MQSeries Client *for* Stratus VOS

Stratus Technologies, Inc.
R450-03A

MQSeries

Contents

Contents iii
About This Document ix
What you need to know ix
MQSeries Publications ix
Evaluating products ix
Planning ix
Administration x
Application programming x
Information about MQSeries on the Internet x
Summary of Changes xi
Changes for this edition include: xi
Overview 13
What is an MQI client 13
Why use MQI clients 14
Preparing for VOS Client Installation 15
Support for MQI Clients 15
Client to server connection 15
Communications 17
Hardware and Software Requirements 19
VOS Operating System Requirements 19
General Requirements 19
Network Interface Requirements: 19
Software and Programming Requirements 19

Compilers for MQSeries Client Applications on VOS 19
Installing VOS MQSeries Version 5.2 Clients 21
Steps to installing MQSeries Clients for VOS 21
Building Applications 23
Preparing VOS GNU C and C++ Clients 23
C Entry Point main() Required 23
Compiling and binding with VOS GNU C and C++ 23
gcc and g++ Command-Line Arguments and Options 24
Archives and Object Libraries 25
Binding VOS C and GNU C/C++ Code Together 26
VOS Standard C Compared to GNU C 29
Language Differences 29
Calling VOS C from GNU C/C++ and Vice-Versa 30
Runtimes and System Calls 30
Frequently Asked Questions 33
File Formats and gcc 33
VOS Tasking and gcc 33
Listing and Error files from gcc 33
Bindfiles and gcc 34
Relocation Overflow When Binding 34
VOS Debug and GNU Code 34
More Information 34
Compiling and Binding COBOL Clients 37
MQSeries Environment Variables 39
MQSeries environment variables 39

Detailed description of MQSeries environment variables and parameters 40
Setting MQSERVER 40
Setting MQCHLTAB and MQCHLLIB 41
Setting MQCCSID 42
Channel exits 43
Channel exits 43
Configuration 45
Before you define your MQI channels: 45
Deciding which network transport type to use 45
Defining a TCP/IP Connection 46
On the server 46
MQSeries Client Security 47
MQSeries Client Security 47
Access Control 48
Establishing communications 50
What is a channel? 50
Message channel 50
MQI channel 50
Connecting the MQSeries client and server - channel definitions 51
Defining your channels 52
On the Server 53
Reference the *clntconn* definition using MQSERVER 54
On the MQI client 54
Reference the *clntconn* definition using MQCHLTAB and MQCHLLIB 54
On the Client 54

Using the MQI 56
Limiting the maximum message length 56
Choose client or server coded character set identifier (CCSID) 56
Design considerations 56
MQINQ considerations 57
Syncpoint considerations 57
Trigger monitor for the client environment 57
Building Applications for MQI clients 58
MQI client environment 58
Channel Exits 59
MQI client and queue managers 60
Client connection to queue manager 60
Using MQSERVER 60
Using the DEFINE CHANNEL command on the server 60
Role of the client channel definition file 61
Queue manager name specified 62
Queue manager name prefixed with an asterisk (*) 63
Queue manager name is blank or an asterisk (*) 64
Problem Determination 66
MQI Client fails to connect 66
Stopping MQI clients 66
Error messages associated with MQI client activity 67
MQCONN Returns 2058 Error Code: MQRC_Q_MGR_NAME_ERROR 67
MQCONN Returns 2059 Error Code: MQRC_Q_MGR_NOT_AVAILABLE 67
IBM Verification Test Programs 68

Setting Library Paths 68
add_lib.cm 68
Compiling and Running the Test Programs 69
set_var.cm 70
c_compile.cm 70
do_c_get.cm 70
do_c_put.cm 70
big_c_put.cm 71
cobol_compile.cm 71
do_cobol_get.cm 72
do_cobol_put.cm 72
big_cobol_put.cm 72
Notices 74
Index 76

About This Document

This book primarily contains information about the MQSeries client with additional information about the server environment.

More information relating to MQSeries clients can be found in the reference material in the other MQSeries books.

This book is intended for system administrators, for anyone who installs and configures the product, and for application programmers who write programs to make use of the Message Queue Interface (MQI).

What you need to know

You should have:

- Experience in installing and configuring the system you use for the server: This can be AS/400, MQSeries for Compaq (DIGITAL), Open VMS, OS/2 Warp, OS/390, Tandem NonStop Kernel (NSK), several UNIX systems,VSE/ESA or Windows NT.
- Experience with any client platforms that you will be using, especially VOS.
- Understanding of the purpose of the Message Queue Interface (MQI).
- Experience of MQSeries programs in general, or familiarity with the content of the other MQSeries publications.

MQSeries Publications

Evaluating products

IBM WebSphere MQ: An Introduction to Messaging and Queuing, GC33-0805-01

Planning

IBM WebSphere MQ Planning Guide, GC33-1349-08

IBM WebSphere MQ Release Guide Version 5.2, GC34-5761-01

Administration

IBM WebSphere MQ System Administration SC33-1873-02

IBM WebSphere MQ Series Clients, GC33-1632-09

IBM WebSphere MQ Series Command Reference, SC33-1369-13

IBM WebSphere MQ Programmable System Management, SC33-1482-08

IBM WebSphere MQ Messages, GC33-1876-02

Application programming

IBM WebSphere MQ Application Programming Guide, SC33-0807-12

IBM WebSphere MQ Application Programming Reference, SC33-1673-08

IBM WebSphere MQ Application Programming Reference Summary, SX33-6095-07

IBM WebSphere MQ Intercommunication, SC33-1872-05

IBM WebSphere MQ Queue Manager Clusters, SC34-5349-03

IBM WebSphere MQ Using C++, SC33-1877-05

Information about MQSeries on the Internet

http://www-306.ibm.com/software/integration/wmq/

http://www.stratus.com

Almost all of the referenced IBM manuals may be found and read or downloaded at the IBM URL above.

Summary of Changes

Original document produced January 1998.

This document has been written specifically for MQSeries Client on VOS. It is based on *MQSeries Client* GC33-1632-09 Tenth Edition (2000).

Revision 01 – Updates for MQSeries for VOS V1.1. support.

Revision 02 – Updates for MQSeries for VOS V5.2 support.

Revision 03 – Adds information for VOS Release 15.0 and on compiling and running IBM verification test programs.

Revision 03A – Adds a note about the requirement for a C main() entry point in applications.

Changes for this edition include:

- VOS now supports the POSIX standard and MQSeries 5.2 was also written to conform to that standard. This means that the upgrade to 5.2 requires that the customer must order and install the GNU Tools product.
- If using TCP/IP as a transport protocol, the VOS Streams TCP/IP product (S115) must be installed and configured. Neither the STCP compatibility library nor the OS TCP product is supported by MQSeries 5.2.
- VOS now has true environment variables, eliminating the emulation that was used in VOS client 1.1
- VOS client 5.2 now supports the security, read and write exits.
- VOS client 5.2 supports the new MQI request MQCONNX that allows runtime selection of the channel to be connected.
- Features supported by some MQSeries servers but not by the VOS client are threads, DCE, Java, and Active Directories.
- In the manual, there are many references to the VOS GNU C (gcc) compiler. If your VOS release does not include this compiler, you can use the VOS vcc compiler. The vcc compiler is available in VOS Release 15.0.

Directory	File Examples
bin	dspmqtrc.pm, runmqtmc, etc.
inc	Approximately 100 include files
lib	libmqic.a, libmqicb.a, libmqmcs.a, libsnastubs.a, etc.
nls	Contains messages
nls>c	amq.cat
nls>C>LC_MESSAGES	amq.cat
samp	Collection of sample programs
errors	Contains Log and error files

Directories installed under (master_disk)>system>mqseries2

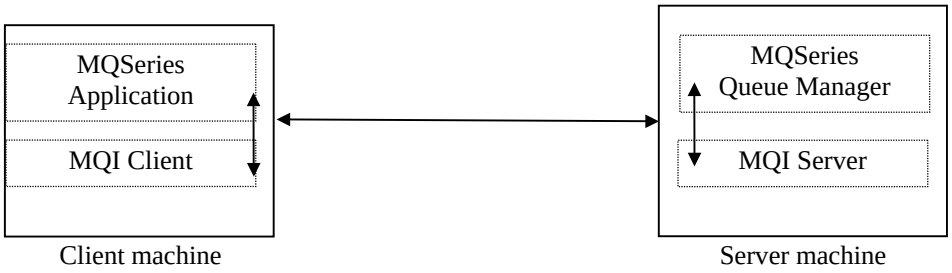
Overview

What is an MQSeries client? What are the benefits of using MQSeries clients? These questions are answered here.

What is an MQI client

An MQSeries client is a component of the MQSeries product that can be installed on its own, on a separate machine from the Base product and Server. You can run an MQSeries application on an MQSeries client and it can interact, by means of a communications protocol, with one or more MQSeries servers and can connect to their queue managers.

There are multiple platforms that can be used in one or both roles (the combinations depend on which MQSeries server product you are using). A list of the available server platforms should be considered when planning projects.



An application that you want to run in the MQSeries client environment must first be linked with the relevant client library. When the application issues an MQI call, the MQSeries client code directs the request to a queue manager where it is processed and from where a reply is sent back to the MQSeries client.

The link between the application and the MQSeries client code is established dynamically at runtime, except in the case of DOS, when it is a static link.

Why use MQI clients

Using MQSeries clients is an efficient way of implementing MQSeries messaging and queuing.

You can have an application that uses the MQI running on one machine and the queue manager running on a different machine, either physical or virtual. The benefits of doing this are:

- There is no need for a full MQSeries implementation on the client machine.
- Hardware requirements on the client system are reduced.
- System administration requirements are reduced.
- An MQSeries application running on a client can connect to multiple queue managers on different systems.

Preparing for VOS Client Installation

This topic details the VOS platform support for clients, explains the communications protocols used, and shows how MQSeries clients fit into your network.

Hardware and software requirements for the VOS client platform are given here.

For other client platform hardware and software requirements, see the *MQSeries Clients* document GC33-1632.

For your server platform hardware and software requirements, see the *WebSphere MQ v5.2 Quick Beginnings manual* for your platform. These manuals are very specific for their platform . VOS and other client implementations do not have such individual manuals at this writing.

