posixipc

Tcl Channel Driver for POSIX ICP Mechanisms

Legal Notices and Information

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.1	November 6, 2018	Initial coding.	GAM
1.0	December 31, 2018	Initial release.	GAM

Contents

List of Examples

Introduction

This document is about a Tcl package named `posixipc`. The `posixipc` package is a "C" based Tcl extension containing commands that allow a Tcl program to access POSIX Inter-Process Communications (IPC) facilities.

This document is also a literate program and contains all the design information and code for the `posixipc` package. Readers unfamiliar with literate programs should consult the appendix for more details.

There are two main sections in the document. The main sections describe the usage of two IPC mechanisms provided by the POSIX standard, namely:

- Message queues
- Shared memory

There is no explicit support for POSIX semaphores, but they are used in the implementation of shared memory access.

Background

Modern operating systems running on modern computing devices provide independent virtual address spaces in which a process runs. The code for a process runs isolated from other process code and interacts with the environment via the operating system. This arrangement means that processes must take special means to interact or communicate.

Many different mechanisms have been developed to allow processes to communicate. Simple communications can happen via the file system. Communications channels are also a viable means of process interaction. The POSIX standard defines the mechanisms mentioned above. Like most standards, the syntax and semantics of the IPC mechanism is defined, but implementation considerations are appropriately left to OS authors.

The mechanisms defined by the POSIX standard are basic IPC mechanisms that have proven useful for some time. Message queues allow processes to send and receive discrete messages of arbitrary content. Shared memory makes memory space available in the virtual addresses of multiple processes that contains the same data.

Unix Specific

Although the POSIX definitions do not imply a particular implementation, the interface we present here is specific to Unix-style operating systems (*e.g.* Linux and FreeBSD). In particular, this set of Tcl commands takes advantage of implementation characteristics available in the implementation of the POSIX IPC features. Specifically, most implementions of the POSIX semantics for IPC use virtual filesystems in Unix-style systems.. For message queues and shared memory, this means the identifier of the facility is also a file descriptor. Native file descriptors greatly facilitate integrating with the Tcl channel system and make dealing with event driven I/O much cleaner. With a file descriptor, much of the heavy lifting for I/O can be accomplished by the Tcl channel code using the virtual channel drivers provided in this package. For these reasons, this code is Unix specific.

Because we giving up portability to other operating systems, we do gain one distinct coding advantage. Since the package is targeted at more modern Unix-like systems, we can assume that the compiler supports newer dialects of "C". We do not take any specific advantage of compiler extensions. However, traditionally Tcl extensions are written in an old K & R style of "C". Here we use the C11 as the standard for the code. This makes for much cleaner code, in particular allowing variable declarations to occur lexically close to their usage and allowing designated initializers for variables.

Also note that the primary development environment for this package is Linux.

Tcl Channel Architecture

The interface to POSIX ICP mechanisms provided by this package is through the Tcl channel architecture. This architecture provides generic script and "C" program access to I/O and uses type-specific channel drivers to accomodate the differences in the underlying I/O mechanism.

In this package, channel drivers are provided for message queues and shared memory. This allows scripts to use conventional and well known commands to perform I/O via the POSIX IPC mechanisms. For example, once a message queue is open

and established it is possible to use `::puts` to write a message to the queue. This also enables the use of stacked channels, implemented at the script level, to handle any structure or protocol defined for a message.

However, the mapping of POSIX IPC mechanism onto the Tcl channel architecture is not perfect. Some features and semantics do not map completely or seamlessly. These issues are discussed later in this document.

Other Approaches

Over the years, there have been several Tcl packages that support IPC. For example, `tcl-mq` by Alexandros Stergiakis, provides relatively direct access to message queue system calls but does not provide a Tcl channel driver. Also by Alexandros Stergiakis, `tcl-mmap` provides a generalized channel driver to memory mapped files that is not specific to POSIX shared memory mechanisms. Both extensions are licensed by the GNU Public License, version 2 which can be an impediment for some applications. There are probably other implementation of which the author is not aware.

Package Considerations

The `posixipc` package contains contains a single ensemble command also named `posixipc`. There are two sub-commands corresponding to two POSIX IPC mechansims. The sub-commands are:

mq
: Commands for POSIX message queues.

shm
: Commands for POSIX share memory objects.

Each of the above sub-commands is also an ensemble command with sub-commands for the specific operations. The `posixipc` sub-commands are also exported from the `posixipc` namespace. This allows selective importing to save the amount of typing required for simple invocations. For example, the `mq` sub-commands can be invoked in two ways:

Example 0.1 Example mq command invocations

```
posixipc mq open /foo
namespace import ::posixipc::mq
mq open /foo
```

Package Initialization

The Tcl mechanism for dynamic loading uses a naming convention to identify the initialization function to invoke to prepare the package for use. In our case, the naming convention requires that we supply a function named, `Posixipc_Init`.

The purpose of the initialization function is to define all the commands supported by the package and then inform the interpreter that the package is loaded and ready for use.

```
<<package external function definitions>>=
int
Posixipc_Init(
    Tcl_Interp *interp)
{
#       ifdef USE_TCL_STUBS
    /*
     * Stubs have to be initialized first.
     */
    if (Tcl_InitStubs(interp, TCL_VERSION, 0) == NULL) {
        return TCL_ERROR ;
    }
#       endif /* USE_TCL_STUBS */
    /*
```

```
     * Create all the commands for the package.
     */
    <<create ipc commands>>
    /*
     * Support for package configuration command.
     */
    <<register package configuration>>
    /*
     * Declare the package as now provided.
     */
    Tcl_PkgProvide(interp, PACKAGE_NAME, PACKAGE_VERSION) ;

    return TCL_OK ;

    /*
     * Release resources on error.
     */
    <<shm release>>
    <<mq release>>
    <<ipc release>>
}
```

Creating IPC Commands

Creating the IPC commands is a straight forward, albeit, rather tedious set of Tcl API invocations. We start with creating a namespace for the package, followed by creating the commands for each type of IPC mechanism. Finally, a namespace emsemble is used to provide access to the entire set of mechanism commands. This list of function invocations is packaged as a set of literate program chunks to show the sequence.

```
<<create ipc commands>>=
<<create package namespace>>
<<create mq command>>
<<create shm command>>
<<create ipc command ensemble>>
```

To create the package namespace, we must supply its name.

```
<<create package namespace>>=
static char const pkgNamespace[] = PACKAGE_NS ;      // ❶
Tcl_Namespace *ipcNs = Tcl_CreateNamespace(interp, pkgNamespace, NULL, NULL) ;

Tcl_Obj *ipcMap = Tcl_NewDictObj() ;                 // ❷
```

❶ The `PACKAGE_NS` macro is defined in the following section.

❷ Here and in the following, we use a dictionary to accumulate the mapping for namespace ensembles. We install the mapping as part of creating the ensemble commands for each of the IPC mechanisms. This technique allows the ensembles to be extended at the Tcl script level if desired.

In the process of creating all the commands and namespaces, we need some strings that can be pieced together from components. These are used to form the fully qualified names for the commands. To accomplish this, we use some pre-processor macros to emphasize the naming pattern. Note that in what follows, we use the ablity of the compiler to concatenate consecutive string literals into a single string literal.

```
<<package macros>>=
#define NS_SEP          "::"

#define PACKAGE_NS      NS_SEP PACKAGE_NAME
```

```
#define CMD_NS(n)        PACKAGE_NS NS_SEP #n

#define MQ_NS            CMD_NS(mq)
#define MQ_SUBCMD(c)     MQ_NS NS_SEP #c

#define SHM_NS           CMD_NS(shm)
#define SHM_SUBCMD(c)    SHM_NS NS_SEP #c
```

Creating the `mq` command

The following chunk establishes a pattern we use for the other IPC command. For the `mq` command and each of the other `posixipc` sub-commands, we create a namespace and a dictionary to hold the ensemble mapping. Then, each sub-command of `mq` is created. The sub-commands are just ordinary Tcl commands implemented as "C" functions and are entered into the ensemble mapping.

Finally, after all the sub-commands are in place, the `mq` command is created as a namespace ensemble. The accumulated mapping is installed on the ensemble command and the ensemble command is exported.

```
<<create mq command>>=
<<create mq namespace>>

<<create mq open command>>
<<create mq send command>>
<<create mq receive command>>
<<create mq unlink command>>

<<create mq ensemble command>>
```

mq namespace

```
<<create mq namespace>>=
static char const mqNsName[] = MQ_NS ;
static char const mqStr[] = "mq" ;
/*
 * Create the namespace in which the sub-command resides.
 */
Tcl_Namespace *mqNs = Tcl_CreateNamespace(interp, mqNsName, NULL, NULL) ;
/*
 * Create the ensemble mapping dictionary.
 */
Tcl_Obj *mqMap = Tcl_NewDictObj() ;
```

The code to create the `mq` sub-commands follows a pattern as established in the following chunk.

mq open

```
<<create mq open command>>=
static char const mqOpenCmdName[] = MQ_SUBCMD(open) ;
static char const openStr[] = "open" ;

Tcl_CreateObjCommand(interp, mqOpenCmdName, mqOpenCmd, NULL, NULL) ;     // ❶
if (Tcl_DictObjPut(interp, mqMap, Tcl_NewStringObj(openStr, -1),
        Tcl_NewStringObj(mqOpenCmdName, -1)) != TCL_OK) {
    goto release_mq ;                                                    // ❷
}
```

❶ Here, `mqOpenCmd` is a "C" function name that implements the command. We will see this function in a following section.

❷ Ok, let's get the "go to" thing out of the way early. Yes, there are `goto` statements in the code. But they are used very sparingly and in a specific way. All the `goto` statements are always forward jumps in the code (backward jumps using `goto` truly disrupt thinking about the program flow) and are used only for error clean up. With no exceptions in "C", a `goto` is often the cleanest way to insure that allocated resources are cleaned up properly. Yes, it can be done other ways, but those other ways always involve either a lot of nesting or more variables or some abuse of the pre-processor. So, just get over any `goto` hysteria you may have.

Tests

```
<<package initialization tests>>=
test package-commands-2.1 {
    Check that posixipc::mq::open was created
} -setup {
} -cleanup {
} -body {
    info commands ::posixipc::mq::open
} -result {::posixipc::mq::open}
```

mq send

```
<<create mq send command>>=
static char const mqSendCmdName[] = MQ_SUBCMD(send) ;
static char const sendStr[] = "send" ;

Tcl_CreateObjCommand(interp, mqSendCmdName, mqSendCmd, NULL, NULL) ;
if (Tcl_DictObjPut(interp, mqMap, Tcl_NewStringObj(sendStr, -1),
        Tcl_NewStringObj(mqSendCmdName, -1)) != TCL_OK) {
    goto release_mq ;
}
```

Tests

```
<<package initialization tests>>=
test package-commands-2.2 {
    Check that posixipc::mq::send was created
} -setup {
} -cleanup {
} -body {
    info commands ::posixipc::mq::send
} -result {::posixipc::mq::send}
```

mq receive

```
<<create mq receive command>>=
static char const mqRecvCmdName[] = MQ_SUBCMD(receive) ;
static char const recvStr[] = "receive" ;

Tcl_CreateObjCommand(interp, mqRecvCmdName, mqReceiveCmd, NULL, NULL) ;
if (Tcl_DictObjPut(interp, mqMap, Tcl_NewStringObj(recvStr, -1),
        Tcl_NewStringObj(mqRecvCmdName, -1)) != TCL_OK) {
    goto release_mq ;
}
```

Tests

```
<<package initialization tests>>=
test package-commands-2.3 {
    Check that posixipc::mq::receive was created
} -setup {
} -cleanup {
} -body {
    info commands ::posixipc::mq::receive
} -result {::posixipc::mq::receive}
```

mq unlink

```
<<create mq unlink command>>=
static char const mqUnlinkCmdName[] = MQ_SUBCMD(unlink) ;
static char const unlinkStr[] = "unlink" ;

Tcl_CreateObjCommand(interp, mqUnlinkCmdName, mqUnlinkCmd, NULL, NULL) ;
if (Tcl_DictObjPut(interp, mqMap, Tcl_NewStringObj(unlinkStr, -1),
        Tcl_NewStringObj(mqUnlinkCmdName, -1)) != TCL_OK) {
    goto release_mq ;
}
```

Tests

```
<<package initialization tests>>=
test package-commands-2.4 {
    Check that posixipc::mq::unlink was created
} -setup {
} -cleanup {
} -body {
    info commands ::posixipc::mq::unlink
} -result {::posixipc::mq::unlink}
```

mq ensemble

```
<<create mq ensemble command>>=
/*
 * Create an ensemble command on the mq namespace.
 */
Tcl_Command mqEnsembleToken = Tcl_CreateEnsemble(interp, mqNsName, mqNs, 0) ;
if (Tcl_SetEnsembleMappingDict(interp, mqEnsembleToken, mqMap) != TCL_OK) {
    goto release_mq ;
}
/*
 * Export the newly created ensemble command.
 */
if (Tcl_Export(interp, ipcNs, mqStr, 0) != TCL_OK) {
    goto release_ipc ;
}
/*
 * Add the mq ensemble command to the mapping for the posixipc command.
 */
if (Tcl_DictObjPut(interp, ipcMap, Tcl_NewStringObj(mqStr, -1),
        Tcl_NewStringObj(mqNsName, -1)) != TCL_OK) {
    goto release_ipc ;
}
```

