC allows for seven (7) queued (or waiting) connections. You can override this value when you create your listen socket (QSDK_CREATE_SOCKET).

Socket No-Wait I/O

It is possible to perform no-wait I/O operations on sockets (both listen and connection) without using PM (privileged mode) code. You can use this to create listeners and servers which have a high degree of parallelism because you will get an immediate return if there is no connection (or no data to read).

Socket Timeouts

Unlike BSD sockets, NetIPC socket operations by default will timeout after 60 seconds. This means if you are listening for a connection on a listen socket and no connection activity occurs for 60 seconds then a timeout error will be generated. Unless you program for this error as a soft error then you will be shutting down your listener for no good reason! This also applies to socket reads (QSDK_READ or QSDK_READTRAN).

There are many ways to handle these timeouts, either by direct code or by setting flags in QSDK. For example, these are ways to handle the situation:

- Loop back if you get a timeout error (QSDKERR=8).
- Set the flag to ignore the timeout errors forever by passing a value of zero.
- Use waited no-wait I/O for your socket operations.
- Disable the timeout value for the socket by setting the appropriate flags and values.

Users Are NOT Logged In!

This might be obvious to the reader, but it sometimes escapes even the most astute so it is mentioned here to help clarify the point. When a client process (say a program written in VB on your PC) connects to the HP 3000 using sockets there is no log-in or other user authentication being provided by MPE. This raises the following issues:

- These users don't count against your user count. This doesn't mean HP won't ever figure out a way to do this, but for the time being you get a free ride for these users.

- You cannot depend on MPE log on security to control your user access. If the user has access to the client software he/she can connect to your system. It is up to you to add some form of user authentication to your application for security purposes.

- Each user is sharing the same job/session space that is accessed by your listener/server process tree. These are usually running in a batch job so whatever account it is logged into is where those programs have access to. The big side affect of this is that jcws, variables, and file equations which are job/session global (not process unique) cannot be used if they can/should be different for each client (user) running your application.

  For example, there can only be one jcw named SITE and it can only have one value. This means all copies of your server application will return the same value for this jcw. The same applies to MPE variables and file equations.

- You cannot do a showjob to show who is logged on using your applications! You can however use the HP utility SOCKINFO.NET.SYS and starting in MPE 5.5 there will be a command to show all socket connections.

Verified Data Transmission/No Size Guarantee

TCP specifications (and HP follows these) are such that you can guarantee the information sent between client and server will be received in the same order it was sent. However, you cannot guarantee the data will be received in the same size chunks (very technical term) as it was sent. In a busy network you might need multiple reads to get the complete message. This means you must allow for this in your code or you will be unpleasantly surprised! It is possible to write something that works in your development environment just fine, but when deployed into production doesn't work at all. So, to be safe you should code everything assuming nothing with regard to the data transmission sizes.

QSDK Samples - HP 3000

The QSDK samples for the HP 3000 are stored in these groups of the QSDK account:

SOURCE	COBOL source
OBJECT	NM compiled object
PROGRAM	NM binary files linked with QSDK.OBJECT
JOB	Jcl files for running the samples

The samples, in alphabetical order are:

QCSMPE	Server which does MPE commands and ships the results back to the client (CSMPE Visual Basic client).
QECHO	Server which echos the text it receives to the modified version of the dsSocket sample echo client.
QLISTEN	Generic listener which prompts for parameters and can start up any server.
QUPERL	Sample super listener program which can start up any number of listeners.
QTRANC	Client program which can connect to another HP 3000 running the QTRANS server for sample file transfers.
QTRANS	Server which uses QSDK_FILE_XFR to transfer an MPE file to the client (QTRANC).

You will find sample jobs in the JOB group which run the super listener/listener/server programs.

QSDK Samples - PC

The PC samples are come zipped in one file called **QSDK.ZIP** which can be found on the enclosed diskette or in the file **QSDKZIP.PROGRAM.QSDK** Unzip this file into its own directory on your PC and you will get these files:

VBRUN300.DLL	Visual Basic 3.0 run-time
DSSOCK.VBX	Run-time Winsock VB custom control from Dolphin Systems, Inc.
ECHO.EXE	VB executable for the ECHO client. This is a slightly modified version of Dolphin Systems, Inc. ECHO sample (changed the port number to 32000).
QCCSMPE.EXE	VB executable for the C/S MPE client.
QCCSMPE.ZIP	The VB source modules for QCCSMPE.
QSSDLL.DLL	QSS encryption routines for the PC.
QSDK.BAS	VB source with declares for QSS.DLL and functions for converting HP data into VB equivalents.