Support for MQI Clients

In general, any MQSeries server will support your MQI Client for VOS, subject to coded character set (CCSID) and communications protocol differences.

Note: Make sure that code conversion from the CCSID of your client is supported by the server. See the Language support tables in the *MQSeries Application Programming Reference,* SC33-1673

Stratus Platform	OS	CCSID
Continuum	VOS	819, 850

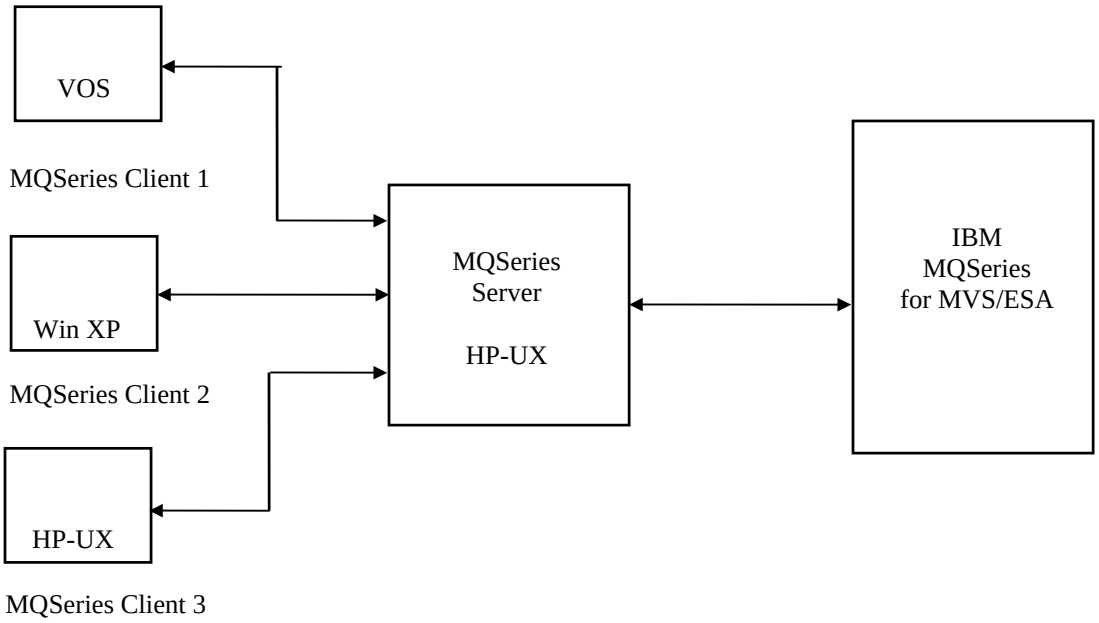
Stratus Platform CCSID Table

Client to server connection

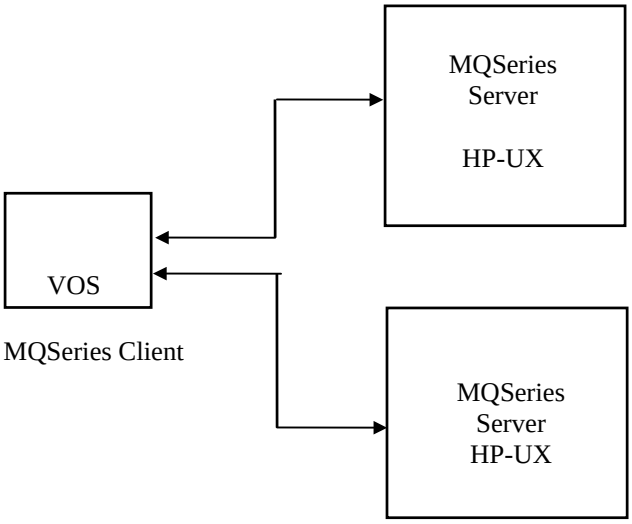
An application running in the MQSeries client environment runs in synchronous mode, as there must be an active connection between the client and server machines.

An application issues an MQCONN or the new MQCONNX call to make a connection. Clients and servers communicate through MQI channels. When the call succeeds, the MQI channel remains connected until the application issues a MQDISC call.

MQSeries Client and Server Connection Examples



Connectivity Example 1



Connectivity Example 2

Communications

MQSeries clients use MQI channels to communicate with the server. A channel definition must be created at both the MQSeries client and server ends of the connection.

An MQSeries application on an MQSeries client can use all the MQI calls in the same way as when the queue manager is local. MQCONN or MQCONNX associates the MQSeries application with the selected queue manager, creating a connection handle. Other calls using that connection handle are then processed by the connected queue manager. This MQSeries client communication is synchronous, in contrast to the communication between queue managers, which is connection and time independent.

The transmission protocol is specified via the channel definition and does not affect the application. The VOS client offers TCP/IP.

Hardware and Software Requirements

VOS Operating System Requirements

MQSeries client for VOS is supported on VOS Release 14.7.0 or greater.

General Requirements

Each machine must be equipped with sufficient hardware, i.e. disk, RAM, and network interface(s), and prerequisite software to meet the application and MQSeries requirements.

Network Interface Requirements:

For TCP/IP connectivity:

Ethernet interface K460, K470, K480, or K104

Software and Programming Requirements

For TCP/IP connectivity:

TCP/IP is supplied as part of the operating system

Compilers for MQSeries Client Applications on VOS

The following compilers are supported:

- VOS cc compiler
- VOS gcc compiler
- VOS g++ compiler
- VOS vcc compiler
- VOS cobol compiler

Installing VOS MQSeries Version 5.2 Clients

The MQSeries Client software for VOS is installed from media supplied by Stratus Computer, Inc.

For the MQSeries Client software to operate properly, MQSeries must be installed and operational on a companion server platform. **The MQSeries server software must be acquired separately from an authorized supplier of IBM software**.

Steps to installing MQSeries Clients for VOS

Before installing the client media, you should create an mqm user_id and an mqm group name. These are default ids that are expected to exist by both server and client

Use the installation instructions provided with the product tape to start the installation process. The installation command install_new_release copies the programs, objects, libraries, messages, samples, etc. into the appropriate system directories. The product directories form a tree whose base is referred to by the term MQROOT, which differs from platform to platform.

MQROOT for VOS is (master_disk)>system>mqseries2.

An overview of the installed tree is outlined on page 8.

MQSeries2 uses an initialization file called "mqs.ini". The IBM manual *MQSeries Clients* (GC33-1632-09) contains more information about the fields in the mqs.ini file. Stratus supplies a simple mqs.ini file which is installed into the directory >system>mqseries2>samp. There must be an mqs.ini file in the directory >system>mqseries2. If there is no mqs.ini file, attempts to connect to the server will result in an error code of 2195 being returned on the connect. You can copy the sample mqs.ini from >system>mqseries2>samp to >system>mqseries2. Additional editing of the mqs.ini file may be needed for your specific installation. Refer to the IBM manual for more information.

Among the installed client components will be sample application program sources. You should build some applications and test them

Configure the communications channel to the server platform, and test by putting messages on the remote queue and recovering them. Details of how to do these tasks will follow.

Building Applications

Preparing VOS GNU C and C++ Clients

GNU C and C++ were originally developed on Unix and have been ported to many Unix and non-Unix platforms. Because the interface retains a Unix look-and-feel it differs from other VOS compilers. Also, the VOS GNU C compiler is not as forgiving as the VOS C compiler. VOS GNU C reports legitimate coding errors in code that compiled without error using VOS Standard C. The following material will give an overview of the ways to use these new compilers and how they differ from the VOS Standard C compiler.

C Entry Point main() Required

Note that the starting point for your client application must be a C entry point called main:

```
main();
```

If your application does not contain such an entry point, the POSIX environment is not initialized correctly, and the client will not work correctly.

Compiling and binding with VOS GNU C and C++

Creating a C program using cc is a two step process: compiling with cc and binding with bind. Creating a C program with gcc can also be two step process, but a single command, gcc, does both. To compile, use the –c option with gcc:

gcc –c *filename*.c

This produces an object file, *filename*.o. Notice that the .c extension is given explicitly on the gcc command line. Unlike cc, the extension is required.

To bind a GNU C program, leave the –c option off:

gcc *filename*.o

This produces an executable program, *filename*.pm.

Compilation and binding can be done in a single gcc invocation:

gcc *filename*.c

which creates *filename*.pm directly from *filename*.c.

g++ is the GNU C++ compiler program, and its functionality is very close to gcc. The two main differences are:

- Given filename.c as input, gcc will assume it is a C source file,while g++ will assume it is a C++ source file;
- g++ automatically includes additional runtime libraries when binding.

Aside from these differences, gcc and g++ are equivalent, and we will use gcc in all cases, but the descriptions apply to g++ as well.

gcc and g++ Command-Line Arguments and Options

The general syntax of the gcc command line is:

gcc argument [argument] . . .

Each argument may be a pathname or an option. The pathnames are input files, and gcc distinguishes types of input files by the *filename* extension :

filename.c *filename*.C	is a C (or C++) source file
filename.cpp *filename*.cxx	are C++ source files
filename.s	is an assembler source file
anything else (including *filename*.o, *filename*.a and *filename*.obj)	is considered an object file.

gcc has literally hundreds of command line options. Here are a few of the more frequently-used ones:

-*ansi*	follow ANSI C standard strictly, disallowing extensions.
-c	preprocess, translate and assemble, but don't bind.
-D*macro* [=*definition*]	

	define a preprocessor macro.
-E	preprocess, but don't translate, assemble or bind.
-g	generate high-level debugging information for use with gdb.
--help	display descriptions of command-line options.
-I*directory*	search *directory* for included files.
-L*directory*	search *directory* for libname.a (see below).
-l*name*	search for file lib*name*.a files (see below).
-o*pathname*	specify output filename.
-O[n]	specify optimization level, where n is 0,1,2 or 3. By default, gcc does no optimization. –O with no *n* is equivalent to –O1. *gcc* –02 gives about the same level of optimization as VOS *cc* –O.
-S	preprocess and translate, but don't assemble or bind.
-traditional	follow pre-ANSI C standard.
-U*macro*	undefine a preprocessor macro.
-v	display subprocess command lines as they are invoked.
-Wl,a*rgument,argument* . . .	Pass *arguments* directly to the binder subprocess.