Tests

```
<<package initialization tests>>=
test package-commands-2.0.1 {
    Check that posixipc::mq is a namespace ensemble command
} -setup {
} -cleanup {
} -body {
    namespace ensemble exists posixipc::mq
} -result {1}
```

```
<<package initialization tests>>=
test package-commands-2.0.2 {
    Verify the ensemble mapping for the mq command
} -setup {
```

```
} -cleanup {
} -body {
    namespace ensemble configure posixipc::mq -map
} -result {open ::posixipc::mq::open send ::posixipc::mq::send receive  ↩
    ::posixipc::mq::receive unlink ::posixipc::mq::unlink}
```

```
<<package initialization tests>>=
test package-commands-2.0.3 {
    Check that posixipc::mq is exported
} -setup {
} -cleanup {
    namespace forget ::posixipc::mq
} -body {
    namespace import ::posixipc::mq
    namespace origin mq
} -result {::posixipc::mq}
```

Creating the `shm` command

Lastly, we create the `shm` command. It follows the pattern previously established.

```
<<create shm command>>=
<<create shm namespace>>

<<create shm open command>>
<<create shm unlink command>>

<<create shm ensemble command>>
```

shm namespace

```
<<create shm namespace>>=
static char const shmNsName[] = SHM_NS ;
static char const shmStr[] = "shm" ;
Tcl_Namespace *shmNs = Tcl_CreateNamespace(interp, shmNsName, NULL, NULL) ;

Tcl_Obj *shmMap = Tcl_NewDictObj() ;
```

shm open

```
<<create shm open command>>=
static char const shmOpenCmdName[] = SHM_SUBCMD(open) ;

Tcl_CreateObjCommand(interp, shmOpenCmdName, shmOpenCmd, NULL, NULL) ;
if (Tcl_Export(interp, shmNs, openStr, 0) != TCL_OK) {
    goto release_shm ;
}
if (Tcl_DictObjPut(interp, shmMap, Tcl_NewStringObj(openStr, -1),
        Tcl_NewStringObj(shmOpenCmdName, -1)) != TCL_OK) {
    goto release_shm ;
}
```

Tests

```
<<package initialization tests>>=
test package-commands-4.0 {
    Check that posixipc::shm::open was created
} -setup {
} -cleanup {
} -body {
```

```
    info commands ::posixipc::shm::open
} -result {::posixipc::shm::open}
```

shm unlink

```
<<create shm unlink command>>=
static char const shmUnlinkCmdName[] = SHM_SUBCMD(unlink) ;

Tcl_CreateObjCommand(interp, shmUnlinkCmdName, shmUnlinkCmd, NULL, NULL) ;
if (Tcl_Export(interp, shmNs, unlinkStr, 0) != TCL_OK) {
    goto release_shm ;
}
if (Tcl_DictObjPut(interp, shmMap, Tcl_NewStringObj(unlinkStr, -1),
        Tcl_NewStringObj(shmUnlinkCmdName, -1)) != TCL_OK) {
    goto release_shm ;
}
```

Tests

```
<<package initialization tests>>=
test package-commands-4.1 {
    Check that posixipc::shm::unlink was created
} -setup {
} -cleanup {
} -body {
    info commands ::posixipc::shm::unlink
} -result {::posixipc::shm::unlink}
```

shm ensemble

```
<<create shm ensemble command>>=
Tcl_Command shmEnsembleToken = Tcl_CreateEnsemble(interp, shmNsName, shmNs, 0) ;
if (Tcl_SetEnsembleMappingDict(interp, shmEnsembleToken, shmMap) != TCL_OK) {
    goto release_shm ;
}
if (Tcl_Export(interp, ipcNs, shmStr, 0) != TCL_OK) {
    goto release_ipc ;
}
if (Tcl_DictObjPut(interp, ipcMap, Tcl_NewStringObj(shmStr, -1),
        Tcl_NewStringObj(shmNsName, -1)) != TCL_OK) {
    goto release_ipc ;
}
```

Tests

```
<<package initialization tests>>=
test package-commands-4.2.0 {
    Check that posixipc::shm is a namespace ensemble command
} -setup {
} -cleanup {
} -body {
    namespace ensemble exists posixipc::shm
} -result {1}
```

```
<<package initialization tests>>=
test package-commands-4.2.1 {
    Verify the ensemble mapping for the shm command
} -setup {
} -cleanup {
} -body {
    namespace ensemble configure posixipc::shm -map
} -result {open ::posixipc::shm::open unlink ::posixipc::shm::unlink}
```

```
<<package initialization tests>>=
test package-commands-4.2.2 {
    Check that posixipc::shm is exported
} -setup {
} -cleanup {
    namespace forget ::posixipc::shm
} -body {
    namespace import ::posixipc::shm
    namespace origin shm
} -result {::posixipc::shm}
```

Creating the `posixipc` command

Finally, after we have created all the IPC commands and their sub-commands, we can create the top level namespace ensemble command for the package. We install the mapping dictionary that has been accumulated for the two sub-commands in the package ensemble.

```
<<create ipc command ensemble>>=
Tcl_Command ipcCmdToken = Tcl_CreateEnsemble(interp, pkgNamespace, ipcNs, 0) ;
if (Tcl_SetEnsembleMappingDict(interp, ipcCmdToken, ipcMap) != TCL_OK) {
    goto release_ipc ;
}
```

Tests

```
<<package initialization tests>>=
test package-commands-1.0 {
    Check that posixipc is a namespace ensemble command
} -setup {
} -cleanup {
} -body {
    namespace ensemble exists posixipc
} -result {1}
```

Releasing acquired resources on error

We must be careful to clean up any acquired resources in the face of errors. In our case, we must deal with the dictionary objects that are used to accumulate the namespace ensemble command mappings.

Also, in the case of any failure, we just delete the namespace that contains the package level ensemble command and all the children and associated commands will be deleted also.

```
<<ipc release>>=
release_ipc:
    Tcl_DecrRefCount(ipcMap) ;
    Tcl_DeleteNamespace(ipcNs) ;
    return TCL_ERROR ;
```

```
<<mq release>>=
release_mq:
    Tcl_DecrRefCount(mqMap) ;
    goto release_ipc ;
```

```
<<shm release>>=
release_shm:
    Tcl_DecrRefCount(shmMap) ;
    goto release_ipc ;
```

Package Configuration

Tcl supports allowing packages to supply configuration information. Registering the package configuration causes a command to be created that other Tcl code can use to query some properties of the package.

```
<<register package configuration>>=
static char const posixipc_copyright[] =
    "This software is copyrighted 2018 by G. Andrew Mangogna."
    " Terms and conditions for use are distributed with the source code." ;

static Tcl_Config const posixipc_config[] = {
    [0] = {"pkgname", PACKAGE_NAME},
    [1] = {"version", PACKAGE_VERSION},
    [2] = {"copyright", posixipc_copyright},
    {NULL, NULL}
} ;

Tcl_RegisterConfig(interp, posixipc_config[0].value, posixipc_config, "utf-8") ;
```

Tests

```
<<package initialization tests>>=
test package-configuration-1.0 {
    Check that all the configuration info keys are present
} -setup {
} -cleanup {
} -body {
    posixipc::pkgconfig list
} -result {pkgname version copyright}
```

```
<<package initialization tests>>=
test package-configuration-2.0 {
    Check package name from the configuration
} -setup {
} -cleanup {
} -body {
    posixipc::pkgconfig get pkgname
} -result {posixipc}
```

```
<<package initialization tests>>=
test package-configuration-3.0 {
    Check package version from the configuration
} -setup {
} -cleanup {
} -body {
    posixipc::pkgconfig get version
} -result {1.0}
```

```
<<package initialization tests>>=
test package-configuration-4.0 {
    Check package copyright info from the configuration
} -setup {
} -cleanup {
} -body {
    posixipc::pkgconfig get copyright
} -result {This software is copyrighted 2018 by G. Andrew Mangogna.\
Terms and conditions for use are distributed with the source code.}
```

Safe Interpreters

Tcl has the concept of a **safe** interpreter that can be used to run unknown or untrusted code. Since this package deals with I/O using Unix system programming, it is **not** safe for untrusted code. Consequently, the package will not load in a safe interpreter.

```
<<package external function definitions>>=
int
Posixipc_SafeInit(
    Tcl_Interp *interp)
{
    return TCL_ERROR ;      // ❶
}
```

❶ Failing the safe initialization function will prevent the package from being loaded into a safe interpreter.

Package Unloading

Since Tcl 8.5, it is possible to unload a dynamically loaded package.

```
<<package external function definitions>>=
#if TCL_MAJOR_VERSION >= 8 && TCL_MINOR_VERSION >= 5
int
Posixipc_Unload(
    Tcl_Interp *interp)
{
    // HERE! Do we need to close all the IPC mechanisms?
    return TCL_OK ;
}
#endif
```

```
<<package external function definitions>>=
#if TCL_MAJOR_VERSION >= 8 && TCL_MINOR_VERSION >= 5
int
Posixipc_SafeUnload(
    Tcl_Interp *interp)
{
    return TCL_ERROR ;
}
#endif
```

Message Queues

In this section we show the implementation of message queue access in the `posixipc` package. We wish to take advantage of the fact that in most implementations POSIX message queues are implemented in a virtual filesystem and the handle for a message queue is actually a file descriptor. This fact means that message queues fit reasonably into the Tcl virtual channel abstraction. We intend to implement message queue access as a Tcl virtual channel by providing an implementation of a channel driver.

By making a message queue appear as a Tcl channel, we can use all the existing Tcl channel infrastructure. For example (for simplicity, we use the `::chan` command rather than the older style individual channel oriented commands):

- `::chan puts` can be used to place a message onto a message queue.
- `::chan read` or `::chan gets` can be used to obtain a message from a message queue.
- Blocking, buffering, and translation via `::chan configure` operate correctly.
- Event driven I/O via `::chan event` works correctly.

Message Queue Semantics

Of course, POSIX message queue semantics do not exactly match those of Tcl channels. In particular, there are several areas that require further attention.

1. Messages posted to message queues have a priority.
2. Like files, message queues are named and potentially have persistence beyond the lifetime of the creating process. Removing a message queue requires a special action.
3. There is no notion of "end of file" for a message queue. A message queue exist until either:
    a. the system is rebooted or
    b. the message queue is unlinked and all open file descriptors that reference the message queue are closed. This last condition is just like that of an ordinary file.

Message Priority

We will handle message priority in two ways. One of the configuration options for a message queue based channel is a default priority. This can be set via `::chan configure -priority` command and will be used when I/O operations on the message queue come via the Tcl virtual channel interface. The priority of the last received message is also available as a read only configuration option, `-lastpriority`.

However, this arrangement is clumsy for use cases that need frequently to use differing priorities. Consequently, a `mq send` command is also provided which will post a message to a message queue with the priority provided as an argument. We also provide `ms receive` to receive both a message and the priority at which it was posted in one command. Both these commands operate outside of the Tcl channel driver arrangement.

Removing Message Queues

POSIX message queues have names and in the implementation these names show up in the filesystem that is usually mounted somewhere in the file system (on Linux it is typically at `/dev/mqueue`). The names in the filesystem have kernel level persistence and remain until explicitly removed or the system is rebooted. As a convenience, the `mq unlink` command is provided to allow removing message queue entities by the name used to create them. Note is is also possible to remove the message queue using the conventional `::file delete` command, but in that case it is necessary to know the fully qualified path name to the message queue (*i.e.* `/dev/mqueue/name` rather than just `/name`). Depending upon how the system is configured, the message queue file system might be mounted in some place other than `/dev/mqueue`.

Message Queues and Buffering

Messages have distinct boundaries. This is a different semantic than streams of bytes and so our mapping of a message queue to the abstraction of a Tcl channel is not perfect. In particular, there can be an interaction between Tcl channel buffering and buffer sizes and the requirements of a message channel. Two characteristics of message queues need to be understood:

1. Each low level read from a message queue requires a buffer that is at least as big as the maximum message size. To avoid I/O errors when reading, we must make sure the buffer size is adequate to hold the maximum size that a message might be and the maximum message size is determined when the message queue is created.
2. Each low level write to a message channel constitutes a message and both the channel buffering mode and the channel buffer size can affect when low level writes happen.

To help in handling the buffering, opening a message queue channel automatically sets the channel buffer size to be the same size as the messages for the message queue. Note the buffer size is independent of the buffering mode.

The following sections, we show some example scripts that demonstrate the interactions between Tcl channel buffering properties and POSIX message queue. The scripts perform double duty. Not only are they examples, but we use them as test cases for the package tests.

Simple text example

In this example, we demonstrate a simple client / server arrangement where plain text messages are passed between them. We want one text record to constitute a message. The client simply runs `eval` on whatever it receives. This gives the server complete control over the execution of the client.