You should copy the DLL files into your \WINDOWS\SYSTEM directory and add the EXE files to your Windows desktop. Feel free to explore the source code to learn how to write client software in VB.

System Requirements - HP 3000

QSDK has been written for the HP 3000 series 9xx computer systems. While NetIPC exists for the Classic (MPE/V) HP 3000 systems, these systems are not supported by QSDK.

Hardware Requirements

The HP 3000 that QSDK will execute on must have a Network Interface Card (NIC) which allows it connect to a network. All HP 3000 series 9xx systems come with a pre-installed NIC which can be used for network communications.

Software Requirements

The HP 3000 must have the TCP/IP networking software which can be acquired by either of these two methods:

- Installing any MPE/iX version 5.0 or later has this software bundled with FOS.
- Customers on pre 5.0 versions can purchase the product ThinLan 3000/iX (36923A) which has been reduced in price significantly.

System Requirements - PC

QSDK comes with one developer license for dsSocket from Dolphin Systems, Inc. This Visual Basic custom control (VBX) is designed to be used on a MS Windows PC that has a Winsock compatible TCP/IP stack. The following TCP/IP stacks have been successfully used by QSS and QSDK customers with dsSocket to connect to servers running on the HP 3000:

- Microsoft TCP-32a stack for Windows for Workgroups
- Microsoft Windows-95 TCP/IP stack (including dial-up PPP)
- Trumpet Winsock dialup SLIP/PPP
- WRQ RNS

This page intentionally left blank.

Super Listener/Listener/Server Anatomy

This chapter describes in narrative form what comprises the different types of processes (programs) you may write using QSDK.

The sample programs included with QSDK are a good source of information you might helpful in identifying the QSDK calls and order of operation required.

Super Listener

This program is used to be a root process for all the listeners you wish to operate on your HP 3000 system. Typically it would prompt for a list of listener/server data items and then use QSDK_START_LISTENER to create a son process for each listener process required. Each listener would be listening on a unique TCP port number. A super listener is handy in that it allows all your socket applications to be hosted by one batch job instead of having separate batch jobs for each listener.

Listener

A listener program is a simple program that will listen for incoming connections which are passed off to a server process. Listener programs typically operate in a loop which is only terminated by a fatal networking error. Listener's typically have this basic flow of operation:

```
QSDK_INIT_PROCESS
QSDK_CREATE_SOCKET

Loop
        QSDK_CREATE_SERVER
        QSDK_LISTEN
        QSDK_START_SERVER

Until quit_listening

QSDK_CLOSE_SOCKET
```

Performance is improved if you have pre-created the server process prior to listening for the next incoming connection. That is why QSDK_CREATE_SERVER doesn't activate the newly created process. It gives you a mechanism to pre-create the server and then when QSDK_LISTEN returns a connection you use QSDK_START_SERVER, which activates the server process.

Server

Server programs are the true application component of the socket solution you are writing. QSDK contains generic and freely useable Super Listener and Listener programs which are easily used in any application. However, the server program is unique to your application and is where all the intelligence of your application resides. It might be responding to requests by reading data from a TurboImage database or maybe reading a KSAM file. Whatever processing it does do, the main form of communication to the client process is through the QSDK subroutimes. A typical server program will have this basic flow of execution:

```
QSDK_INIT_PROCESS
QSDK_OPEN

Loop
        QSDK_READ or QSDK_READTRAN

        ...application specific logic

        QSDK_WRITE

Until done_processing

QSDK_CLOSE
```

Process Tree

This is what a typical process tree would look like when using Super Listener/Listener/Server processes:

```
Super Listener

          Listener (port a)

                    Server #1
                        ...
                    Server #n


          Listener (port b)

                    Server #1
                        ...
                    Server #n


          Listener (port c)

                    Server #1
                        ...
                    Server #n
```

This page intentionally left blank.