Archives and Object Libraries

Archives and object libraries are two different mechanisms for implementing the same functionality: conditionally binding the object modules a program requires. On VOS, the traditional way of implementing this is object libraries. An object library is nothing more than a directory containing .obj files. The add_entry_names command creates links to map symbol names to object filenames. The binder searches an object library to resolve external names after it has read all other object files.

On Unix, this functionality is implemented with archive files. An archive is created using the ar utility:

```
ar cru libname.a name1.o name2.o
```

This creates an archive file named libname.a and puts two object files, name1.o name2.o, on it. To list the contents of an archive, use ar again:

```
ar tv lib name.a . . .
```

To use the archive, just include it on the bind command line or bindfile:

```
bind . . . libname.a . . .
```

The binder will search the archive only at the point it occurs in the command line. This is different from object libraries, which are searched after the binder has processed everything else in the command line. So if an undefined symbol is first referenced after the archive that defines it, the binder will not load it.

VOS gcc also supports a search mechanism commonly found on Unix systems: these are the –l and –L options. Given a list of directories defined with the –L option, gcc will search them for libname.a when –lname argument is seen. So for example, given the following command line:

```
gcc foo.o –L . ./ . . –lmine
```

gcc will look for . ./libmine.a, and if it doesn't find it, . . / . . /libmine.a to bind in.

The two mechanisms, object libraries and archive files, are quite separate. .obj files can only go into object libraries, and cannot be put in archive files. Conversely, .o files can only be put in archive files and cannot be put in object libraries.

Using this information for building a client pm, the command line would look as follows:

```
gcc –o filename.pm filename.c –lmqic –lmqmcs –lsnastubs  &+
  –Wl,-retain_all,-map -D_POSIX_C_SOURCE=200112L
```

Note that for gcc/vcc, all calls to MQxxx functions must be in object files that are specified in the above command line or in the binder control file.

The following is an example using a binder control file:

```
!gcc -o &file_name& &obj_file_name& -lmqic –lmqmcs  &+
  -lmqicb -lsnastubs -Wl,-control,&short_file&.bind  &+
  -Wl,-retain_all -Wl,-map
```

Binding VOS C and GNU C/C++ Code Together

.o, .a and .obj files can be intermixed on the gcc command line and bound together into a single program. So, for example:

```
gcc –o prog.pm alpha.o beta.a gamma.obj
```

will bind alpha.o, beta.a and gamma.obj into a single program named prog.pm.

Any program that includes any gcc-generated C object modules must be bound using gcc. The following sequence of commands:

```
cc cat.c

gcc –c mouse.c

bind cat.obj mouse.o
```

is invalid and may produce bind errors. Replacing bind with gcc in the last command corrects this.

The main () function of any program that includes any g++-generated C++ code *must* be compiled with gcc or g++. Otherwise, some global variables may not be correctly initialized.

function main() must be in a C routine, otherwise bind reports the following error:

```
bind: Undefined entry point: main, first referenced by
s_start_c_program.
```

VOS Standard C Compared to GNU C

Both VOS Standard C (cc) and GNU C (gcc) comply with the 1989 ANSI C standard, but there are many parts of the language that the standard leaves up to the particular implementation. Also each has made numerous extensions to the language, extensions that the other does not support. When writing or porting code for compilation with gcc, users must be aware of these differences.

Language Differences

VOS Standard C and GNU C have the following implementation-specific differences:

- By default, VOS C aligns structure members using the so-called shortmap rules. GNU C aligns data using longmap rules. Setting the

  -mapping_rules longmap

  option on cc causes it to use the same alignment rules as gcc.
- cc and gcc use different include file search rules. Setting the –u on cc causes it to use search rules that are compatible with gcc.
- If a bit field is not explicitly declared as signed or unsigned, VOS C makes it unsigned. GNU C makes it signed.
- GNU C does not recognize any of the #pragma options (system_programming, longmap, etc.) that VOS C supports, and vice-versa.
- The maximum length of global identifiers in VOS C is 32 characters. In GNU C, it's 32767 characters.

In addition, both VOS cc and GNU gcc have made extensions to the language that are not supported by the other. Coders must be aware of which compiler supports what extensions. Refer to Appendix B of *The VOS Standard C Reference Manual* (R363) for a list of VOS Standard C extensions. Except as listed below, none of these extensions is supported by GNU C. The exceptions are:

- The GNU C preprocessor does not define the symbols (__CHAR_IS_SIGNED__,__PROTOTYPES__,__VOS__ and the processor type symbols (__HPPA__ et. al.).
- GNU C also allows the $ character in identifier names.
- GNU C has only limited support for the char_varying data type, while GNU C++ supports almost all of VOS Standard C's char_varying functionality. char_varying is defined in a GNU header file, <char_varying.h>, which is automatically included by <string.h>.

- GNU C supports the $longmap storage alignment keyword, but not $shortmap.
- GNU C also allows types other than int in bit-field declarations.
- GNU C also allows incomplete types.
- Any library function that can be called from VOS Standard C can also be called from GNU C. VOS C built-in functions are *not* accessible from GNU C. So, for example, strcpy_vstr_nstr may be called from GNU C, but not $substr.

Users should also avoid using GNU extensions to the C Standard that aren't supported by VOS Standard C.

Calling VOS C from GNU C/C++ and Vice-Versa

No special provisions are needed for calls from VOS C to GNU C or vice-versa. Functions may be declared and defined in either language and called from the other.

In any C++, not just GNU C++, special provisions must be made to call from C++ to C and vice-versa. To call a C function from C++, it must be declared with C linkage:

```
extern "C" char getenv (const  char  *);

pwd = getenv ("CWD");
```

Conversely, to make a function written in C++ callable from C, it must also have C linkage:

```
extern "C" char findit (char *searchstring)

{

}
```

As long as the rules for C linkage are followed, a GNU C++ function may be called from VOS C and vice-versa.

Runtimes and System Calls

VOS C Runtimes

GNU C and C++ on VOS use the native VOS Standard C runtimes. Therefore, any library functionality available in VOS C is also accessible from GNU C and C++. This includes all the VOS C extensions to the C standard. These include:

- The string manipulation functions for char_varying data—strcpy_vstr_nstr(), etc.

- VOS-specific file options (file organization, locking mode, etc.) in file I/O.
- %v (char_varying) conversion specifier in the formatted I/O.

GNU C and C++ on VOS use an alternate implementation of setjmp () and longjmp () than VOS C uses by default. If your program will be doing a longjmp () from GNU C to VOS C code or vice-versa, the VOS C code needs to use this alternative implementation as well. This is done by defining the _UX_SETJMP preprocessor symbol when compiling the VOS code.

POSIX Runtimes

When compiling or binding GNU C or C++ programs, the VOS POSIX runtime option should be used. When compiling, one of the POSIX preprocessor symbols (_POSIX_C_SOURCE) should be defined with the appropriate value. When binding, your object library paths should be set to contain the following directories in the order given:

(master_disk)>system>posix_object_library

(master_disk)>system>c_object_library

(master_disk)>system>object_library

If your application uses POSIX threads, define the _REENTRANT preprocessor symbol and set your object library paths to

(master_disk)>system>posix_object_library>pthreads

(master_disk)>system>posix_object_library

(master_disk)>system>c_object_library

(master_disk)>system>object_library

in the order shown.

If your application uses Berkeley Unix extensions to POSIX, define the _BSD_SOURCE preprocessor symbol, set your include library paths to

(master_disk)>system>include_library>bsd

(master_disk)>system>include_library

in the order shown, and your object library paths to

(master_disk)>system>posix_object_library>bsd

(master_disk)>system>posix_object_library

(master_disk)>system>c_object_library

(master_disk)>system>object_library

in the order shown.

If your application uses System V extensions to POSIX, define the _SVID_SOURCE preprocessor symbol and set your include library paths to

(master_disk)>system>include_library>sysv

(master_disk)>system>include_library

in the order shown, and your object library paths to

(master_disk)>system>posix_object_library>sysv

(master_disk)>system>posix_object_library

(master_disk)>system>c_object_library

(master_disk)>system>object_library

in the order shown.

System Calls

On VOS, system calls are made like ordinary subroutine calls, so no special considerations are needed when calling system subroutines from GNU C code. For C++ code, system subroutines need to be declared with C linkage.

Other GNU Tools

In addition to ar, gcc ang g++, the VOS GNU tools package comes with a large number of other useful program development tools. Many of these require extensive Unix experience to know how to use well, but here are a few:

bash Unix-style command processor. Powerful but cryptic command syntax.

c++filt

In order to support name overloading, g++ "mangles" external function names. Some tools like gdb will automatically "demangle" these names, but other tools (such as the binder) don't. If the binder reports a strange undefined symbol name, say boomer__CPCc, pass it to c++filt:

```
c++filt  boomer__CPCc
```

which will display the name and parameters of the function that's undefined:

```
boomer (char  const  *)  const
```

gdb Interactive debugger for GNU-generated code. Can also debug code generated by the VOS compilers.

gmake Standard Unix build tool. Again, powerful but cryptic command syntax.

nm, objdump and size

Tools for displaying .o files. objdump – disassemble displays the machine code in a .o file.

All of these options accept a --help option, which should give you enough information to get you started.

These tools accept either VOS-style greater-than sign (>) and less-than sign (<) pathname delimiters or the Unix-style slant (/) pathname delimiter in pathnames. Also, they may be invoked from either the VOS command prompt or from bash. Since the VOS pathname characters greater-than sign (>), less-than sign (<) and the number sign (#) have special meanings in bash, only Unix-style pathnames should be used there.

Frequently Asked Questions

File Formats and gcc

Question: Can I use a gcc program to create non-Unix-like files, such an indexed or sequential files?

Answer: Yes. Since VOS gcc uses the VOS Standard C runtimes, all its VOS-specific extensions to the C runtime are available to gcc code. So, for example, the “q” (sequential) mode option on the fopen () call can be used in gcc code.

VOS Tasking and gcc

Question: Can I use gcc to compile code for a VOS tasking program?

Answer: Unfortunately, no. The code that generates gcc generates will not operate correctly in a tasking program. But you can use gcc to create a program that uses POSIX threads.

Listing and Error files from gcc

Question: How can I get .error and/or .list files out of gcc?

Answer: gcc does not generate a separate .error file. Instead, it reports all compilation errors to standard error, which is usually your terminal or batch output file. Likewise, gcc does not generate .list files. But some of the information that VOS cc puts in a list file is available from gcc. To see the source with macros expanded, use the –E option. To see the generated assembly-language code, use the –S option.