In the body of the test, the server requests the client to echo three text records that consist of just a simple decimal number. The server receives the echoed response and accumulates a total of the received numbers. After the arithmetic, the client is commanded to break out of its loop resulting in the client exiting.

```
<<mq examples>>=
test text-record-messages-1.0 {
    Pass simple text records as messages.
} -body {
    set total 0
    foreach num {1 2 3} {
        chan puts $clientChan "chan puts $num"
        set response [gets $serverChan]
        log::debug "server: received, \"$response\""

        set total [expr {$total + $response}]
    }

    chan puts $clientChan break
    return $total
} -setup {
    set serverQueue /mqtest_txt_server
    <<trm1: create server message queue>>

    set clientQueue /mqtest_txt_client
    <<trm1: create client message queue>>

    <<trm1: create client script file>>

    <<trm1: run client script>>
} -cleanup {
    chan close $serverChan
    chan close $clientChan

    mq unlink $clientQueue
    mq unlink $serverQueue
} -result {6}
```

The test setup has more interesting details. First, we create a message queue for the server. The server will only read from it. The `EXCL` flag is used to insure that the queue doesn't exist when it is created. This insures things got cleaned up properly from the last test run. The `-buffering line` buffer mode means that each completed text record is immediately sent as a message.

```
<<trm1: create server message queue>>=
set serverChan [mq open $serverQueue {CREAT EXCL RDONLY}]
chan configure $serverChan -buffering line
log::debug "server queue, \"$serverQueue\", open"
```

Similarly, we create the client message queue. From the server's perspective, the queue is write only for sending commands to the client.

```
<<trm1: create client message queue>>=
set clientChan [mq open $clientQueue {CREAT EXCL WRONLY}]
chan configure $clientChan -buffering line
log::debug "client queue, \"$clientQueue\", open"
```

The client script must also open both queues. But from the clients perspective, the client queue is only read and the server queue is only written. After getting everything opened, the client loops evaluating each record it receives from the server.

```
<<trm1: create client script file>>=
makeFile {
    if {$::tcl_platform(os) eq "Linux"} {
        load ../tea/build/x86_64-linux-tcl8.6/libposixipc1.0.so
    } else {
        package require posixipc
    }

    namespace import ::posixipc::mq

    set clientQueue /mqtest_txt_client
    set clientChan [mq open $clientQueue RDONLY]
    chan configure $clientChan -buffering line

    set serverQueue /mqtest_txt_server
    chan close stdout                                   ; # ❶
    set serverChan [mq open $serverQueue WRONLY]
    chan configure $serverChan -buffering line

    while {[gets $clientChan line] != -1} {
        try {
            eval $line
        } on error {result} {
            chan puts "error: $result"
            break
        }
    }

    chan close $serverChan
    chan close $clientChan
} txt-client.tcl
```

❶ This is a bit of trickery here. By closing `stdout` before opening the server queue, the Tcl channel system will insure that the newly open channel is used as `stdout`. This behavior follows the usual behavior for UNIX file descriptors. UNIX file descriptors are assigned the lowest number possible. This behavior of taking over the `stdout` channel since it was closed means the server can send a `chan puts` command without a channel specifier and insure that the response is echoed back to the server's message queue.

The client is run in the background. The server sends an `break` command to force the client to exit.

```
<<trm1: run client script>>=
exec tclsh txt-client.tcl &
log::debug "client started in the background"
```

Multi-record text example

This example is an extension of the previous example. Rather than each message being a single text record, we will group multiple records together into a message. You can think of the multiple records as a type of message header.

We can accomplish sending multiple records in a message by changing the `-buffering` option to `full` and then performing a `flush` when we have buffered all the records for the message. On the receiving side, we can still use `line` buffering, since the Tcl channel code will read the message as a single block and then parcel it out line by line. So on the server side we set up the server message queue channel with `line` buffering and the client message queue channel with `full` buffering. On the client side, we invert the sense, setting the client message queue channel to `line` buffering and the server message queue channel to `full` buffering.

```
<<mq examples>>=
test text-record-messages-2.0 {
```

```
    Pass multiple text records as a message.
} -body {
    foreach num {3 4 5} {
        chan puts $clientChan "chan puts $num"
    }
    chan puts $clientChan break ; # causes client to break out of its loop
    chan flush $clientChan

    set total 0
    for {set i 0} {$i < 3} {incr i} {
        set response [gets $serverChan]
        log::debug "server: received, \"$response\""
        set total [expr {$total + $response}]
    }

    return $total
} -setup {
    set serverQueue /mqtest_mtxt_server
    <<trm2: create server message queue>>

    set clientQueue /mqtest_mtxt_client
    <<trm2: create client message queue>>

    <<trm2: create client script file>>

    <<trm2: run client script>>
} -cleanup {
    chan close $serverChan
    chan close $clientChan

    mq unlink $clientQueue
    mq unlink $serverQueue
} -result {12}
```

On the server side, the server channel is set to `line` buffering. This makes it easy for the server to read each record in the message as a single line. This happens despite each message containing multiple text records.

```
<<trm2: create server message queue>>=
set serverChan [mq open $serverQueue {CREAT EXCL RDONLY}]
chan configure $serverChan -buffering line
log::debug "server queue, \"$serverQueue\", open"
```

Still on the server side, we make the client channel `full` buffering causing the Tcl channel place multiple text records in the same message.

```
<<trm2: create client message queue>>=
set clientChan [mq open $clientQueue {CREAT EXCL WRONLY}]
chan configure $clientChan -buffering full
log::debug "client queue, \"$clientQueue\", open"
```

The client side inverts the sense of the buffering of the two channels. This allows the client to read and evaluate each text record on a line by line basis. However, the response to the server are batched up.

```
<<trm2: create client script file>>=
makeFile {
    if {$::tcl_platform(os) eq "Linux"} {
        load ../tea/build/x86_64-linux-tcl8.6/libposixipc1.0.so
    } else {
        package require posixipc
    }

    namespace import ::posixipc::mq
```

```
    set clientQueue /mqtest_mtxt_client
    set clientChan [mq open $clientQueue RDONLY]
    chan configure $clientChan -buffering line

    set serverQueue /mqtest_mtxt_server
    chan close stdout                                  ; # ❶
    set serverChan [mq open $serverQueue WRONLY]
    chan configure $serverChan -buffering full

    while {[gets $clientChan line] > 0} {
        try {
            eval $line
        } on error {result} {
            chan puts "error: $result"
            break
        }
    }
    chan flush stdout    ; # ❷

    chan close $serverChan
    chan close $clientChan
} mtxt-client.tcl
```

❶ This is the same bit of trickery we used before.

❷ The `flush` causes the multiple, buffered responses to be sent as a single message back to the server.

```
<<trm2: run client script>>=
exec tclsh mtxt-client.tcl &
log::debug "client started in the background"
```

Event driven text example

This example is like the simple single record in a message example, except we run the client using event driven I/O.

```
<<mq examples>>=
test text-record-messages-3.0 {
    Pass simple text records as messages.
} -body {
    set total 0
    foreach num {7 8 9} {
        chan puts $clientChan "chan puts $num"
        set response [gets $serverChan]
        log::debug "server: received, \"$response\""

        set total [expr {$total + $response}]
    }

    chan puts $clientChan exit  ; # ❶
    return $total
} -setup {
    set serverQueue /mqtest_edtxt_server
    <<trm3: create server message queue>>

    set clientQueue /mqtest_edtxt_client
    <<trm3: create client message queue>>

    <<trm3: create client script file>>
```

```
    <<trm3: run client script>>
} -cleanup {
    chan close $serverChan
    chan close $clientChan

    mq unlink $clientQueue
    mq unlink $serverQueue
} -result {24}
```

❶ Since the client is event driven and not running a loop, we require an exit command to terminate.

We set up the server side just as before using `line` buffering.

```
<<trm3: create server message queue>>=
set serverChan [mq open $serverQueue {CREAT EXCL RDONLY}]
chan configure $serverChan -buffering line
log::debug "server queue, \"$serverQueue\", open"
```

```
<<trm3: create client message queue>>=
set clientChan [mq open $clientQueue {CREAT EXCL WRONLY}]
chan configure $clientChan -buffering line
log::debug "client queue, \"$clientQueue\", open"
```

The client also uses `line` buffering. However, the `while` loop is now gone and is replaced by a file event which triggers a `proc` to evaluate the line and interact with the server.

```
<<trm3: create client script file>>=
makeFile {
    if {$::tcl_platform(os) eq "Linux"} {
        load ../tea/build/x86_64-linux-tcl8.6/libposixipc1.0.so
    } else {
        package require posixipc
    }

    namespace import ::posixipc::mq

    set clientQueue /mqtest_edtxt_client
    set clientChan [mq open $clientQueue RDONLY]
    chan configure $clientChan -buffering line -blocking false

    set serverQueue /mqtest_edtxt_server
    chan close stdout
    set serverChan [mq open $serverQueue WRONLY]
    chan configure $serverChan -buffering line

    proc ::readMsg {mqchan} {
        set line [chan gets $mqchan]
        try {
            eval $line
        } on error {result} {
            chan puts "error: $result"
            return -code error $result
        }
    }

    chan event $clientChan readable [list ::readMsg $clientChan]
    vwait forever
} ed-txt-client.tcl
```

The `chan event` sets up the channel so that `::readMsg` is invoked when a message arrives. Of course, the event loop must be entered via `vwait` to process the I/O events.

```
<<trm3: run client script>>=
exec tclsh ed-txt-client.tcl &
log::debug "client started in the background"
```

Message Queue Channel Driver

A Tcl channel driver consists of a specific set of functions that match a fixed API and a Tcl level command to open or create the underlying channel. In what follows, we present the Tcl channel driver functions first, followed by the Tcl level commands.

Channel Driver Components

In this section, we describe the functions supplied for the Tcl channel driver that is used for POSIX message queues. The concept behind a channel driver is to delegate specific operations to supplied functions so that the generic Tcl channel code can support a variety of channel types. Whenever the generic code needs a type-specific operation, it invokes it via a table of function pointers supplied when the channel was created.

The following is an initialized variable of the proper structure that is given to `Tcl_CreateChannel()` to define the message queue operations.

```
<<mq static data>>=
static Tcl_ChannelType MqChannelType = {
    .typeName = "mq",           /* The name of the channel type in Tcl
                                 * commands. This storage is owned by channel
                                 * type. */
    .version = TCL_CHANNEL_VERSION_5,
                                /* Version of the channel type. */
    .closeProc = mqCloseProc,
                                /* Function to call to close the channel, or
                                 * TCL_CLOSE2PROC if the close2Proc should be
                                 * used instead. */
    .inputProc = mqInputProc,
                                /* Function to call for input on channel. */
    .outputProc = mqOutputProc,
                                /* Function to call for output on channel. */
    .seekProc = NULL,
                                /* Function to call to seek on the channel.
                                 * May be NULL. */
    .setOptionProc = mqSetOptionProc,
                                /* Set an option on a channel. */
    .getOptionProc = mqGetOptionProc,
                                /* Get an option from a channel. */
    .watchProc = mqWatchProc,
                                /* Set up the notifier to watch for events on
                                 * this channel. */
    .getHandleProc = mqGetHandleProc,
                                /* Get an OS handle from the channel or NULL
                                 * if not supported. */
    .close2Proc = NULL,
                                /* Function to call to close the channel if
                                 * the device supports closing the read &
                                 * write sides independently. */
    .blockModeProc = mqBlockModeProc,
                                /* Set blocking mode for the raw channel. May
                                 * be NULL. */
    /*
     * Only valid in TCL_CHANNEL_VERSION_2 channels or later
```

```
     */
    .flushProc = NULL,
                                /* Function to call to flush a channel. May be
                                 * NULL. */
    .handlerProc = NULL,
                                /* Function to call to handle a channel event.
                                 * This will be passed up the stacked channel
                                 * chain. */
    /*
     * Only valid in TCL_CHANNEL_VERSION_3 channels or later
     */
    .wideSeekProc = NULL,
                                /* Function to call to seek on the channel
                                 * which can handle 64-bit offsets. May be
                                 * NULL, and must be NULL if seekProc is
                                 * NULL. */
    /*
     * Only valid in TCL_CHANNEL_VERSION_4 channels or later
     * TIP #218, Channel Thread Actions
     */
    .threadActionProc = NULL,
                                /* Function to call to notify the driver of
                                 * thread specific activity for a channel. May
                                 * be NULL. */

    /*
     * Only valid in TCL_CHANNEL_VERSION_5 channels or later
     * TIP #208, File Truncation
     */
    .truncateProc = NULL
                                /* Function to call to truncate the underlying
                                 * file to a particular length. May be NULL if
                                 * the channel does not support truncation. */
} ;
```

Notice that not all functions are provided (*i.e.* some members of the structure are set to `NULL`). Those members are:

seekProc
: It is not possible to seek on a message queue. Messages have discrete boundaries and cannot be arbitrarily indexed in the same way that a stream of bytes can.

close2Proc
: It is not possible to close read and write directions independently for a message queue.

flushProc
: This is reserved.

handlerProc
: Used only for stacked drivers.

wideSeekProc
: As previously, seeking is not an allowed operation on a message queue.

threadActionProc
: There is no thread specific driver state.

truncateProc
: The concept of truncation does not have a parallel for a message queue. The maximum number of messages allowed to be queued can only be set when the message queue is created.

When a new Tcl channel is created, some instance data for the channel may also be supplied. That instance data is then provided as an argument to each of the functions for the driver. This makes things much more convenient for the driver function code

to have access to the state of the channel. The following data structure defines the information that constitutes the state of an individual message queue channel.

```
<<mq data types>>=
typedef struct {
    Tcl_Channel channel ;       // ❶
    mqd_t mqdes ;               // ❷
    int tclMask ;               // ❸
    unsigned lastpriority ;     // ❹
    unsigned defpriority ;      // ❺
} MqState ;
```

❶ The Tcl channel handle as returned from `Tcl_CreateChannel()`.

❷ The message queue descriptor. In reality this is an ordinary file descriptor.

❸ A mask of allowed operations on the channel. This is combination of TCL_READABLE or TCL_WRITABLE.

❹ The priority of the last received message from the queue.

❺ The priority to use to post a message to a queue.

There turns out to be a small difference in the way Linux and FreeBSD hand a value of type `mqd_t`. In Linux, a value of type `mqd_t` is just an ordinary file descriptor. In FreeBSD, a value of type `mqd_t` is a pointer to the file descriptor. So we will use a small inline function to handle the conversion of `mqd_t` values to file descriptors.

```
<<static inline function definitions>>=
static inline int
mqdToFd(mqd_t mq)
{
#if defined(__linux__)
    return (int)mq ;
#elif defined(__FreeBSD__)
    return *(int *)mq ;
# endif
}
```

The following sections describe each of the functions supplied as part of the message queue channel driver.