Bindfiles and gcc

Question: Can I use bindfiles (*filename*.bind) with gcc? If so, how?

Answer: Yes, you can use bindfiles with gcc by having gcc pass them directly through to the binder with the Wl option:

gcc –Wl,-control,*filename*.bind –o *filename*.pm

It's always a good idea to explicitly set the output program filename using the –o option, particularly when using bindfiles.

Relocation Overflow When Binding

Question: I get a lot of "relocation overflow" errors when I bind my GNU C program. I also have some undefined symbol errors.

Answer: First, resolve the undefined symbol errors. The relocation overflow errors will probably go away after that.

VOS Debug and GNU Code

Question: Can I use VOS debug to debug code generated by gcc?

Answer: Though it's not impossible, it is rather difficult. The GNU compilers don't generate debugging information in the same format as the VOS compilers, so the VOS debugger can't read it. In VOS debug, GNU code can only be debugged in machine mode--there are no symbols, no source files, no statement boundaries.

The reverse situation is much better—you can use gdb to debug code generated by the VOS compilers. gdb can read the debugging information generated by the VOS compilers, so you can use symbolic names, source files and statement numbers there.

More Information

For more complete documentation, refer to the following:

- *Software Release Bulletin: VOS GNU C++ and GNU Tools Release 2.0.2* (R468), available online at http://stratadoc.stratus.com
- *VOS GNU C++ User's Guide* (R453), available online at http:stratadoc.stratus.com
- *Using and Porting the GNU Compiler Collection* (for gcc-2.95), shipped with the product in HTML and PostScript form. Also available at http://gcc.gnu.org
- *The C Preprocessor* (for GCC version 2), shipped with the product in HTML and PostScript form. Also available at http://gcc.gnu.org

- *VOS Standard C Reference Manual* (R363), available online at http://stratadoc.stratus.com
- *VOS POSIX.1 Reference Guide* (R502), available online at http://stratadoc.stratus.com

Compiling and Binding COBOL Clients

VOS COBOL complies with both Cobol 78 and Cobol 85 standards as well as implementing several extensions to each. The programmer must produce source files that will compile error-free in at least one of these compiler versions. This compilation produces a .obj file. The product distribution supplies an archive library, libmqicb.a containing other necessary COBOL routines. It also needs a final reference to libmqmcs.a. For the VOS client then, the final pm is built by the following command:

```
gcc –o filename.pm  filename.obj
   –L(master_disk)>system>mqseries2>lib –lmqicb -lmqmcs
```

This would be different on other client platforms.

MQSeries Environment Variables

This chapter describes the MQSeries **environment variables** required on VOS that you will need to use with the MQI applications.

While all the environment variables listed below are used by some implementations, not all are relevant to the MQSeries client implementation on VOS.

Detailed descriptions are provided for only those environment variables relevant to the current release of *MQSeries for VOS*. For a description of the role of the remaining environment variables on other platforms see *MQSeries Clients* GC33-1632.

For the 5.2 version of MQSeries for VOS, a variable is set by the command:

'export COMMAND=xxx'

when using the GNU shell, and by

'set COMMAND=yyy'

when using the VOS command shell.

In earlier versions of MQSeries the environment variables were defined in the user application probram as external shared variables, e.g.:

```
dcl MQSERVER              char(48) external shared;
MQSERVER = 'CHANNEL1/TCP/134.111.199.164';
```

In MQSeries 5.2, VOS provides true environment variables set via the VOS command line, e.g.:

```
set MQSERVER=CHANNEL1/TCP/'134.111.199.164'
```

In order to set the environment variables from a user application program, you need to use the posix setenv function.

MQSeries environment variables

The MQSeries environment variables on VOS are:

- MQSERVER
- MQCHLLIB
- MQCHLTAB
- MQCCSID

- MQREMOTELU (VOS only)
- MQREMOTETP (VOS only)

The following MQSeries environment variables are not used by the VOS 5.2 client.

- MQSNOAUT (Server OAM only)
- MQSPATH (reserved for use by service personnel)
- MQDATA (DOS, Windows 3.1, and Windows NT only)
- MQNAME (NetBIOS only)
- MQ_USER_ID (used in VOS 1.1)
- MQ_PASSWORD (used in VOS 1.1)
- MQTRACE (DOS and Windows 3.1 only)
- NLSPATH (UNIX only)
- LANG (UNIX only)

Detailed description of MQSeries environment variables and parameters

Setting MQSERVER

The MQSERVER environment variable provides one method for creating a client-connection channel. It specifies the location of the MQSeries server and the communication method to be used. The ConnectionName must be the fully- qualified network name. When it is used to define a channel a maximum message length of 4 MB is used. If larger messages are required, the client- connection channel must be defined on the server using the DEFINE CHANNEL mechanism.

Set the value of MQSERVER as follows:

export `MQSERVER ='[`*`connection`*`]'`

where *connection* represents:

[ChannelName]/[TransportType]/[ConnectionName]

where:

ChannelName must be a channel name as defined on the server.

TransportType must **TCP**.

ConnectionName is the name of the server machine as defined to the communications protocol (TransportType).

For a TCP/IP *connection*:

`'[ChannelName]/TCP/[host]([port])'`

For example,

export MQSERVER=TCPCH1/TCP/mq.stratus.com(1414)'

or

set MQSERVER=TCPCH1/TCP/mg.stratus.com(1414)'

Setting MQCHLTAB and MQCHLLIB

Set the MQCHLTAB and MQCHLLIB environment variables when using a common channel definition file among many systems. MQCHLLIB holds the path to the directory and file containing the client channel definition table, which is created by the server. This file can be copied across to the MQSeries client machine. Set the value of MQCHLTAB to the name of the file. Alternatively MQSeries looks for a default file name of amqclchl.tab. Copy the AMQCLCHL.TAB file created on the server to a VOS directory accessible by your .pm programs. On the server, the AMQCLCHL.TAB file is located in a known location ,depending on the server platform type. Be sure to transfer the file in binary mode, not character mode.

Set the value of MQCHLTAB as follows:

export MQCHLTAB=[*file name*]

or

set MQCHLTAB=[*file name*]

where file name represents the AMQCLCHL.TAB file described above.

For example,

export MQCHLTAB=AMQCLCHL.TAB

or

set MQCHLTAB=AMQCLCHL.TAB

Set the value of MQCHLLIB as follows:

export MQCHLLIB=[*path*]

or

set MQCHLLIB=[*path*]

where path represents the path to where the AMQCLCHL.TAB file is located.

For example,

export MQCHLLIB=’system/mqseries2/usr/src

or

set MQCHLLIB=>system>mqseries>usr>src

Setting MQ_USER_ID and MQ_PASSWORD

For the 5.2 version MQ_USER_ID and MQ_PASSWORD do not need to be set up as environmental variables. The client supplies them automatically.

Setting MQCCSID

The MQCCSID environment variable represents the coded character set number used by the client and overides the machines configured CCSID.

Set the value of MQCCID as follows:

export MQCCSID=`[*codepage*]’

or

set MQCCSID=`[*codepage*]’

where *codepage* represents a codepage identifier supported by the target server machine.

For Example,

export MQCCSID=819

or

set MQCCSID=819

Channel exits

Channel exit programs are available to the MQSeries client environment on VOS. There are three exit types::

- Security exit
- Send exit
- Receive.exit

These exits are available at both the client and server end of the channel unless you are using the MQSERVER environment variable.

The channel security exit can be used to verify that the partner at the other end of the channel is genuine.

The send and receive exits may operate as a pair to perform tasks such as data compression and decompression, data encryption and decryption. You can specify a list of send and receive programs to be run in succession.

The exact nature and action of these exit programs are the responsibility of the programmer. The complete details of using these programs are available in the MQSeries Intercommunication manual (SC33-1872).

To use an exit function on a channel, a field in the MQCD (Channel data structure) must be set up with the entry name of the function. Most platforms that support dynamic linking place all of their exit functions in a dynamic link libraries so that the function loads and executes when referenced at run-time.

VOS does not support dynamic linking, and so must statically link in all of the exit functions its application might need. Thus VOS requires an entry map of all the exit functions that might be called from the channel code. To generate such a map, the invocation of gcc must include the –Wl,-retain_all option to pass the request for the map through to the binder. It may then look up the location of the function using the name in the MQCD entry at run-time.

Configuration

This chapter describes the configuration requirements to support an MQSeries Client on VOS. It includes how to configure items on a VOS system and what needs to be configured on the server system where the queue manager resides.

In MQSeries the logical communication links are called MQI channels. You set up channel definitions at each end of your link so that your MQSeries application on the MQSeries client can communicate with the queue manager on the server. There is a detailed description of how to do this in the chapter on, "Establishing communications".

Before you define your MQI channels:

Define the connection at each end:

- Configure the connection.
- Record the values of the parameters that you will need for the channel definitions later on.
- Enable the server to detect incoming network requests from your MQSeries client. This involves starting a listener.

Deciding which network transport type to use

When you define your MQI channels, each channel definition must specify a Transmission protocol (Transport Type) attribute. A server is not restricted to one protocol, so different channel definitions can specify different protocols.

For MQSeries clients, it may be useful to have alternate MQI channels using different transmission protocols.

Your choice of transmission protocol also depends on your particular combination of MQSeries client and server platforms. Check the MQseries server documentation for details regarding support for a particular network transport.

A client style connection supports only one type of channel definitions – the *clntconn* and *srvconn* set.

Defining a TCP/IP Connection

The steps that you must take to establish a TCP/IP connection for MQSeries are detailed below.

On the MQSeries client Initialize TCP/IP.

On the server There are three things to do:

1. Decide on a port number.

   The default port connection is 1414. Port number 1414 is assigned by the Internet Assigned Numbers Authority to MQSeries.

2. Initialize STCP/IP, and record the network address of the server machine.

3. Configure files (or run a command) to specify the port number and to run a listener program (non-MVS/ESA). On MVS/ESA, start a channel initiator and a listener.