Block mode procedure

```
<<mq forward function declarations>>=
static int mqBlockModeProc(ClientData instanceData, int mode) ;
```

The blocking mode of a message queue can be set as one of its attributes using `mq_setattr()`.

```
<<mq static function definitions>>=
static int
mqBlockModeProc(
    ClientData instanceData,
    int mode)
{
    struct mq_attr newattrs = {
        .mq_flags = mode == TCL_MODE_NONBLOCKING ? O_NONBLOCK : 0,   // ❶
        .mq_maxmsg = 0,
        .mq_msgsize = 0,
        .mq_curmsgs = 0,
    } ;

    MqState *mqPtr = instanceData ;
```

```
    int err = mq_setattr(mqPtr->mqdes, &newattrs, NULL) ;
    return err < 0 ? errno : err ;
}
```

❶ The only member in `mq_attr` that has any effect when calling `mq_setattr()` is `mq_flags`. All the other members are ignored, per the man page.

Close procedure

```
<<mq forward function declarations>>=
static int mqCloseProc(ClientData instanceData, Tcl_Interp *interp) ;
```

The close procedure is simply a thin veneer on `mq_close()`. However, there is some other clean up required.

```
<<mq static function definitions>>=
static int
mqCloseProc(
    ClientData instanceData,
    Tcl_Interp *interp)
{
    MqState *mqPtr = instanceData ;

    Tcl_DeleteFileHandler(mqdToFd(mqPtr->mqdes)) ;    // ❶

    int err = mq_close(mqPtr->mqdes) ;
    ckfree(mqPtr) ;                          // ❷

    return err < 0 ? errno : err ;
}
```

❶ Closing the message queue implies any handlers are also deleted.

❷ Deallocate the state information as it is not longer required.

Input procedure

```
<<mq forward function declarations>>=
static int mqInputProc(ClientData instanceData, char *buf, int bufSize,
        int *errorCodePtr) ;
```

Input from a message queue is obtained by calling `mq_receive()`. This procedure simply bridges between the different API conventions.

```
<<mq static function definitions>>=
static int
mqInputProc(
    ClientData instanceData,
    char *buf,
    int bufSize,
    int *errorCodePtr)
{
    MqState *mqData = instanceData ;

    int read = mq_receive(mqData->mqdes, buf, bufSize, &mqData->lastpriority) ; // ❶
    if (read == -1) {
        *errorCodePtr = errno ;
    }
    return read ;
}
```

❶ We save the priority of the last message so it can be queried later if desired.

Tests

```
<<mq tests>>=
test mq-input-1.0 {
    Read in non-blocking mode when queue is empty
} -setup {
    set queueName /mq-input-1.0
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    set mqchan [mq open $queueName {CREAT EXCL RDWR NONBLOCK}]
    log::debug "mq channel = $mqchan"
    set nread [chan read $mqchan 1]
    log::debug "nread, \"$nread\""
    log::debug "nread length = [string length $nread]"
    set eof [chan eof $mqchan]
    log::debug "eof = $eof"
    return [expr {($nread eq {}) && !$eof}]
} -result {1}
```

Output procedure

```
<<mq forward function declarations>>=
static int mqOutputProc(ClientData instanceData, char const *buf, int toWrite,
        int *errorCodePtr) ;
```

Like the input procedure, the output procedure's main function is simply to bridge between the message queue system call and the Tcl API requirements.

```
<<mq static function definitions>>=
static int
mqOutputProc(
    ClientData instanceData,
    char const *buf,
    int toWrite,
    int *errorCodePtr)
{
    MqState *mqPtr = instanceData ;

    int err = mq_send(mqPtr->mqdes, buf, toWrite, mqPtr->defpriority) ; // ❶
    if (err == -1) {
        *errorCodePtr = errno ;
        return -1 ;
    }
    return toWrite ;
}
```

❶ Note that the `mq_send()` manual page states that 0 length messages are allowed. Also, since we are in the channel driver, we supply the message priority as the default priority.

Tests

```
<<mq tests>>=
test mq-output-1.0 {
    Write in non-blocking mode when queue is full
} -setup {
```

```
    set queueName /mq-output-1.0
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    set mqchan [mq open -maxmsg 1 $queueName {CREAT EXCL RDWR NONBLOCK}]
    log::debug "mq channel = $mqchan"
    chan configure $mqchan -buffering line
    chan puts $mqchan "Hello 1"
    set second "Hello 2"
    chan puts $mqchan $second
    set pending [chan pending output $mqchan]
    return [expr {$pending == [string length $second] + 1}] ; # +1 for newline
} -result {1}
```

Configuration options access support

Before we can get to the procedures that get and set configuration options on a message queue, we build some supporting data structures to look up option names. The message queue options fall into two categories:

1. Options that can only be set on open.
2. Options that can be set at any time.

Note that all options may be read. The restrictions only apply to setting options and those restrictions arise from the manner in which the underlying POSIX messages queue functions are designed.

It is useful to have a symbolic encoding of the option names in addition to the strings that specify the options. However, we must be careful about this encoding is created. We want increasing sequential numerical order to match the increasing alphabetical order of the option names.

```
<<mq data types>>=
typedef enum {
    Mq_curmsgs_opt = 0,
    Mq_lastpriority_opt,
    Mq_maxmsg_opt,
    Mq_msgsize_opt,
    Mq_priority_opt,

    Mq_OPTION_COUNT             // must be last
} MqOptionType ;
```

We build a table of options that we can use to look up option names and determine their characteristics.

```
<<mq static data>>=
struct MqConfigOpt {
    char const *optName ;
    bool setOnOpen ;
    bool setOnConfig ;
} const mqOptions[Mq_OPTION_COUNT] = {  // Ordered by ascending option name!
    [Mq_curmsgs_opt] = {
        .optName = "-curmsgs",
        .setOnOpen = false,
        .setOnConfig = false,
    },
    [Mq_lastpriority_opt] = {
        .optName = "-lastpriority",
        .setOnOpen = false,
        .setOnConfig = false,
    },
```

```
    [Mq_maxmsg_opt] = {
        .optName = "-maxmsg",
        .setOnOpen = true,
        .setOnConfig = false,
    },
    [Mq_msgsize_opt] = {
        .optName = "-msgsize",
        .setOnOpen = true,
        .setOnConfig = false,
    },
    [Mq_priority_opt] = {
        .optName = "-priority",
        .setOnOpen = true,
        .setOnConfig = true,
    },
} ;
```

Note we have purposely ordered the options in ascending alphabetical order since we intend to use a binary search for lookup.

It is also convenient to have a string that is just a list of the options without any leading dash (-) characters. This is used by Tcl function for constructing error messages in the case of a bad channel option. Although in theory we could construct such a list at run time from the above data, it is simpler to just have the list ready at hand and not have the complication of determining how much memory is required to hold the list. Also, placing the list lexically close to the above options array may help prevent mistakes in the future if more options are added. Well, we can at least hope.

```
<<mq static data>>=
static char const mqOptionsList[] =
        "curmsgs lastpriority maxmsg msgsize priority" ;
```

We need a function to look up an option and return the information we have about the option.

```
<<mq forward function declarations>>=
static struct MqConfigOpt const *lookUpMqOpt(char const *optionName) ;
```

The `lookUpMqOpt()` function is just a thin wrapper on the standard library `bsearch()` function.

```
<<mq static function definitions>>=
static struct MqConfigOpt const *
lookUpMqOpt(
    char const *optionName)
{
    struct MqConfigOpt key = {
        .optName = optionName
    } ;

    return bsearch(&key, mqOptions, COUNTOF(mqOptions),
            sizeof(struct MqConfigOpt), mqOptCompare) ;
}
```

We must supply the comparison function needed by `bsearch()`

```
<<mq forward function declarations>>=
static int mqOptCompare(void const *m1, void const *m2) ;
```

The comparison is based on option names, so the standard library `strcmp()` function has the right characteristics.

```
<<mq static function definitions>>=
static int
mqOptCompare(
    void const *m1,
    void const *m2)
{
```

```
    struct MqConfigOpt const *o1 = m1 ;
    struct MqConfigOpt const *o2 = m2 ;

    return strcmp(o1->optName, o2->optName) ;
}
```

Get option procedure

```
<<mq forward function declarations>>=
static int mqGetOptionProc(ClientData instanceData, Tcl_Interp *interp,
        char const *optionName, Tcl_DString *optionValue) ;
```

The get option procedure has a complication associated with the way option names are specified. If the *optionName* argument is `NULL`, then this implies that an alternating list of all options and their values are to be returned. Otherwise, we are returning a single option value.

```
<<mq static function definitions>>=
static int
mqGetOptionProc(
    ClientData instanceData,
    Tcl_Interp *interp,
    char const *optionName,
    Tcl_DString *optionValue)
{
    MqState *mqPtr = instanceData ;

    if (optionName != NULL) {
        <<mqGetOptionProc: get single option>>
    } else {
        <<mqGetOptionProc: get all options>>
    }
    return TCL_OK ;
}
```

In the case of a single option, we must look up the option name and format the option value as a string. The value is returned in the Tcl dynamic string provided as an argument.

Getting a single option

```
<<mqGetOptionProc: get single option>>=
<<mqGetOptionProc: look up option>>
<<mqGetOptionProc: format option value>>

Tcl_DStringAppendElement(optionValue, buf) ;
```

The option is looked up by name using the lookUpMqOpt() function we described previously. A `NULL` return on the look up implies that the option was not found. There is a specific Tcl function provided to channel drivers to report bad channel options.

```
<<mqGetOptionProc: look up option>>=
struct MqConfigOpt const *optProp = lookUpMqOpt(optionName) ;

if (optProp == NULL) {
    return Tcl_BadChannelOption(interp, optionName, mqOptionsList) ;
}
```

At this point, we have a good option name and we need to fetch the value of the option and format it. There is specific code that needs to be executed for each different option. Rather than compare the option names all over again, the following design computes the array index into the `mqOptions` array. The array index is then equated to a value from the `MqOptionType` enumeration. That enumeration declaration has been purposely contrived to match the array index of the corresponding option. By that contrivence, we have a convenient symbolic name to use in the code. The array index encoded as an enumeration can then be used in a `switch` statement to locate the option specific code that fetches and formats the option value.

```
<<mqGetOptionProc: format option value>>=
struct mq_attr attrs ;                      // retrieved mq attributes
char buf[TCL_INTEGER_SPACE + 16] ;          // formated option values

static char const getAttrFailedMsg[] = "get option: mq_getattr() failed" ;

MqOptionType whichOpt = (MqOptionType)(optProp - mqOptions) ;
switch (whichOpt) {
    case Mq_curmsgs_opt:
    {
        <<mqGetOptionProc: format curmsgs value>>
    }
        break ;
    case Mq_lastpriority_opt:
    {
        <<mqGetOptionProc: format lastpriority value>>
    }
        break ;
    case Mq_maxmsg_opt:
    {
        <<mqGetOptionProc: format maxmsg value>>
    }
        break ;
    case Mq_msgsize_opt:
    {
        <<mqGetOptionProc: format msgsize value>>
    }
        break ;
    case Mq_priority_opt:
    {
        <<mqGetOptionProc: format priority value>>
    }
        break ;
    default:        // ❶
        Tcl_Panic("posixipc mq: get option failure: found option named, "
                "\"%s\", but no code to process the option", optionName) ;
        break ;
}
```

❶ The default case protects against the circumstance where additions are made to the `mqOptions` array, but the `enum` definition is not modified to account for the addition. Of course, any changes in the `mqOptions` array must be reflected in the code here since we have tied finding the correct code for formating an option to the layout (*i.e.* the array indices) of the `mqOptions` array.

Now it is just a matter of finding and formatting the options values for each particular option. Some option values are located in the state data passed in and some option values are retrieved by making calls to `mq_getattr()`. Each case sorts things out as necessary.

curmsgs formatting

```
<<mqGetOptionProc: format curmsgs value>>=
int err = mq_getattr(mqPtr->mqdes, &attrs) ;
if (err < 0) {
    return mqFailure(interp, getAttrFailedMsg) ;
}
snprintf(buf, sizeof(buf), "%ld", attrs.mq_curmsgs) ;
```

lastpriority formatting

```
<<mqGetOptionProc: format lastpriority value>>=
snprintf(buf, sizeof(buf), "%u", mqPtr->lastpriority) ;
```

maxmsg formatting

```
<<mqGetOptionProc: format maxmsg value>>=
int err = mq_getattr(mqPtr->mqdes, &attrs) ;
if (err < 0) {
    return mqFailure(interp, getAttrFailedMsg) ;
}
snprintf(buf, sizeof(buf), "%ld", attrs.mq_maxmsg) ;
```

msgsize formatting

```
<<mqGetOptionProc: format msgsize value>>=
int err = mq_getattr(mqPtr->mqdes, &attrs) ;
if (err < 0) {
    return mqFailure(interp, getAttrFailedMsg) ;
}
snprintf(buf, sizeof(buf), "%ld", attrs.mq_msgsize) ;
```

priority formatting

```
<<mqGetOptionProc: format priority value>>=
snprintf(buf, sizeof(buf), "%u", mqPtr->defpriority) ;
```

Tests

```
<<mq tests>>=
test mq-get-option-1.0 {
    Get message queue options one at a time
} -setup {
    set queueName /mq-get-option-1.0
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    set mqchan [mq open $queueName w+]
    log::debug "mq channel = $mqchan"

    set optionNames {-curmsgs -lastpriority -maxmsg -msgsize -priority}
    foreach opt $optionNames {
        set optValue($opt) [chan configure $mqchan $opt]
        log::debug "$opt = $optValue($opt)"
    }

    array set optExpected {
        -curmsgs 0
        -lastpriority 0
        -maxmsg 10
        -msgsize 8192
        -priority 0
    }

    foreach opt $optionNames {
        if {$optValue($opt) != $optExpected($opt)} {
            return 0 ;
        }
    }
    return 1
} -result {1}
```

```
<<mq tests>>=
test mq-get-option-2.0 {
    Bad option name
```

```
} -setup {
    set queueName /mq-get-option-2.0
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    set mqchan [mq open $queueName w+]
    log::debug "mq channel = $mqchan"

    chan configure $mqchan -foo
} -result {bad option "-foo": should be one of -blocking, -buffering,\
        -buffersize, -encoding, -eofchar, -translation, -curmsgs,\
        -lastpriority, -maxmsg, -msgsize, or -priority}\
    -returnCodes error
```

In the case of gettting all the options with a single command, the end result must be a string formatted as an alternating list of option name / option value pairs. As with the single option case, the string is returned via the Tcl dynamic string, `optionValue`, passed in as an argument. Since we are getting all the options, we can start by getting the message queue attributes available in the system call, `mq_getattr()`. Then the resulting list can be constructed.