On the server

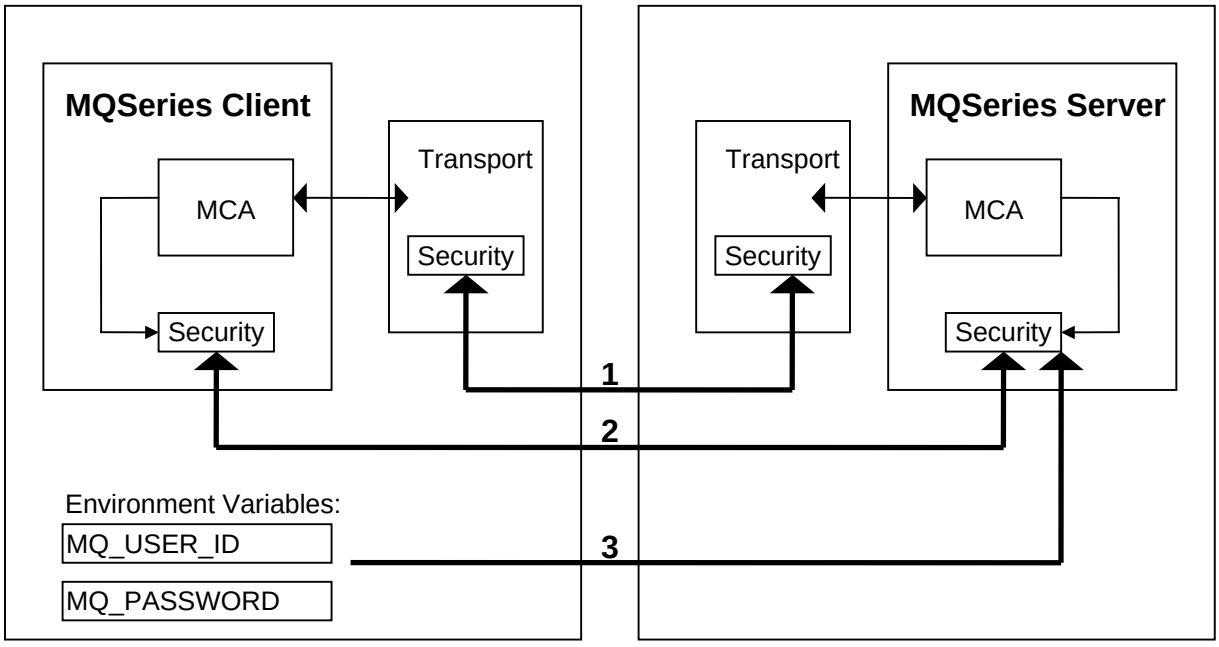
1. Start a listener, or create a listening attachment (non-MVS/ESA) by configuring the invocable TP RECV for the executable named amqcrs6a.

Or Start a channel initiator and a listener (MVS/ESA).

MQSeries Client Security

You must consider MQSeries client security so that the client applications do not have unrestricted access to resources on the server.

There are two aspects to security between a client application and its queue manager server: authentication and access control.



1. Channel security exits

The channel security exits for client to server communication can work in the same way as for server to server communication. A protocol independent pair of exits provide mutual authentication of both the client and the server. A full description is given in the MQSeries Intercommunication manual.

If no security exits are provided, see "Access Control" topic for details.

2. User ID and password passed to a channel security exit

In client to server communication, the channel security exits do not have to operate as a pair. The exit on the MQSeries client side may be omitted. In this case the user ID is placed in the channel descriptor

(MQCD) and the security exit can alter it, if required. For the VOS 5.2 client,the user ID that is passed to the server is the currently logged-on user ID on the client.

The values of the user ID and , if available, the password can be used by the server security exit to establish the identity of the MQSeries client..

If no security exits are provided, see "Access Control" topic for details.

On the VOS 5.2 client the MQ_USER_ID and MQ_PASSWORD environment variables are not supported..

Access Control

Access control in MQSeries is based upon the user identifier associated with the process making MQI calls. For MQSeries clients, the process that issues the MQI calls is the server Message Channel Agent. The user identifier used by the server MCA is that contained in the MCAUserIdentifier field of the MQCD. The contents of MCAUserIdentifier are determined by the following:

- Any values set by security exits
- The user_ID for VOS and unix clients
- MCAUSER (in server-connection channel definition)
- Default MCAUSER value (from SYSTEM.DEF.SVRCONN)

Depending upon the combination of settings of the above, MCAUserIdentifier is set to the appropriate value. If security exits are provided, MCAUserIdentifier may be set by the exit. Otherwise MCAUserIdentifier is determined as shown in the following table:

MQI Client MQ_USER_ID	Server channel MCAUSER	Value Used	Note
Set or Not Set	Set	MCAUSER	
Set	Blanks	MQ_USER_ID	
Set or Not Set	Not Set or Blanks	TCP: User ID from inted.conf entry	1,2

Notes:

1. If no MCAUSER parameter is set, MCAUSER is set to the user ID associated with the incoming conversation.

Establishing communications

This topic begins with a description of what channels are and how they are defined in an MQSeries client and server environment.

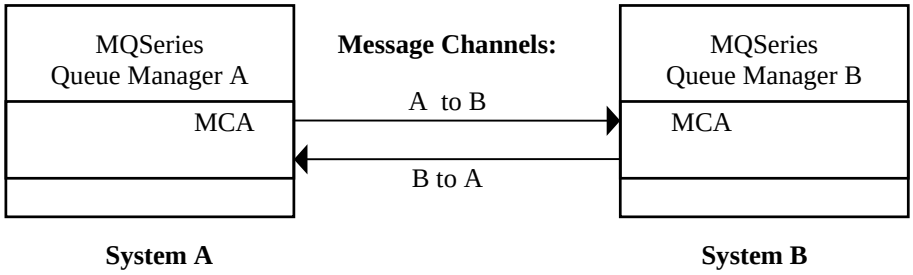
Then it gives the steps you must follow to define your channels.

What is a channel?

A channel is a logical communication link. There are two different categories of channel in MQSeries (with different channel types within these categories):

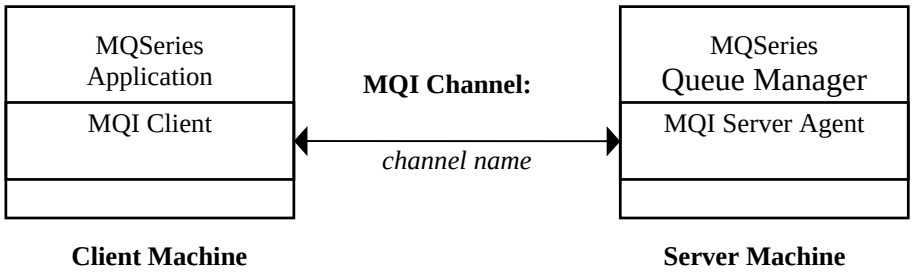
Message channel

This connects two queue managers via message channel agents (MCAs), and is unidirectional. Its purpose is to transfer messages from one queue manager to another. A channel definition exists at the sending end of the link and at the receiving end.



MQI channel

This connects an MQSeries client to a queue manager on a server machine, and is established when you issue an MQCONN or MQCONNX call. It is for the transfer of MQI calls and responses only and it is bi-directional. A channel definition exists for each end of the link but all definitions reside on the Queue Manager Server. There are different ways of creating and using the channel definitions (see "Connecting the MQSeries client and server - channel definitions").



An MQI channel can be used to connect a client to a single queue manager, or to a queue manager that is part of a queue-sharing group.

For more information on shared queues see the MQSeries for OS/390 Concepts and Planning Guide and the MQSeries Intercommunication book.

Channel definitions, of both categories described above, must include a channel type as well as a channel name. You can choose to use different channel types according to the application you are designing, but the same channel name must be used at both ends of each combination.

Message Channel Types

The various types of MQSeries **message channel** do not apply to the MQI client and server environment. See the *MQSeries Distributed Queuing Guide* for details.

MQI Channel Types

There are two types of MQI channel definitions. Together they define a bi-directional MQI channel. All channel definitions are created and reside on the queue manager server platform.

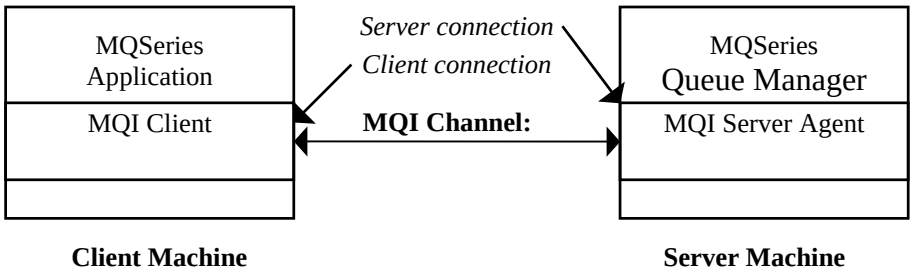
Client Connection: This type is used by the MQI client.

Server Connection: This type is used by the server running the queue manager with which the MQSeries application, running in a MQI client environment, will communicate.

Connecting the MQSeries client and server - channel definitions

MQSeries channel definitions must exist and environment variables that reference the channel definitions must be set to establish communications. Use either MQSERVER alone or the MQCHLLIB and MQCHLTAB environment variables. See the MQSeries Environment Variables section in this document for a more detailed explanation.

The connection between the MQSeries client and the queue manager on the server is a bi-directional MQI channel that is established when you issue an MQCONN call. To create any new channel you have to create two channel definitions, one for each end of the connection, using the same channel name and compatible channel types. In this case the channel types are server connection (*svrconn*) and client connection.



There are two different ways of referencing MQSeries channel definitions to enable the client machine MQSeries application access to the channel.

Both methods rely on *clntconn* and *srvconn* channel definitions having been created on the MQSeries server. You create the both the client connection channel and the server connection channel definitions on the server machine,

1. Reference the *clntconn* channel definition on the client machine using MQSERVER.

This is the easiest method, and it applies to any combination of MQSeries client and server platforms. Use it when you are getting started on the system, or to test your set up.

Use the environment variable MQSERVER on the MQSeries client machine to define a simple client connection channel (see "Using MQSeries environment variables").

2. Reference the *clntconn* channel definition on the client machine using the MQCHLTAB file.

Use this method when you are setting up a number of channels and MQSeries client machines at once.

Use the environment variables MQCHLLIB and MQCHLTAB on the MQSeries client machine to access the MQSeries client channel definition table (see "Using MQSeries environment variables").

Defining your channels

First start the queue manager on the server.

Second, define the channels on your server.

Third, go to the section that describes the method you are going to use:

- "Reference the *clntconn* definition using MQSERVER"
- "Reference the *clntconn* definition using MQCHLTAB and MQCHLLIB"

For platforms other than VOS see the *MQSeries Clients* document GC33-1632.

On the Server

On the server machine use MQSeries commands (MQSC) to define channels. For more details about the MQSC, refer to the *MQSeries Command Reference*.

On the server machine, define a channel with your chosen name and a channel type of server connection.

For example:

```
DEFINE CHANNEL(TCPCH1) CHLTYPE(SVRCONN) TRPTYPE(TCP) +

MCAUSER(‘mqmusr’) DESCR('Server connection to Client') +

REPLACE
```

This channel definition is kept in the channel definition table associated with the queue manager running on the server.

Also on the server machine, define a channel with the same name and a channel type of *client connection*.

The connection name (CONNAME) must be stated. For TCP/IP this is the network address of the server machine. It is a good idea to specify the queue manager name (QMNAME) to which you want your MQSeries application, running in the client environment, to connect. See "Running applications on MQSeries clients".

For example:

```
DEFINE CHANNEL(TCPCH1) CHLTYPE(CLNTCONN) TRPTYPE(TCP) +

CONNAME(9.20.4.26) QMNAME(QM2) DESCR('Client connection) +

REPLACE
```

This channel definition is kept in the client channel definition table associated with the queue manager running on the server. This file is called **AMQCLCHL.TAB** and is in the directory:

For OS/2 or Windows NT: \mqm\qmgrs\queuemanagername\@ipcc

For UNIX systems: /*mqmtop*/qmgrs/QUEUEMANAGERNAME/@ipcc

Note that QUEUEMANAGERNAMEis case sensitive for UNIX systems.

For MVS/ESA systems it is kept with all other object definitions on pageset zero.

Reference the *clntconn* definition using MQSERVER

On the MQI client

When you require a simple channel definition use the single environment variable MQSERVER (see "Using MQSeries environment variables").

A simple TCP/IP channel may be defined on VOS as follows:

```
export MQSERVER= '[ChannelName]/TCP/[ip addr(port)]'
```

For example, on VOS:

```
export MQSERVER=  'TCPCH1/TCP/mqseries.stratus.com(1414)'
```

On the MQSeries client, all MQCONN requests then attempt to use the channel you have defined.

Note: The MQSERVER environment variable takes priority over any client channel definition pointed to by MQCHLLIB and MQCHLTAB.

Reference the *clntconn* definition using MQCHLTAB and MQCHLLIB

On the Client

On the MQSeries client machine, use the environment variables MQCHLLIB and MQCHLTAB to enable the MQSeries application to access the client channel definition table copied to the client machine from the server (not a server on OS/400 or MVS/ESA, however).

MQCHLLIB Specifies the path to the directory containing the channel definition file. If not specified, the default used is DefaultPrefix from the mqs.ini file.

Note: You must copy the channel definition table to the location specified in the MQCHLLIB environment variable. Remember to copy this file from the server in binary mode, not character (ASCII) mode.

MQCHLTAB Specifies the name of the file to use. If not specified, the default client channel definition table name (AMQCLCHL.TAB) is used.

For example, to set the environment variables on a VOS system in a BASH shell :

```
export MQCHLLIB= '/system/mqseries2/qmgrs/QUEUEMANAGERNAME/@ipc'
```

```
export MQCHLTAB=AMQCLCHL.TAB'
```

The MQCHLLIB and MQCHLTAB variables might be used to point to a client channel definition table located on a file server accessible from many MQSeries client machines.

Alternatively, copy the client channel definition table, AMQCLCHL.TAB (a binary file) onto the client machine and again use MQCHLLIB and MQCHLTAB to specify the location of client channel definition table.

If you use FTP to copy the file, remember to type bin to set binary mode; do not use the default ascii mode.

Notes:

1. The MQSERVER environment variable overrides the client channel definition pointed to by MQCHLLIB and MQCHLTAB.

Using the MQI

When you write your MQSeries application, you need to be aware of the differences between running it in an MQSeries client environment and running it in the full MQSeries queue manager environment.

For information on the MQI programming see:

MQSeries Application Programming Guide, SC33-0807

MQSeries Application Programming Reference, SC33-1673

Limiting the maximum message length

The MAXMSGL parameter in a channel definition can be used to limit the maximum message length of a message allowed to be transmitted along a client connection. If any attempt is made by an MQSeries application to use the MQPUT call or the MQGET call with a message larger than this, an error code is returned to the application.

The maximum message size that can be specified on any platform is 4 MB (4 194 304 bytes).

Choose client or server coded character set identifier (CCSID)

The data passed across the MQI from the application to the client stub should be in the local CCSID (coded character set identifier), encoded for the MQSeries client. If the connected queue manager requires the data to be converted, this will be done by the client support code.

The client code will assume that the character data crossing the MQI in the client is in the CCSID configured for that machine. If this CCSID is an unsupported CCSID or is not the required CCSID, it can be overridden with the MQCCSID environment variable, for example:

```
export MQCCSID=819
```

Note: This does not apply to application data in the message.

Design considerations

There are no particular design considerations peculiar to the VOS environment.

MQINQ considerations

Some values queried using MQINQ will be modified by the client code.

CCSID is set to the client CCSID, not that of the queue manager.

MaxMsgLength is reduced if it is restricted by the channel definition. This will be the lower of:

- The value defined in the queue definition, or
- The value defined in the channel definition.

Syncpoint considerations

Within MQSeries, one of the roles of the queue manager is syncpoint coordination within an application. If the application has been linked to a client stub, then it can issue MQCMIT and MQBACK, but there will be no syncpoint coordination. Synchronization is limited to MQSeries server resources only.

Trigger monitor for the client environment

MQSeries client application triggering is supported only in those environments for which an MQSeries queue manager operates. There is no trigger monitor that operates within the VOS environment thus application triggering is not supported by MQSeries client applications operating in a VOS environment.

Triggering is explained in detail in the *MQSeries Application Programming Guide*.

Building Applications for MQI clients

You can write applications in C language or COBOL.

You must link it to the relevant client library file.

This topic lists points to consider when running an application in an MQSeries client environment and describes how to compile and link your application code with the MQSeries client code.

MQI client environment

You can run an MQSeries application in both a full MQSeries environment and in an MQSeries client environment without changing your code, providing:

- It does not need to connect to more than one queue manager concurrently
- The queue manager name is not prefixed with an asterisk (*) on an MQCONN call

Note: The libraries you use at link-edit time determine the environment your application must run in.

When working in the MQSeries client environment, remember:

- Each application running in the MQSeries client environment has its own connections to servers. It will have one connection to every server it requires, a connection being established with each MQCONN call the application issues.
- An application sends and gets messages synchronously.
- All data conversion is done by the server, but see also the topic concerning "MQCCSID".
- Triggering in the MQSeries client environment is supported in UNIX systems, OS/2, Windows 3.1, and Windows NT environments only. The trigger monitor and the application to be started must be on the same system.
- Messages sent by MQSeries applications running on MQSeries clients contribute to triggering in exactly the same way as any other messages, and they can be used to trigger programs on the server.

Channel Exits

The channel exits available to the VOS MQSeries client environment are:

- Send exit
- Receive exit
- Security exit

These exits are available at both the client and server ends of the channel.

Remember, exits are not available to your application if you are using the MQSERVER environment variable. Exits are explained in the *MQSeries Distributed Queuing Guide*.

The send and receive exit work together. There are several possible ways in which you may choose to use them:

- Segmenting and reassembling a message
- Compressing and decompressing data in a message
- Encrypting and decrypting user data
- Journaling each message sent and received

You can use the security exit to ensure that the MQSeries client and server machines are correctly identified, as well as to control access to each machine.

MQI client and queue managers

This topic explains the various ways in which an application running in an MQSeries client environment can connect to a queue manager. It covers the relationship of the MQSERVER environment variable provided by MQSeries, and the role of the client channel definition file created by MQSeries.

Client connection to queue manager

When an application running in an MQSeries client environment issues an MQCONN call, the client code identifies how it is to make the connection:

1. If the MQSERVER environment variable is set, the channel it defines will be used.

2. If the MQCHLLIB and MQCHLTAB environment variables are set, the client channel definition table will be used.

Notes:

1. If the client code fails to find any of these, the MQCONN call will fail.
2. The channel name established from either the first segment of the MQSERVER variable or from the client channel definition table must match the SVRCONN channel name defined on the server for the MQCONN call to succeed.

Using MQSERVER

If you use the MQSERVER environment variable to define the channel between your MQSeries client machine and a server machine, this is the only channel available to your application and no reference is made to the client channel definition table. In this situation, the 'listening' program that you have running on the server machine determines the queue manager that your application will connect. It will be the same queue manager to which the listener program is connected.

If the MQCONN request specifies a queue manager other than the one the listener is connected to, the MQCONN request fails with return code MQRC_Q_MGR_NAME_ERROR.

Using the DEFINE CHANNEL command on the server

If you use the MQSC DEFINE CHANNEL command, the details you provide are placed in the client channel definition table. It is this file that the client code accesses, in channel name sequence, to determine the channel an application will use.

The contents of the Name parameter of the MQCONN call determines what processing will be carried out at the server end.