Getting all options

```
<<mqGetOptionProc: get all options>>=
<<mqGetOptionProc: get mq attributes>>
<<mqGetOptionProc: format all option name/value pairs>>
```

Getting message queue options

```
<<mqGetOptionProc: get mq attributes>>=
struct mq_attr attrs ;
int err = mq_getattr(mqPtr->mqdes, &attrs) ;
if (err < 0) {
    return mqFailure(interp, "get option: mq_getattr() failed") ;
}
```

Failed system calls

In the last code chunk, we had to admit to the possiblity that a system call (`mq_getattr()` in this case) might fail. We want to put together a common scheme to handle such failures. Tcl want to have `errno` set properly and it is useful to set the result of the command to a meaninful error string. The `ipcFailure()` function below handles the error.

```
<<mq forward function declarations>>=
static int ipcFailure(Tcl_Interp *interp, char const *type, char const *msg) ;
```

```
<<mq static function definitions>>=
static int
ipcFailure(
    Tcl_Interp *interp,
    char const *type,
    char const *msg)
{
    Tcl_SetErrno(errno) ;
    Tcl_Obj *errObj = Tcl_ObjPrintf("posixipc %s: %s: %s", type, msg,
            Tcl_PosixError(interp)) ;
    Tcl_SetObjResult(interp, errObj) ;
    return TCL_ERROR ;                  // ❶
}
```

❶ This is just a convenient, allowing code to return the value of `ipcFailure()` as an error.

Since it is a system call that has failed, we need a reasonable error message and must make sure to set Tcl's notion of `errno` indicating the cause of the error.

We specialize the failure function for message queues. We will see an additional specialization for shared memory.

```
<<mq forward function declarations>>=
static int mqFailure(Tcl_Interp *interp, char const *msg) ;
```

```
<<mq static function definitions>>=
static int
mqFailure(
    Tcl_Interp *interp,
    char const *msg)
{
    return ipcFailure(interp, "mq", msg) ;
}
```

Formatting all the options as name / value pairs is just a long sequence of fetching the option value, formatting it and then adding it as a list element to the returned string. We obtain the option name from the `mqOptions` array, since we already have the name stored there. We simply march down the array and its order definition determines the code we need to run to do the formatting.

```
<<mqGetOptionProc: format all option name/value pairs>>=
char buf[TCL_INTEGER_SPACE + 16] ;      // buffer space for the formatting
struct MqConfigOpt const *optProp = mqOptions ;

<<mqGetOptionProc: format curmsgs name/value pair>>
<<mqGetOptionProc: format lastpriority name/value pair>>
<<mqGetOptionProc: format maxmsg name/value pair>>
<<mqGetOptionProc: format msgsize name/value pair>>
<<mqGetOptionProc: format priority name/value pair>>
```

So, each option has a small piece of code required to add the option name to the result, fetch the option value, format the value into a string and the add the value string to the accumulated result.

curmsgs formatting

```
<<mqGetOptionProc: format curmsgs name/value pair>>=
Tcl_DStringAppendElement(optionValue, optProp->optName) ;
optProp++ ;                                                    // ❶
snprintf(buf, sizeof(buf), "%ld", attrs.mq_curmsgs) ;
Tcl_DStringAppendElement(optionValue, buf) ;
```

❶ After obtaining the name, we point to the next option in the array, setting up for the next formatting code.

lastpriority formatting

```
<<mqGetOptionProc: format lastpriority name/value pair>>=
Tcl_DStringAppendElement(optionValue, optProp->optName) ;
optProp++ ;
snprintf(buf, sizeof(buf), "%u", mqPtr->lastpriority) ;
Tcl_DStringAppendElement(optionValue, buf) ;
```

maxmsg formatting

```
<<mqGetOptionProc: format maxmsg name/value pair>>=
Tcl_DStringAppendElement(optionValue, optProp->optName) ;
optProp++ ;
snprintf(buf, sizeof(buf), "%ld", attrs.mq_maxmsg) ;
Tcl_DStringAppendElement(optionValue, buf) ;
```

msgsize formatting

```
<<mqGetOptionProc: format msgsize name/value pair>>=
Tcl_DStringAppendElement(optionValue, optProp->optName) ;
optProp++ ;
snprintf(buf, sizeof(buf), "%ld", attrs.mq_msgsize) ;
Tcl_DStringAppendElement(optionValue, buf) ;
```

priority formatting

```
<<mqGetOptionProc: format priority name/value pair>>=
Tcl_DStringAppendElement(optionValue, optProp->optName) ;
optProp++ ;                                                    // ❶
snprintf(buf, sizeof(buf), "%u", mqPtr->defpriority) ;
Tcl_DStringAppendElement(optionValue, buf) ;
```

❶ Not strictly necessary since this is the last option. But consistency and the possibility of future additions are worth a single code statement.

Tests

```
<<mq tests>>=
test mq-get-option-3.0 {
    Get all options for a message queue in one command
} -setup {
    set queueName /mq-get-option-3.0
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    set mqchan [mq open $queueName w+]
    log::debug "mq channel = $mqchan"

    set opts [chan configure $mqchan]
    log::debug "option = $opts"
    return [dict get $opts -msgsize]
} -result {8192}
```

Set option procedure

```
<<mq forward function declarations>>=
static int mqSetOptionProc(ClientData instanceData, Tcl_Interp *interp,
        char const *optionName, char const *newValue) ;
```

Because of restrictions on when options can be set, there is a bit more logic here. First, we check if we can look up the option and therefore it is a good, known option. We next make sure that it is an option that can be set as part of a configuration call. Finally, we can convert value string to internal representation and set the option value.

```
<<mq static function definitions>>=
static int
mqSetOptionProc(
    ClientData instanceData,
    Tcl_Interp *interp,
    char const *optionName,
    char const *newValue)
{
    MqState *mqPtr = instanceData ;

    struct MqConfigOpt const *optProp = lookUpMqOpt(optionName) ;
    if (optProp == NULL) {
```

```
        return Tcl_BadChannelOption(interp, optionName, mqOptionsList) ;
    } else if (!optProp->setOnConfig) {
        Tcl_Obj *errObj = Tcl_ObjPrintf("posixipc mq: set option: "
                "%s option may not be set by configuration", optionName) ;
        Tcl_SetObjResult(interp, errObj) ;
        return TCL_ERROR ;
    } else {
        <<mqSetOptionProc: set option to new value>>
    }

    return TCL_OK ;
}
```

Like in the case of `mqGetOptionProc()`, we must execute option specific code to set the option value. We use the same design strategy here, namely, computing the array index into the `mqOptions` array as an integer encoding of the option and use an enumeration value that has the same integer values. A switch statement can then take us to the option specific code. As it turns out, there is only one option that may be set in a configure command. We leave it as a switch statement for future updates.

```
<<mqSetOptionProc: set option to new value>>=
MqOptionType whichOpt = (MqOptionType)(optProp - mqOptions) ;
switch (whichOpt) {
    case Mq_priority_opt:
    {
        <<mqSetOptionProc: set priority option>>
    }
        break ;

    default:
        Tcl_Panic("posixipc mq: set option failure: found option named, "
                "\"%s\", but no code to process the option", optionName) ;
        break ;
}
```

Message priorities are unsigned numbers. Tcl provides convenience functions to convert strings to numeric types. We need only make sure things are positive.

```
<<mqSetOptionProc: set priority option>>=
int priority ;
if (Tcl_GetInt(interp, newValue, &priority) != TCL_OK) {
    return TCL_ERROR ;
}
if (priority < 0) {
    Tcl_Obj *errObj = Tcl_ObjPrintf("posixipc mq: set option: "
            "invalid negative message priority: %d", priority) ;
    Tcl_SetObjResult(interp, errObj) ;
    return TCL_ERROR ;
}
mqPtr->defpriority = priority ;
```

Tests

```
<<mq tests>>=
test mq-set-option-1.0 {
    Set / get priority option on open queue
} -setup {
    set queueName /mq-set-option-1.0
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    set mqchan [mq open $queueName w+]
    log::debug "mq channel = $mqchan"
```

```
    chan configure $mqchan -priority 20
    chan configure $mqchan -priority
} -result {20}
```

```
<<mq tests>>=
test mq-set-option-1.1 {
    Set priority option to negative number
} -setup {
    set queueName /mq-set-option-1.1
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    set mqchan [mq open $queueName w+]
    log::debug "mq channel = $mqchan"
    chan configure $mqchan -priority -20
} -result {posixipc mq: set option: invalid negative message priority: -20}\
  -returnCodes error
```

```
<<mq tests>>=
test mq-set-option-2.0 {
    Attempt to set create time option
} -setup {
    set queueName /mq-set-option-2.0
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    set mqchan [mq open $queueName w+]
    log::debug "mq channel = $mqchan"
    chan configure $mqchan -msgsize 128
} -result {posixipc mq: set option: -msgsize option may not be set by\
        configuration}\
  -returnCodes error
```

```
<<mq tests>>=
test mq-set-option-3.0 {
    Attempt to set bad option
} -setup {
    set queueName /mq-set-option-3.0
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    set mqchan [mq open $queueName w+]
    log::debug "mq channel = $mqchan"
    chan configure $mqchan -foo 128
} -result {bad option "-foo": should be one of -blocking, -buffering,\
    -buffersize, -encoding, -eofchar, -translation, -curmsgs, -lastpriority,\
    -maxmsg, -msgsize, or -priority}\
  -returnCodes error
```

Watch procedure

```
<<mq forward function declarations>>=
static void mqWatchProc(ClientData instanceData, int mask) ;
```

The code for the watch procedure was taken directly from the Tcl sources for the watch procedure for files. Since the message queue descriptor is actually a file descriptor, we can use treat the message queue just like a file for the purposes of watching.

```
<<mq static function definitions>>=
static void
mqWatchProc(
    ClientData instanceData,
    int mask)
{
    MqState *mqPtr = instanceData ;

    mask &= mqPtr->tclMask ;
    if (mask != 0) {
        Tcl_CreateFileHandler(mqdToFd(mqPtr->mqdes), mask,
                (Tcl_FileProc *)Tcl_NotifyChannel, mqPtr->channel) ;
    } else {
        Tcl_DeleteFileHandler(mqdToFd(mqPtr->mqdes)) ;
    }
}
```

Get handle procedure

```
<<mq forward function declarations>>=
static int mqGetHandleProc(ClientData instanceData, int direction,
        ClientData *handlePtr) ;
```

The message queue descriptor is the OS handle for the message queue, so we simply return it.

```
<<mq static function definitions>>=
static int
mqGetHandleProc(
    ClientData instanceData,
    int direction,
    ClientData *handlePtr)
{
    MqState *mqPtr = instanceData ;

    if ((direction & mqPtr->tclMask) == 0) {
        return TCL_ERROR ;
    }

    *handlePtr = (ClientData)(intptr_t)(mqPtr->mqdes) ; // ❶
    return TCL_OK ;
}
```

❶ This little contortion of casts suppresses the compiler warning about casting from integers to pointers of different sizes. The `ClientData` type is a pointer so we first cast the message queue descriptor to be something that will be pointer sized, but still integer typed and then, with the right sized object in hand, we can cast it to a pointer without generating a warning. Tcl sources have macros to do this, but they are only in the internal header file, which we are scrupulously avoiding in this code.

Processing Open Mode Options

Before describing how a message queue channel is opened, we must take an aside to deal with processing the mode options for opening a channel. This option processing also will be used for shared memory opening, so we are factoring it into functions that can be used when we take up other forms of IPC.

We pattern the option specifications after the Tcl `open` core command. This command takes two forms of open mode options. The first form is like `fopen(3)` modes and the second form is like `open(2)` flags.

Processing fopen style modes

First, we deal with mode arguments that are patterned after `fopen(3)`. We use a data driven approach. All possible values of the mode are put into a table that is used to map mode name to the flags and other information used in the opening of the message queue.

```
<<common forward function declarations>>=
static int processFopenModeOption(Tcl_Interp *interp, char const *const mode,
        int *flags, int *mask, bool *binary) ;
```

Processing the mode option in this case implies searching for the mode argument. We search the initialized array using standard library `bsearch(3)` function. Upon finding the mode string, then we return the open flags, a mask indicating the direction of data flow requested and whether the channel should be put into binary translation mode.

```
<<common static function definitions>>=
static int
processFopenModeOption(
    Tcl_Interp *interp,
    char const *const mode,
    int *flags,
    int *mask,
    bool *binary)
{
    <<processFopenModeOption: fopen style modes>>

    struct fopenModeDes const *modeSpec ;
    <<processFopenModeOption: search for mode match>>

    if (modeSpec == NULL) {
        Tcl_SetObjResult(interp, Tcl_ObjPrintf("unknown open mode, \"%s\": "
                "should be one of %s, %s %s, %s, %s, %s, %s, or %s",
                mode,
                fopenModes[0].modeName, fopenModes[1].modeName,
                fopenModes[3].modeName, fopenModes[4].modeName,
                fopenModes[5].modeName, fopenModes[6].modeName,
                fopenModes[8].modeName, fopenModes[9].modeName)) ;
        return TCL_ERROR ;
    }

    *flags = modeSpec->oFlags ;
    *mask = modeSpec->tclMask ;
    *binary = modeSpec->isBinary ;

    return TCL_OK ;
}
```

search for a mode name match

```
<<processFopenModeOption: search for mode match>>=
struct fopenModeDes key = {
    .modeName = mode,
} ;

modeSpec = bsearch(&key, fopenModes, COUNTOF(fopenModes),
        sizeof(struct fopenModeDes), fopenModeCompare) ;
```

We define a data structure to hold the mode name string and its associated data.

```
<<common data types>>=
struct fopenModeDes {
    char const *modeName ;          // ❶
    int oFlags ;                    // ❷
```

```
    int tclMask ;                   // ❸
    bool isBinary ;                 // ❹
} ;
```

❶ The string value to indicate the opening mode.

❷ The flags implied by the mode string. These are the `O_*` flags used for opening the message queue.

❸ A corresponding set of flags as used by Tcl. We will need to check whether the message is readable or writable.

❹ Indicates whether the message queue is opened in binary mode. We need to know this so the translation configuration on the channel can be properly set.

We use an initialized array variable to hold the set of possible open modes. Note there are no *append* modes since this is not a supported concept. Also note, the array must be ordered by ascending `modeName`.

```
<<processFopenModeOption: fopen style modes>>=
static struct fopenModeDes const fopenModes[] = { // ordered by ascending modeName!
    {
        .modeName = "r",
        .oFlags = O_RDONLY,
        .tclMask = TCL_READABLE,
        .isBinary = false,
    }, {
        .modeName = "r+",
        .oFlags = O_RDWR,
        .tclMask = TCL_READABLE | TCL_WRITABLE,
        .isBinary = false,
    }, {
        .modeName = "r+b",
        .oFlags = O_RDWR,
        .tclMask = TCL_READABLE | TCL_WRITABLE,
        .isBinary = true,
    }, {
        .modeName = "rb",
        .oFlags = O_RDONLY,
        .tclMask = TCL_READABLE,
        .isBinary = true,
    }, {
        .modeName = "rb+",
        .oFlags = O_RDWR,
        .tclMask = TCL_READABLE | TCL_WRITABLE,
        .isBinary = true,
    }, {
        .modeName = "w",
        .oFlags = O_WRONLY | O_CREAT,
        .tclMask = TCL_WRITABLE,
        .isBinary = false,
    }, {
        .modeName = "w+",
        .oFlags = O_RDWR | O_CREAT,
        .tclMask = TCL_READABLE | TCL_WRITABLE,
        .isBinary = false,
    }, {
        .modeName = "w+b",
        .oFlags = O_RDWR | O_CREAT,
        .tclMask = TCL_READABLE | TCL_WRITABLE,
        .isBinary = true,
    }, {
        .modeName = "wb",
        .oFlags = O_WRONLY | O_CREAT,
```

```
        .tclMask = TCL_WRITABLE,
        .isBinary = true,
    }, {
        .modeName = "wb+",
        .oFlags = O_RDWR | O_CREAT,
        .tclMask = TCL_READABLE | TCL_WRITABLE,
        .isBinary = true,
    },
} ;
```

As required when using `bsearch(3)`, we must supply a comparison function.

```
<<common forward function declarations>>=
static int fopenModeCompare(void const *m1, void const *m2) ;
```

Since we are searching based on string values, `strcmp(3)` provides the correct interface.

```
<<common static function definitions>>=
static int
fopenModeCompare(
    void const *m1,
    void const *m2)
{
    struct fopenModeDes const *o1 = m1 ;
    struct fopenModeDes const *o2 = m2 ;

    return strcmp(o1->modeName, o2->modeName) ;
}
```

Fopen style mode testing

The following are tests to verify the parsing of the fopen style modes.

```
<<mq tests>>=
test fopen-mode-1.0 {
    Open the message queue in "w" mode
} -setup {
    set queueName /fopen-mode-1.0
    set queueFile $mqFSMount$queueName
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    set mqchan [mq open $queueName w]
    log::debug "mq channel = $mqchan"

    set fattrs [file attributes $queueFile]
    log::debug "mq file attributes = $fattrs"
    set rightUser [expr {[dict get $fattrs -owner] eq $::tcl_platform(user)}]

    set exists [file exists $queueFile]

    return [expr {$rightUser && $exists}]
} -result {1}
```

```
<<mq tests>>=
test fopen-mode-1.1 {
    Open the message queue in "wb+" mode
} -setup {
    set queueName /fopen-mode-1.1
```

```
    set queueFile $mqFSMount$queueName
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    set mqchan [mq open $queueName wb+]
    log::debug "mq channel = $mqchan"

    set fattrs [file attributes $queueFile]
    log::debug "mq file attributes = $fattrs"
    set rightUser [expr {[dict get $fattrs -owner] eq $::tcl_platform(user)}]

    set exists [file exists $queueFile]

    set trans [chan configure $mqchan -translation]
    log::debug "translation = \"$trans\""
    set isBinary [expr {$trans eq "lf lf"}]

    return [expr {$rightUser && $exists && $isBinary}]
} -result {1}
```

```
<<mq tests>>=
test fopen-mode-2.0 {
    Open the message queue with bad mode
} -setup {
    set queueName /fopen-mode-2.0
} -cleanup {
} -body {
    mq open $queueName a
} -result {unknown open mode, "a": should be one of r, r+ rb, rb+, w, w+, wb,\
        or wb+}\
    -returnCodes error
```

Processing open style modes

The open style modes use a list of key strings to indicate the mode. We use the same strategy for processing the options as we did previously only now we will have to iterate across a list of key strings. A mapping structure is built to map the mode name string to the set of properties we need to open a message queue in the specified mode.

```
<<common forward function declarations>>=
static int processOpenModeOption(Tcl_Interp *interp, char const *const mode,
        int *flags, int *mask, bool *binary) ;
```

The mode argument is a list of flags names to use for opening. We must split up the list and then iterate across it to gather the set of specified flags. The implication of the list is that the various flags implied by the list elements are **OR**'ed together.

```
<<common static function definitions>>=
static int
processOpenModeOption(
    Tcl_Interp *interp,
    char const *const mode,
    int *flags,
    int *mask,
    bool *binary)
{
    int resultCode = TCL_OK ;

    <<processOpenModeOption: open style modes>>

    <<processOpenModeOption: split mode items list>>
```

```
    <<processOpenModeOption: iterate over mode items>>

    <<processOpenModeOption: check access direction>>

    *flags = oflags ;
    *mask = tclMask ;
    *binary = bintrans ;

release_list_items:
    ckfree(modeItems) ;
    return resultCode ;
}
```

Tcl provides a function to do the heavy lifting of splitting a list.

```
<<processOpenModeOption: split mode items list>>=
int modeCount ;
char const **modeItems ;

if (Tcl_SplitList(interp, mode, &modeCount, &modeItems) != TCL_OK) {
    return TCL_ERROR ;
}
```

Iterating over each item in the list, we search the flags table for a match. Finding a match, the associated data is **OR**'ed into the variables used to accumulate the bit encoded flags.

```
<<processOpenModeOption: iterate over mode items>>=
int oflags = 0 ;
int tclMask = 0 ;
bool bintrans = false ;
int accessFlagCount = 0 ;

for (int itemIndex = 0 ; itemIndex < modeCount ; itemIndex++) {
    struct openFlagDes key = {
        .modeName = modeItems[itemIndex],
    } ;

    struct openFlagDes const *flagSpec =
            bsearch(&key, openFlags, COUNTOF(openFlags),
            sizeof(struct openFlagDes), openFlagCompare) ;

    if (flagSpec == NULL) {
        Tcl_SetObjResult(interp,
                Tcl_ObjPrintf("invalid access mode \"%s\": must be "
                "%s, %s, %s, %s, %s, %s, or %s", modeItems[itemIndex],
                openFlags[0].modeName, openFlags[1].modeName,
                openFlags[2].modeName, openFlags[3].modeName,
                openFlags[4].modeName, openFlags[5].modeName,
                openFlags[6].modeName)) ;
        resultCode = TCL_ERROR ;
        goto release_list_items ;          // ❶
    } else {
        oflags |= flagSpec->oFlags ;
        tclMask |= flagSpec->tclMask ;
        bintrans = bintrans || flagSpec->isBinary ;
        accessFlagCount += flagSpec->isAccessMode ;
    }
}
```

❶ Spliting the list causes memory to be allocated which must be returned. Don't bleed memory on an error.

One of the restrictions when opening using a list of open mode flags is that some access direction, read, write or both, must be specified. This is not a problem in the `fopen(3)` style mode string, since an access direction is implied by the mode string value. However, when specifying individual flags, we need to perform the check. The rules are that exactly one of read only, write only, or read / write must be specified. For this reason, we have accumulated a count of the number flags that specify access modes as we iterated across the list of flag names. The check for specifying an access mode becomes a simple test against one.

```
<<processOpenModeOption: check access direction>>=
if (accessFlagCount != 1) {
    Tcl_SetObjResult(interp, Tcl_NewStringObj(
            "access mode must specify exactly one of RDONLY, WRONLY, or RDWR",
            -1)) ;
    resultCode = TCL_ERROR ;
    goto release_list_items ;
}
```

Note the mode names follow those used in Tcl and are restricted to modes that apply to IPC channels. Again, the initializers for the array must be ordered by ascending `modeName` element value.

```
<<common data types>>=
struct openFlagDes {
    char const *modeName ;          // ❶
    int oFlags ;                    // ❷
    int tclMask ;                   // ❸
    bool isBinary ;                 // ❹
    bool isAccessMode ;             // ❺
} ;
```

❶ The string value to indicate the opening mode.

❷ The flags implied by the mode string. These are the `O_*` flags used for opening a file.

❸ A corresponding set of flags as used by Tcl. We will need to check whether things are readable or writable.

❹ Indicates whether binary mode is requested on opening. We need to know this so the translation configuration on the channel can be properly set.

❺ Indicates whether this flag controls the access mode when opening the message queue. The reason we need this member is because O_RDONLY is actually defined to be zero. So we can't test the value of the `oFlags` member for non-zero to indicate that the RDONLY flag was specified. Bother!

```
<<processOpenModeOption: open style modes>>=
static struct openFlagDes const openFlags[] = { // order by ascending modeName!
    {
        .modeName = "BINARY",
        .oFlags = 0,            // BINARY does not affect the open flags.
        .tclMask = 0,
        .isBinary = true,
        .isAccessMode = false,
    }, {
        .modeName = "CREAT",
        .oFlags = O_CREAT,
        .tclMask = 0,
        .isBinary = false,
        .isAccessMode = false,
    }, {
        .modeName = "EXCL",
        .oFlags = O_EXCL,
        .tclMask = 0,
        .isBinary = false,
        .isAccessMode = false,
    }, {
```

```
        .modeName = "NONBLOCK",
        .oFlags = O_NONBLOCK,
        .tclMask = 0,
        .isBinary = false,
        .isAccessMode = false,
    }, {
        .modeName = "RDONLY",
        .oFlags = O_RDONLY,
        .tclMask = TCL_READABLE,
        .isBinary = false,
        .isAccessMode = true,
    }, {
        .modeName = "RDWR",
        .oFlags = O_RDWR,
        .tclMask = TCL_READABLE | TCL_WRITABLE,
        .isBinary = false,
        .isAccessMode = true,
    }, {
        .modeName = "WRONLY",
        .oFlags = O_WRONLY,
        .tclMask = TCL_WRITABLE,
        .isBinary = false,
        .isAccessMode = true,
    },
} ;
```

As required when using `bsearch(3)`, we must supply a comparison function.

```
<<mq forward function declarations>>=
static int openFlagCompare(void const *m1, void const *m2) ;
```

Again, since we are searching based on string values, `strcmp(3)` provides the correct interface.

```
<<mq static function definitions>>=
static int
openFlagCompare(
    void const *m1,
    void const *m2)
{
    struct openFlagDes const *f1 = m1 ;
    struct openFlagDes const *f2 = m2 ;

    return strcmp(f1->modeName, f2->modeName) ;
}
```

Open style mode testing

The following are tests to verify the parsing of the open style modes.

```
<<mq tests>>=
test open-mode-1.0 {
    Open the message queue for writing and create if necessary
} -setup {
    set queueName open-mode-1.0
    set queueFile [file join $mqFSMount $queueName]
} -cleanup {
    chan close $mqchan
    file delete $queueFile
} -body {
    set mqchan [mq open /$queueName {WRONLY CREAT}]
```

```
    log::debug "mq channel = $mqchan"

    set fattrs [file attributes $queueFile]
    log::debug "mq file attributes = $fattrs"
    set rightUser [expr {[dict get $fattrs -owner] eq $::tcl_platform(user)}]

    set exists [file exists $queueFile]

    return [expr {$rightUser && $exists}]
} -result {1}
```

```
<<mq tests>>=
test open-mode-2.0 {
    Attempt to open with a bad flag name.
} -setup {
    set queueName /open-mode-2.0
} -cleanup {
} -body {
    mq open $queueName FOO
} -result {invalid access mode "FOO": must be BINARY, CREAT, EXCL,\
        NONBLOCK, RDONLY, RDWR, or WRONLY}\
    -returnCodes error
```

```
<<mq tests>>=
test open-mode-3.0 {
    Attempt to open with no access mode flag
} -setup {
    set queueName /open-mode-3.0
} -cleanup {
} -body {
    mq open $queueName CREAT
} -result {access mode must specify exactly one of RDONLY, WRONLY, or RDWR}\
    -returnCodes error
```

```
<<mq tests>>=
test open-mode-3.1 {
    Attempt to open with multiple access mode flags
} -setup {
    set queueName /open-mode-3.1
} -cleanup {
} -body {
    mq open $queueName {RDONLY RDWR}
} -result {access mode must specify exactly one of RDONLY, WRONLY, or RDWR}\
    -returnCodes error
```