Role of the client channel definition file

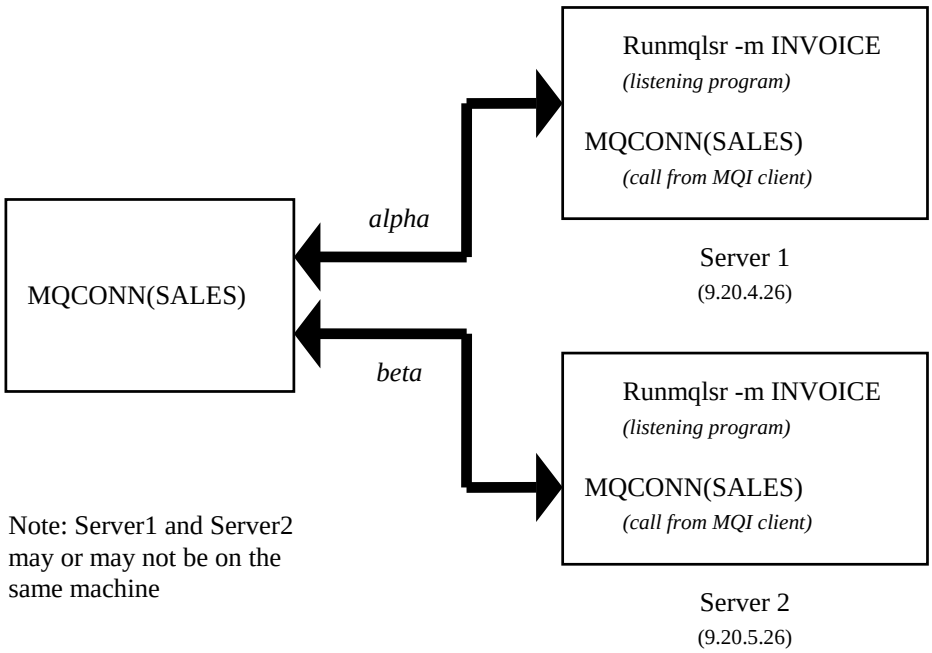
The client channel definition table is created when you define the first of the connections between an MQSeries client and a server. See "Connecting the MQSeries client and server - channel definitions" for more information on what you have to define and how you do it.

Note: The same file may be used by more than one MQSeries client. You change the name and location of this file using the **MQCHLLIB** and **MQCHLTAB** MQSeries environment variables. See "Using MQSeries environment variables" for details of these and all the other MQSeries environment variables.

You may choose to define connections to more than one server machine because:

- You need a backup system.
- You want to be able to move your queue managers without changing any application code.
- You need to access multiple queue managers and this requires the least resource.

In each of the following examples, the network is the same; there is a connection defined to two servers from the same MQSeries client. There are two queue managers running on each server machine, one named SALES and the other named INVOICE.



The definition for the channels in these examples are:

```
DEFINE CHANNEL(ALPHA) CHLTYPE(SVRCONN) TRPTYPE(TCP) +
DESCR('Server connection to MQSeries client')

DEFINE CHANNEL(APLHA) CHLTYPE(CLNTCONN) TRPTYPE(TCP) +
CONNAME(9.20.4.26) DESCR('MQSeries client connection to server 1') +
QMNAME(SALES)

DEFINE CHANNEL(BETA) CHLTYPE(SVRCONN) TRPTYPE(TCP) +
DESCR('Server connection to MQSeries client')

DEFINE CHANNEL(BETA) CHLTYPE(CLNTCONN) TRPTYPE(TCP) +
CONNAME(9.20.5.26) DESCR('MQSeries client connection to server 2') +
QMNAME(SALES)
```

In each example the MQCONN call is different. Here is an explanation of what would happen in general terms followed by what happens in the specific example presented.

Queue manager name specified

The application requires a connection to a specific queue manager, with the name of SALES, as seen in the MQI call:

MQCONN (SALES)

In general terms, MQSeries will search the client channel definition table, in channel name order, looking in the queue manager field, for a SALES entry.

If a match is found:

1. The transmission protocol and the associated connection name are extracted.

2. An attempt is made to start the channel to the machine (identified by the connection name). If this is successful, the application will continue. It requires:
   - A listener to be running on the server
   - The listener to be connected to the same queue manager as the one to which the client wishes to connect.

3. If the attempt to start the channel fails and there is more than one entry in the client channel definition table (in this example there are two entries), the file is searched for a further match. If a match is found, processing continues at step 1.

4. If no match is found, or the channel fails to start, the application is unable to connect. An appropriate reason code and completion code are returned in the MQCONN call. The application must take action based on the reason and completion codes returned.

Following these rules, this is exactly what will happen in this instance:

1. The client channel definition table is scanned for the entry for the channel name ALPHA.

2. The queue manager name in the definition is SALES, matching with the application MQCONN call.

3. An attempt to start the channel is made - this is successful.

4. A check to see that a listener is running shows that there is one running, but it is not connected to the SALES queue manager.

5. The client channel definition table is scanned again, this time for the entry for the channel name BETA.

6. The queue manager name in the definition is SALES, matching with the application MQCONN call.

7. An attempt to start the channel is made - this is successful.

8. A check to see that a listener is running, shows that there is one running, and it is connected to the SALES queue manager. The MQI call sent by the application in the MQSeries client environment has been processed successfully by the server 2 machine and the application will continue its processing.

Queue manager name prefixed with an asterisk (*)

In this example the application is not concerned about to which queue manager it connects; the application issues:

MQCONN (*SALES)

MQSeries will search the client channel definition table, in channel name order, looking in the queue manager field, for a SALES entry.

If a match is found:

1. The transmission protocol and the associated connection name are extracted.

2. An attempt is made to start the channel to the machine (identified by the connection name). If this is successful, the application will continue. It requires:

- A listener to be running on the server

3. If the attempt to start the channel fails and there is more than one entry in the client channel definition table (in this example there are two entries), the file is searched for a further match. If a match is found, processing continues at step 1.

4. If no match is found, or there are no more entries in the client channel definition table and the channel has failed to start, the application is unable to connect. An appropriate reason code and completion code are returned in the MQCONN call. The application must take action based on the reason and completion codes returned.

Following these rules, this is exactly what will happen in this instance:

1. The client channel definition table is scanned for the entry for the channel name ALPHA.

2. The queue manager name in the definition is SALES, matching with the application MQCONN call.

3. An attempt to start the channel is made - this is successful.

4. A check to see that a listener is running, shows that there is one running. It is not connected to the SALES queue manager, but because the MQI call parameter has an asterisk (*) in front of it, no check is made. The application will be connected to the INVOICE queue manager and will continue processing.

Queue manager name is blank or an asterisk (*)

In this example the application is not concerned about to which queue manager it connects. This is treated in the same way as the previous example.

Note: If this application was running in an environment other than an MQSeries client, and the name is blank, it would be attempting to connect to the default queue manager. This is not the case when it is run from a client environment, as there is no concept of a default queue manager. The application issues:

MQCONN ("")

MQCONN (*)

MQSeries will search the client channel definition table, in channel name order, looking in the queue manager field, for a blank entry.

If a match is found:

1. The transmission protocol and the associated connection name are extracted.

2. An attempt is made to start the channel to the machine identified by the connection name. If this is successful, the application will continue. It requires:

- A listener to be running on the server

3. If the attempt to start the channel fails and there is more than one entry in the client channel definition table (in this example there are two

entries), the file is searched for a further match. If a match is found, processing continues at step 1.

4. If no match is found, or there are no more entries in the client channel definition table and the channel has failed to start, the application is unable to continue processing and needs to output a suitable message.

Following these rules, this is exactly what will happen in this instance:

1. The client channel definition table is scanned for the entry for the channel name ALPHA.

2. The queue manager name in the definition is SALES. This does not match the MQCONN call parameter, which requires the queue manager name to be blank.

3. Again the client channel definition table is scanned. The next entry is for the channel name BETA.

4. The queue manager name in the definition is SALES. Once again, this does not match the MQCONN call parameter, which requires the queue manager name to be blank.

5. There are no further entries in the client channel definition table, the application cannot continue and will be returned error number 2059 - Queue Manager unavailable.

Problem Determination

Here the return codes, error logs, and error messages are discussed. Some common problems when running applications in the MQSeries client environment are examined.

An application running in the MQSeries client environment receives MQRC_* reason codes in the same way as MQSeries server applications. However, there are additional reason codes for error conditions associated with MQSeries clients. For example:

- Remote machine not responding
- Communications line error
- Invalid machine address

The most common time for errors to occur is when an application issues an **MQCONN** and receives the response **MQRC_Q_MQR_NOT_AVAILABLE**. Look in the client error log AMQERR01.LOG for a message indicating the cause of the failure. There may also be errors logged at the server, depending on the nature of the failure. Also, check that the application on the MQSeries client is linked with the correct library file.

MQI Client fails to connect

When the MQSeries client issues an MQCONN call to a server, socket and port information (TCP/IP) is exchanged between the MQSeries client and the server. For any exchange of information to take place, there must be a program on the server machine whose role is to 'listen' on the communications line for any activity. If there is no program doing this, or there is one but it has problems of its own, the MQCONN fails and the MQSeries application is returned the relevant reason code.

If the connection is successful, MQSeries protocol messages are then exchanged and further checking takes place. It is not until all these checks are successful that the MQCONN call will succeed.

During the MQSeries protocol checking phase, some aspects are negotiated while others cause the connection to fail.

For full details of the MQRC_* reason codes see the *MQSeries Application Programming Reference*.

Stopping MQI clients

Even though an MQSeries client has stopped it is still possible for the process at the server to be holding its queues open. The queues will be closed when the communications layer detects that the partner has gone.

Error messages associated with MQI client activity

When an error occurs with an MQSeries client system, error messages are put into the error files associated with the server, if possible. If the error cannot be placed there, the MQSeries client code attempts to place the error message in an error log in the root directory of the MQSeries client machine.

MQCONN Returns 2058 Error Code: MQRC_Q_MGR_NAME_ERROR

If you are using the MQCHLIB and MQCHLTAB approach to client channel definitions, you may have performed the file transfer (FTP) in ASCII mode. Make sure you transfer the MQCHLTAB file in binary mode.

MQCONN Returns 2059 Error Code: MQRC_Q_MGR_NOT_AVAILABLE

Check the connection name definitions. If using TCP/IP, can you ping the connection name specified?

Check the channel definitions on the server side. Make sure the clntconn definition includes a valid queue manager name.

IBM Verification Test Programs

This section provides sample command macros for setting up and running IBM verification tests.