Opening a Message Queue Channel

We provide an external function that may be called by a “C” based extension to open an message queue channel. We don’t expect that to be a common use case, but it is convenient when necessary and we will use the same function to provide the Tcl script level command to open a message queue channel. This function is patterned off of the Tcl “C” library function, `Tcl_FSOpenFileChannel()`. So it accepts arguments that are the same as the `mq open` command, parses them and creates the message queue channel.

```
<<mq external functions definitions>>=
Tcl_Channel
PosixIPC_OpenMQChannel(
    Tcl_Interp *interp,
    Tcl_Obj *name,
    char const *mode,
```

```
    mode_t permissions,
    int maxmsg,
    int msgsize,
    unsigned priority)
{
    int flags = 0 ;
    int tclMask = 0 ;
    bool bintrans = false ;

    if (islower(*mode)) {
        if (processFopenModeOption(interp, mode, &flags, &tclMask, &bintrans) !=
                TCL_OK) {
            return NULL ;
        }
    } else {
        if (processOpenModeOption(interp, mode, &flags, &tclMask, &bintrans) !=
                TCL_OK) {
            return NULL ;
        }
    }

    <<OpenMQChannel: open message queue>>

    Tcl_Channel mqChan ;

    <<OpenMQChannel: create message queue channel>>

    <<OpenMQChannel: set translation option>>

    <<OpenMQChannel: set buffer size>>

    return mqChan ;

release_chan:
    Tcl_Close(interp, mqChan) ;
    return NULL ;
}
```

Once all the options are parsed and the flag values indicated by the options are known, we can actually open the message queue. The only rule here is that if the flags indicate that the message queue is to be created, then `mq_open(3)` requires two additional arguments to specify the file permissions and the properties of the message queue. Note that message queue properties of maximum number of messages in the queue and message size can only be set when the message queue is created and cannot be changed afterwards.

```
<<OpenMQChannel: open message queue>>=
char const *mqname = Tcl_GetString(name) ;
struct mq_attr attrs = {
    .mq_flags = 0,
    .mq_maxmsg = maxmsg,
    .mq_msgsize = msgsize,
    .mq_curmsgs = 0
} ;

mqd_t mqdes = mq_open(mqname, flags, permissions, &attrs) ;
if (mqdes == (mqd_t)(-1)) {
    mqFailure(interp, mqname) ;
    return NULL ;
}
```

Now that the message queue is open, we can create the Tcl channel associated with the message queue. After creating the channel, we store the data needed to manage the channel in the instance data associated to it.

```
<<OpenMQChannel: create message queue channel>>=
char chanName[TCL_INTEGER_SPACE + 16] ;
snprintf(chanName, sizeof(chanName), "posixmq%d", mqdToFd(mqdes)) ;
MqState *mqdata = ckalloc(sizeof(MqState)) ;
mqChan = Tcl_CreateChannel(&MqChannelType, chanName, mqdata, tclMask) ;

mqdata->channel = mqChan ;
mqdata->mqdes = mqdes ;
mqdata->tclMask = tclMask ;
mqdata->lastpriority = 0 ;
mqdata->defpriority = priority ;
```

If the channel was opened in binary mode, we need to instruct the Tcl channel system to set the translation properly.

```
<<OpenMQChannel: set translation option>>=
if (bintrans) {
    if (Tcl_SetChannelOption(interp, mqChan, "-translation", "binary") !=
            TCL_OK) {
        goto release_chan ;
    }
}
```

Finally, we want to make sure that the buffering in the Tcl channel matches the size of a message. This is a convenience for users having to remember to set the buffer size properly. Forgetting to do so often results in an error when using message queue. The default Tcl channel buffer size is 4 KiBytes. There is little reason to have the buffer size be any different than the message size.

```
<<OpenMQChannel: set buffer size>>=

int syserr = mq_getattr(mqdata->mqdes, &attrs) ;
if (syserr < 0) {
    mqFailure(interp, "mq_getattr failed during open") ;
    goto release_chan ;
}

Tcl_SetChannelBufferSize(mqChan, attrs.mq_msgsize) ;
```

Message Queue Commands

In this section, we show the functions that implement the script level commands for message queues.

mq open command

`posixipc mq open` ?options? mqname
`posixipc mq open` ?options? mqname ?access?
`posixipc mq open` ?options? mqname ?access? ?permissions?

options
The following options are write only and must be supplied to the `open` command, but may be read as channel configuration options:

-maxmsg *count*
The maximum number of messages that may be queued. If not specified, a system default value is used.

-msgsize *len*
The maximum length in bytes any message. If not specified, a system default value is used.

The following options are read / write and may be supplied to the `open` command or may be accessed as configuration options. -priority *priority_number*;; The default message priority as a non-negative number. If not specified, the default priority is 0. Messages are placed on the message queue in decreasing order of priority.

The following options are read only and can only be read as configuration options:

-curmsgs
The number of messages currently in the queue.

-lastpriority
The message priority of the last received message.

mqname
The name of message queue. Message queue names must start with a slash (/) character and may **not** contain any other slash characters.

access
The *access* argument, if present, indicates the way in which the message queue is to be accessed. In the first form, *access* may have any of the following forms:

r
Open the message queue for reading only. The message queue must already exist. This is the default if *access* is not specified.

r+
Open the message queue for reading and writing. The message queue must already exist.

w
Open the message queue for writing only. Create the message queue if it does not already exist.

w+
Open the message queue for reading and writing. Create the message queue if it does not already exist.

All legal *access* values above may have the character **b** added as the second or third character in the value to indicate that the opened channel should be configured as if with the **fconfigure -translation binary** option, making the channel suitable for reading or writing of binary data.

In the second form, *access* consists of a list of any of the following flags, all of which have the standard POSIX meanings. Exactly one flags must be either **RDONLY**, **WRONLY**, or **RDWR**.

RDONLY
Open the message queue for reading only.

WRONLY
Open the message queue for writing only.

RDWR
Open the message queue for both reading and writing.

BINARY
Configure the opened channel with the **-translation binary** option.

CREAT
Create the message queue if it does not already exist.

EXCL
If **CREAT** is also specified, an error is returned if the message queue already exists.

NONBLOCK
Prevent the process from blocking in subsequent I/O operations.

permissions
If the message queue is created as part of opening it, *permissions* is used to set the permissions for the new message queue in conjunction withthe process's file mode creation mask. *Permissions* defaults to 0666.

```
<<mq forward function declarations>>=
static int mqOpenCmd(ClientData clientData, Tcl_Interp *interp, int objc,
        Tcl_Obj *const objv[]) ;
```

Like much of the "C" code for Tcl commands, we spend a good deal of effort just to parse and validate the input arguments.

```
<<mq static function definitions>>=
static int
mqOpenCmd(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const objv[])
{
    static char const usage[] = "?options? queuename ?access? ?permissions?" ;
    if (objc < 2) {
```

```
        Tcl_WrongNumArgs(interp, 1, objv, usage) ;
        return TCL_ERROR ;
    }

    <<mqOpenCmd: establish default option values>>

    <<mqOpenCmd: parse options arguments>>

    <<mqOpenCmd: gather remaining arguments>>

    <<mqOpenCmd: create channel>>

    return TCL_OK ;
}
```

We assign the argument defaults to variables. These variables are then overwritten as matching arguments are found. For the maximum number of messages (`maxmsg`) and message size (`msgsize`) arguments, the values are the default for many installations.

```
<<mqOpenCmd: establish default option values>>=
int maxmsg = 10 ;
int msgsize = 8192 ;
int priority = 0 ;
char const *mode = "r" ;
mode_t permissions = 0666 ;
```

Because some properties of a message queue can only be specified when the queue is created, it is necessary for the `mq open` command to accept these arguments. The arguments are just the ones we have seen before when dealing with the configuration procedures.

We will deal with these open options in the same way as for the configurtion procedures, namely, look them up in our table and then convert the option value into an internal representation.

There are two minor issues to be handled:

1. *Options* must be specified as option name / option value pairs.
2. The *options* are optional. They may not be specified and so we must to be able to skip them to find the required arguments.

To handle these issues, this design uses an `objindex` as a cursor to track our current location in the command argument handling. The index is incremented as we iterate across the arguments to the command and can then be used to locate option values.

```
<<mqOpenCmd: parse options arguments>>=
int objindex = 1 ;                                    // ❶
for ( ; objindex < objc ; objindex++) {
    <<mqOpenCmd: look up the option name>>

    <<mqOpenCmd: index the option value>>

    <<mqOpenCmd: convert the option value>>
}
```

❶ Skip the command name and refer to the first real argument. Also, we need `objindex` later, so it must be declared outside of the `for` statement.

To look up the option name, we use `lookUpMqOpt()` that was described previously.

```
<<mqOpenCmd: look up the option name>>=
char const *optionName = Tcl_GetString(objv[objindex]) ;
if (strcmp(optionName, "--") == 0) {                          // ❶
    objindex++ ;                                              // ❷
    break ;
}
struct MqConfigOpt const *optProp = lookUpMqOpt(optionName) ;
if (optProp == NULL) {                                        // ❸
    if (*optionName == '-') {
        return Tcl_BadChannelOption(interp, optionName, mqOptionsList) ;
    } else {
        break ;
    }
}
if (!optProp->setOnOpen) {                                    // ❹
    Tcl_SetObjResult(interp,
            Tcl_ObjPrintf("option, %s, cannot be set on open: "
            "should be one of: %s, %s, or %s",
            optProp->optName, mqOptions[2].optName, mqOptions[3].optName,
            mqOptions[4].optName)) ;
    return TCL_ERROR ;
}
```

❶ Using the usual convention, "--" stops option process so that the next argument could start with a dash character.

❷ Skip the option termination argument and stop option processing.

❸ The first command argument that doesn't look like a valid option halts option processing.

❹ Options that can't be set on an open call are in error.

Having found the option to be allowed, we increment `objindex` to index its value.

```
<<mqOpenCmd: index the option value>>=
if (++objindex >= objc) {                                     // ❶
    Tcl_WrongNumArgs(interp, 1, objv, usage) ;
    return TCL_ERROR ;
}
```

❶ We must not increment off the end of the argument array.

There is specific code required to convert the option value into the proper internal form. Again we will fall back on our previous stategy by computing the array index of the option in the option table and using that as an integer encoding in a `switch` statement.

```
<<mqOpenCmd: convert the option value>>=
MqOptionType whichOpt = (MqOptionType)(optProp - mqOptions) ;
switch (whichOpt) {
    case Mq_maxmsg_opt:
        if (Tcl_GetIntFromObj(interp, objv[objindex], &maxmsg) !=
                TCL_OK) {
            return TCL_ERROR ;
        }
        break ;
    case Mq_msgsize_opt:
        if (Tcl_GetIntFromObj(interp, objv[objindex], &msgsize) !=
                TCL_OK) {
            return TCL_ERROR ;
        }
```

```
        break ;
    case Mq_priority_opt:
        if (Tcl_GetIntFromObj(interp, objv[objindex], &priority) !=
                TCL_OK) {
            return TCL_ERROR ;
        }
        if (priority < 0) {                                    // ❶
            Tcl_SetObjResult(interp,
                    Tcl_ObjPrintf("negative values for option, \"%s\", "
                    "are not allowed: got %d",
                    optionName, priority)) ;
            return TCL_ERROR ;
        }
        break ;
    default:
        Tcl_Panic("posixipc mq: open option failure: "
                "found option named, "
                "\"%s\", but no code to process the option",
                optionName) ;
        break ;
}
```

❶ Negative priorities are not allowed.

Having dealt with options that might affect the message creation, we need to gather up the message queue name and any access mode or creation permission arguments that were specified. We start by computing the number of remaining arguments. From there, we can decide if there were enough arguments and which of the optional arguments that come after the message queue name are present.

```
<<mqOpenCmd: gather remaining arguments>>=
int remain = objc - objindex ;
if (remain < 1 || remain > 3) {
    Tcl_WrongNumArgs(interp, 1, objv, usage) ;
    return TCL_ERROR ;
}

if (remain > 1) {
    mode = Tcl_GetString(objv[objindex + 1]) ;
}

if (remain == 3) {
    int permArgValue ;
    if (Tcl_GetIntFromObj(interp, objv[objindex + 2], &permArgValue)
            != TCL_OK) {
        return TCL_ERROR ;
    }
    permissions = permArgValue ;
}
```

Finally, we can invoke `PosixIPC_OpenMQChannel()` to create the message queue channel.

```
<<mqOpenCmd: create channel>>=
Tcl_Channel chan = PosixIPC_OpenMQChannel(interp, objv[objindex], mode,
        permissions, maxmsg, msgsize, priority) ;
if (chan == NULL) {
    return TCL_ERROR ;
}
Tcl_RegisterChannel(interp, chan) ;                            // ❶
Tcl_SetObjResult(interp, Tcl_NewStringObj(Tcl_GetChannelName(chan), -1)) ;
```

❶ Newly created channels must be registered to be used in the interpreter.