Setting Library Paths

The following command macro sets the library paths required to compile and run the tests.

add_lib.cm

```
&echo command_lines
&
& Set up library paths to find gcc and MQSeries 5.2
& components.
&
add_library_path object &+
  (master_disk)>system>object_library  &+
  -after '(current_dir)'
add_library_path object &+
  (master_disk)>system>c_object_library &+
  -after '(current_dir)'
&
add_library_path object &+
  (master_disk)>system>posix_object_library &+
-after '(current_dir)'
add_library_path object &+
  (master_disk)>system>posix_object_library>bsd &+
  -after '(current_dir)'
&
add_library_path object &+
  (master_disk)>system>stcp>object_library>sbsd &+
  -after '(current_dir)'
add_library_path object &+
  (master_disk)>system>stcp>object_library>socket &+
  -after '(current_dir)'
add_library_path object &+
  (master_disk)>system>stcp>object_library>net &+
  -after '(current_dir)'
add_library_path object &+
  (master_disk)>system>stcp>object_library>common &+
  -after '(current_dir)'
add_library_path object &+
  (master_disk)>system>stcp>object_library -after
'(current_dir)'
&
add_library_path command &+
  (master_disk)>system>gnu_library>bin &+
  -after '(current_dir)'
&
add_library_path command &+
  (master_disk)>system>mqseries2>bin -after '(current_dir)'
add_library_path object  &+
  (master_disk)>system>mqseries2>lib -after '(current_dir)'
```

```
add_library_path include &+
  (master_disk)>system>mqseries2>inc -after '(current_dir)'
&
list_library_path
```

After you have run add_lib.cm, the output of the list_library_paths command should appear as follows:

```
include library directories:
     (current_dir)
     %es#m30>system>mqseries2>inc
     %es#m30>system>include_library
     %es#m30>system>stcp>include_library

object library directories:
     (current_dir)
     %es#m30>system>mqseries2>lib
     %es#m30>system>stcp>object_library
     %es#m30>system>stcp>object_library>common
     %es#m30>system>stcp>object_library>net
     %es#m30>system>stcp>object_library>socket
     %es#m30>system>stcp>object_library>sbsd
     %es#m30>system>posix_object_library>bsd
     %es#m30>system>posix_object_library
     %es#m30>system>c_object_library
     %es#m30>system>object_library

command library directories:
     (current_dir)
     %es#m30>system>mqseries2>bin
     %es#m30>system>gnu_library>bin
     %es#m30>system>command_library
     %es#m30>system>tools_library
     %es#m30>system>applications_library
     %es#m30>system>stcp>command_library

message library directories:
     (referencing_dir)>(language_name)
     (referencing_dir)
     %es#m30>system>message_library>(language_name)
     %es#m30>system>message_library>us_english
```

Compiling and Running the Test Programs

You can use the sample command macros in this section to compile and run two test programs, amqsget0 and amqsput0, from Chapter 6 of the manual *IBM MQSeries Clients* (GC33-1632-09), “Verifying the Installation.” See that manual for more details and instructions on running the progams.

set_var.cm

This command macro sets up the MQSeries environment variables as described on page 39.

```
&echo command_lines
&
set MQCCSID=819
&
set MQSERVER=CHANNEL1/TCP/'134.111.199.164'
&
```

c_compile.cm

This command macro compiles a C program. You can use it to compile amqsget0 and amqsput0.

```
&begin_parameters
     target program:string, required
     debug  switch(-debug)
&end_parameters
&echo command_lines
&
&if &debug&
&then vcc -g -o &target&.pm &target&.c -lmqic -lmqmcs &+
  -lmqicb -lsnastubs -Wl,-retain_all,-map &+
  -D_POSIX_C_SOURCE=200112L

&else vcc -o &target&.pm &target&.c -lmqic -lmqmcs &+
  -lmqicb -lsnastubs -Wl,-retain_all,-map &+
  -D_POSIX_C_SOURCE=200112L
```

do_c_get.cm

```
display_line Displays data previously entered with &+
  do_c_put.cm.
&echo command_lines
amqsget0 QUEUE1 queue.manager.1
```

do_c_put.cm

```
display_line Type some lines of data followed by a &+
  blank line.
&echo command_lines
amqsput0 QUEUE1 queue.manager.1
```

big_c_put.cm

This command macro sends data to the server queue.

```
&echo command_lines input_lines
&attach_input
amqsput0 QUEUE1 queue.manager.1
Test Line 0001 With some data
Test Line 0002 With some data
Test Line 0003 With some data
Test Line 0006 With some data
Test Line 0007 With some data
Test Line 0008 With some data
Test Line 0009 With some data
Test Line 0010 With some data
Test Line 0011 With some data
Test Line 0012 With some data
Test Line 0013 With some data
Test Line 0014 With some data
Test Line 0015 With some data
Test Line 0016 With some data
Test Line 0017 With some data
Test Line 0018 With some data
Test Line 0019 With some data
Test Line 0020 With some data
Test Line 0021 With some data
Test Line 0022 With some data
Test Line 0023 With some data
Test Line 0024 With some data
Test Line 0025 With some data
Test Line 0026 With some data
Test Line 0027 With some data
Test Line 0028 With some data
Test Line 0029 With some data
```

(The line after `Test Line 0029 With some data` **must be blank**.)

cobol_compile.cm

This command macro compiles a COBOL program. You can use it to compile amqsget0 and amqsput0.

```
&begin_parameters
     target program:string, required
&end_parameters
&echo command_lines
&
cobol &target&
&
vcc -o &target&.pm &target&.obj &+
  -L(master_disk)>system>mqseries2>lib -lmqicb -lmqmcs &+
  -lsnastubs  -Wl,-retain_all,-map
&
```

do_cobol_get.cm

```
display_line Displays data from previously entered &+
  with do_cobol_put.cm.
&echo command_lines input_lines
&attach_input
&
& amq0get0 QUEUE1 queue.manager.1
amq0get0
QUEUE1
```

do_cobol_put.cm

```
display_line Type some lines of data followed by a &+
  blank line.
&echo command_lines input_lines
&attach_input
& amq0put0 QUEUE1 queue.manager.1
amq0put0
QUEUE1
```

big_cobol_put.cm

This command macro sends data to the server queue.

```
&echo command_lines input_lines
&attach_input
& amq0put0 QUEUE1 queue.manager.1
amq0put0
QUEUE1
Test Line 0001 With some data
Test Line 0002 With some data
Test Line 0003 With some data
Test Line 0006 With some data
Test Line 0007 With some data
Test Line 0008 With some data
Test Line 0009 With some data
Test Line 0010 With some data
Test Line 0011 With some data
Test Line 0012 With some data
Test Line 0013 With some data
Test Line 0014 With some data
Test Line 0015 With some data
Test Line 0016 With some data
Test Line 0017 With some data
Test Line 0018 With some data
Test Line 0019 With some data
Test Line 0020 With some data
Test Line 0021 With some data
Test Line 0022 With some data
Test Line 0023 With some data
Test Line 0024 With some data
Test Line 0025 With some data
```

```
Test Line 0026 With some data
Test Line 0027 With some data
Test Line 0028 With some data
Test Line 0029 With some data

```

(The line after `Test Line 0029 With some data` **must be blank**.)

Notices

The following paragraph does not apply to any country where such provisions are inconsistent with local law:

STRATUS TECHNOLOGIES, INC. and INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDE THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact Laboratory Counsel, Mail Point 151, IBM United Kingdom Laboratories, Hursley Park, Winchester, Hampshire SO21 2JN, England. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

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

The Customer is responsible for obtaining and complying with licensing requirements for any MQSeries server product which the MQSeries Client for VOS is connected to, including having to obtain an MQSeries Server Extended Authorization from IBM or a Distributor authorized by IBM.

Index

Access Control, 48
AMQCLCHL.TAB, 41, 54
Archives, 25
authentication, 47
C language, 59
CCSID, 15, 57
CCSID Table, 15
channel definition, 17
channel definition table, 61
channel definitions, 45, 52, 53
channel exits, 60
Channel exits, 43
Channel Exits, 60
Channel security exits, 47
channels, 51
Client Connection, 52
client environment, 59
Client Security, 47
clntconn, 55
COBOL, 59
COBOL Clients, 37
coded character set identifier, 57
common problems, 67
compile, 59
Configuration, 45
CONNAME, 54
connect to a queue manager, 61
data conversion, 59
default port connection, 46
environment variables, 39
Environment Variables, 39
error logs, 67
error messages, 67
exits, 47
FTP, 56
GNU Tools, xi
install_new_release, 21
installing, 21
invocable TP, 46
LANG, 40
libraries, 59
link, 59
listener, 46
Machine Requirements, 19
maximum message length, 57
maximum message size, 57
MAXMSGL, 57
MaxMsgLength, 58
MCAUSER, 48
MCAUserIdentifier, 48
Message channel, 51
Message Channel Types, 52
MQ_USER_ID, 40, 42, 48
MQBACK, 58
MQCCSID, 39
MQCD, 43, 48
MQCHLLIB, 39, 41, 52, 55, 61
MQCHLTAB, 41, 52, 55, 61
MQCMIT, 58
MQCONN, 15, 17, 52, 55, 59, 61, 62, 63, 64, 67
MQCONN (""), 65
MQCONN (*), 65
MQCONNX, xi, 17
MQDATA, 40
MQDISC, 15
MQGET, 57
MQI channel, 51
MQI Channel Types, 52
MQI programming, 57
MQINQ, 58
mqm group, 21
mqm user_id, 21
MQNAME, 40
MQPUT, 57
MQRC_* reason codes, 67
MQRC_Q_MGR_NAME_ERROR, 61, 68
MQRC_Q_MGR_NOT_AVAILABLE, 68
MQRC_Q_MQR_NOT_AVAILABLE, 67
MQREMOTELU, 40, 46
MQREMOTETP, 40, 46
MQROOT, 21
mqs.ini file, 55
MQSC, 54
MQSC DEFINE CHANNEL, 61
MQSERVER, 39, 43, 52, 55, 60, 61
MQSNOAUT, 40
MQSPATH, 40
MQTRACE, 40
name is blank or an asterisk, 65
name prefixed with an asterisk, 64
NLSPATH, 40
Object Libraries, 25
password, 47
platforms, 13
platforms other than VOS, 53
POSIX Runtimes, 31
POSIX standard, xi
Preparing VOS GNU C and C++ Clients, 23
protocols, 15
QMNAME, 54
QUEUEMANAGERNAME, 54
Receive exit, 43, 60
return codes, 67
Security exit, 43, 60

Send exit, 43, 60
Server Connection, 52
Setting MQCCSID, 42
Setting MQSERVER, 40
software requirements, 15
SVRCONN, 61
synchronous mode, 15
syncpoint, 58
TCP/IP, xi, 17, 19, 45
trigger monitor, 58
triggering, 58
Triggering, 59
User ID, 47
VOS C Runtimes, 30
VOS Client Installation, 15
VOS: hardware and software requirements, 19