Tests

```
<<mq tests>>=
test mq-open-1.0 {
    Open a message queue
} -setup {
    set queueName /mq-open-1.0
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    set mqchan [mq open $queueName w+]
    log::debug "mq channel = $mqchan"
    return $mqchan
} -result {posixmq[0-9]*} -match glob
```

```
<<mq tests>>=
test mq-open-2.0 {
    Open a message queue with create time options
} -setup {
    set queueName /mq-open-2.0
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    set mqchan [mq open -msgsize 256 -priority 10 -- $queueName w+]
    log::debug "mq channel = $mqchan"
    chan configure $mqchan -msgsize
} -result {256}
```

```
<<mq tests>>=
test mq-open-2.1 {
    Open a message queue with incorrect create time options
} -setup {
    set queueName /mq-open-2.1
} -cleanup {
} -body {
    set mqchan [mq open -curmsgs 256 $queueName w+]
} -result {option, -curmsgs, cannot be set on open: should be one of:\
        -maxmsg, -msgsize, or -priority}\
   -returnCodes error
```

```
<<mq tests>>=
test mq-open-2.2 {
    Open a message queue with message size too large
} -setup {
    set queueName /mq-open-2.2
} -cleanup {
} -constraints linuxOnly -body {
    set mqchan [mq open -msgsize 16000 $queueName w+]
} -result {posixipc mq: /mq-open-2.2: invalid argument}\
   -returnCodes error
```

```
<<mq tests>>=
test mq-open-2.3 {
    Open a message queue lacking option argument
} -setup {
    set queueName /mq-open-2.3
```

```
} -cleanup {
} -body {
    set mqchan [mq open -msgsize 200 -priority]
} -result {wrong # args: should be "mq open ?options? queuename ?access?\
        ?permissions?"} -returnCodes error
```

```
<<mq tests>>=
test mq-open-2.4 {
    Open a message queue with bad message size
} -setup {
    set queueName /mq-open-2.4
} -cleanup {
} -body {
    set mqchan [mq open -msgsize foo $queueName w+]
} -result {expected integer but got "foo"} -returnCodes error
```

```
<<mq tests>>=
test mq-open-2.5 {
    Open a message queue with bad max messages
} -setup {
    set queueName /mq-open-2.5
} -cleanup {
} -body {
    set mqchan [mq open -maxmsg foo $queueName w+]
} -result {expected integer but got "foo"} -returnCodes error
```

```
<<mq tests>>=
test mq-open-2.6 {
    Open a message queue with bad priority
} -setup {
    set queueName /mq-open-2.6
} -cleanup {
} -body {
    set mqchan [mq open -priority foo $queueName w+]
} -result {expected integer but got "foo"} -returnCodes error
```

```
<<mq tests>>=
test mq-open-2.7 {
    Open a message queue with wrong number of arguments
} -setup {
} -cleanup {
} -body {
    set mqchan [mq open]
} -result {wrong # args: should be "mq open ?options? queuename ?access?\
        ?permissions?"} -returnCodes error
```

```
<<mq tests>>=
test mq-open-2.8 {
    Open a message queue with option name
} -setup {
    set queueName /mq-open-2.8
} -cleanup {
} -body {
    set mqchan [mq open -foobar 27 $queueName w+]
} -result {bad option "-foobar": should be one of -blocking, -buffering,\
        -buffersize, -encoding, -eofchar, -translation, -curmsgs,\
        -lastpriority, -maxmsg, -msgsize, or -priority} -returnCodes error
```

```
<<mq tests>>=
```

```
test mq-open-2.9 {
    Open a message queue with negative priority
} -setup {
    set queueName /mq-open-2.9
} -cleanup {
} -body {
    set mqchan [mq open -priority -30 $queueName w+]
} -result {negative values for option, "-priority", are not allowed: got -30}\
        -returnCodes error
```

```
<<mq tests>>=
test mq-open-2.10 {
    Open a message queue with wrong number of arguments after options
} -setup {
} -cleanup {
} -body {
    set mqchan [mq open -priority 20]
} -result {wrong # args: should be "mq open ?options? queuename ?access?\
        ?permissions?"} -returnCodes error
```

```
<<mq tests>>=
test mq-open-2.11 {
    Open a message queue with negative priority
} -setup {
    set queueName /mq-open-2.11
} -cleanup {
} -body {
    set mqchan [mq open $queueName w+ foo]
} -result {expected integer but got "foo"} -returnCodes error
```

```
<<mq tests>>=
test mq-open-2.12 {
    Open a message queue with non-default permissions
} -setup {
    set queueName /mq-open-2.12
    set queueFile $mqFSMount$queueName
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    set mqchan [mq open $queueName w+ 0640]

    set fattrs [file attributes $queueFile]
    log::debug "mq file attributes = $fattrs"
    dict get $fattrs -permissions
} -result {00640}
```

```
<<mq tests>>=
test mq-open-3.0 {
    Write to a message queue
} -setup {
    set queueName /mq-open-3.0
    set mqchan [mq open -maxmsg 1 -msgsize 128 $queueName w+]
    chan configure $mqchan\
        -buffering line
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    log::debug "mq channel = $mqchan"
```

```
    log::debug "max messages = [chan configure $mqchan -maxmsg]"
    log::debug "message size = [chan configure $mqchan -msgsize]"

    chan puts $mqchan "Write a message"
    chan gets $mqchan
} -result {Write a message}
```

mq send command

`posixipc mq send` chan msg ?priority?

chan
A message queue channel token as returned from `mq open`. The channel must have been opened for writing.

msg
A message to be placed on the message queue.

priority
An optional unsigned number specifying the priority of the message. If *priority* is not specfied, then the current value of the `-priority` configuration option is used as the message priority.

The `mq send` command operates outside of the buffering of the Tcl virtual channel system and directly sends *msg* to the message queue associated with *chan*. If specified, the message is sent with *priority* priority. This command is provided for applications where message priority changes frequently, such as on a message by message basis.

```
<<mq forward function declarations>>=
static int mqSendCmd(ClientData clientData, Tcl_Interp *interp, int objc,
        Tcl_Obj *const objv[]) ;
```

```
<<mq static function definitions>>=
static int
mqSendCmd(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const objv[])
{
    static char const usage[] = "mqchan msg ?priority?" ;
    if (objc < 3 || objc > 4) {
        Tcl_WrongNumArgs(interp, 1, objv, usage) ;
        return TCL_ERROR ;
    }

    char const *chanName = Tcl_GetString(objv[1]) ;
    int mode ;
    Tcl_Channel channel = Tcl_GetChannel(interp, chanName, &mode) ;
    if (channel == NULL) {
        return TCL_ERROR ;
    }
    MqState *mqdata = Tcl_GetChannelInstanceData(channel) ;

    int msglen ;
    unsigned char *msg = Tcl_GetByteArrayFromObj(objv[2], &msglen) ;

    int priority = mqdata->defpriority ;
    if (objc == 4) {
        if (Tcl_GetIntFromObj(interp, objv[3], &priority) != TCL_OK) {
            return TCL_ERROR ;
```

```
        }
        if (priority < 0) {
            Tcl_SetObjResult(interp,
                    Tcl_ObjPrintf("negative values for priority "
                    "are not allowed: got %d", priority)) ;
            return TCL_ERROR ;
        }
    }

    int err = mq_send(mqdata->mqdes, (char const *)msg, msglen, priority) ;
    if (err < 0) {
        return mqFailure(interp, "mq_send failed") ;
    }

    Tcl_SetObjResult(interp, Tcl_NewIntObj(msglen)) ;
    return TCL_OK ;
}
```

Tests

```
<<mq tests>>=
test mq-send-1.0 {
    Write to a message queue using send command
} -setup {
    set queueName /mq-send-1.0
    set mqchan [mq open $queueName w+]
    chan configure $mqchan -buffering line
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    log::debug "mq channel = $mqchan"

    mq send $mqchan "Write a message\n"
    chan gets $mqchan
} -result {Write a message}
```

```
<<mq tests>>=
test mq-send-2.0 {
    send message -- wrong args
} -setup {
} -cleanup {
} -body {
    mq send
} -result {wrong # args: should be "mq send mqchan msg ?priority?"}\
    -returnCodes error
```

```
<<mq tests>>=
test mq-send-2.1 {
    send message -- bad priority
} -setup {
    set queueName /mq-send-2.1
    set mqchan [mq open $queueName w+]
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    mq send $mqchan "Hello World" foobar
} -result {expected integer but got "foobar"} -returnCodes error
```

```
<<mq tests>>=
test mq-send-2.2 {
    send message -- negative priority
} -setup {
    set queueName /mq-send-2.2
    set mqchan [mq open $queueName w+]
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    mq send $mqchan "Hello World" -6
} -result {negative values for priority are not allowed: got -6}\
    -returnCodes error
```

```
<<mq tests>>=
test mq-send-2.3 {
    send message -- bad channel
} -setup {
} -cleanup {
} -body {
    mq send posixmq27 "Hello World"
} -result {can not find channel named "posixmq27"} -returnCodes error
```

```
<<mq tests>>=
test mq-send-3.0 {
    Send a message -- non-blocking mode and queue full
} -setup {
    set queueName /mq-send-3.0
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    set mqchan [mq open -maxmsg 1 $queueName {CREAT EXCL RDWR NONBLOCK}]
    mq send $mqchan "Hello World"
    mq send $mqchan "Hello World"
} -result {posixipc mq: mq_send failed: resource temporarily unavailable}\
    -returnCodes error
```

mq receive command

`posixipc mq receive` chan ?varname?

chan
A message queue channel token as returned from `mq open`. The channel must have been opened for reading.

varname
An optional variable name where the value of the priority of the received message is set. The priority of the last received message is also available as the `-lastpriority` configuration option.

The `mq receive` command operates outside of the buffering of the Tcl virtual channel system and returns the value of the next message on the message queue referenced by *chan*. If the variable, *varname*, is specified, the priority of the message is set as the variable's value. This command is provided for those applications where message priority changes frequently, such as on a message by message basis. The message priority of the last received message is also available as the `-lastpriority` configuration option.

```
<<mq forward function declarations>>=
static int mqReceiveCmd(ClientData clientData, Tcl_Interp *interp, int objc,
```

```
        Tcl_Obj *const objv[]) ;
```

```
<<mq static function definitions>>=
static int
mqReceiveCmd(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const objv[])
{
    static char const usage[] = "mqchan ?priorvar?" ;
    if (objc < 2 || objc > 3) {
        Tcl_WrongNumArgs(interp, 1, objv, usage) ;
        return TCL_ERROR ;
    }

    char const *chanName = Tcl_GetString(objv[1]) ;
    int mode ;
    Tcl_Channel channel = Tcl_GetChannel(interp, chanName, &mode) ;
    if (channel == NULL) {
        return TCL_ERROR ;
    }

    MqState *mqdata = Tcl_GetChannelInstanceData(channel) ;
    struct mq_attr attrs ;
    int syserr = mq_getattr(mqdata->mqdes, &attrs) ;
    if (syserr < 0) {
        return mqFailure(interp, "mq_getattr failed during receive") ;
    }

    Tcl_Obj *resultObj = Tcl_NewByteArrayObj(NULL, attrs.mq_msgsize) ;
    unsigned char *value = Tcl_GetByteArrayFromObj(resultObj, NULL) ;

    syserr = mq_receive(mqdata->mqdes, (char *)value, attrs.mq_msgsize,
            &mqdata->lastpriority) ;
    if (syserr < 0) {
        mqFailure(interp, "mq_receive failed") ;
        goto release_result ;
    }

    if (objc == 3) {
        Tcl_Obj *priorObj = Tcl_NewIntObj(mqdata->lastpriority) ;
        Tcl_IncrRefCount(priorObj) ;
        Tcl_Obj *newValue = Tcl_ObjSetVar2(interp, objv[2], NULL, priorObj,
                TCL_LEAVE_ERR_MSG) ;
        Tcl_DecrRefCount(priorObj) ;

        if (newValue == NULL) {
            goto release_result ;
        }
    }

    Tcl_SetObjResult(interp, resultObj) ;
    return TCL_OK ;

release_result:
    Tcl_DecrRefCount(resultObj) ;

    return TCL_ERROR ;
}
```

Tests

```
<<mq tests>>=
test mq-receive-1.0 {
    Receive a message using the receive command
} -setup {
    set queueName /mq-receive-1.0
    set mqchan [mq open $queueName w+]
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    log::debug "mq channel = $mqchan"

    set msgvalue "Receive a message"
    append msgvalue "\0"
    mq send $mqchan $msgvalue 20
    set msg [mq receive $mqchan msgpriority]
    set endmsg [expr {[string first \0 $msg] + 1}]
    binary scan $msg a$endmsg msgstr

    log::debug "message = \"$msgstr\""
    log::debug "priority = $msgpriority"
    log::debug "last priority = [chan configure $mqchan -lastpriority]"

    expr {$msgstr eq $msgvalue && $msgpriority == 20 &&\
            [chan configure $mqchan -lastpriority] == 20}
} -result {1}
```

```
<<mq tests>>=
test mq-receive-2.0 {
    Receive a message -- wrong arguments
} -setup {
} -cleanup {
} -body {
    mq receive
} -result {wrong # args: should be "mq receive mqchan ?priorvar?"}\
    -returnCodes error
```

```
<<mq tests>>=
test mq-receive-2.1 {
    Receive a message -- non-blocking mode and no messages
} -setup {
    set queueName /mq-receive-2.1
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    set mqchan [mq open $queueName {CREAT EXCL RDWR NONBLOCK}]
    mq receive $mqchan
} -result {posixipc mq: mq_receive failed: resource temporarily unavailable}\
    -returnCodes error
```

```
<<mq tests>>=
test mq-receive-2.2 {
    Receive a message -- bad channel
} -setup {
} -cleanup {
} -body {
    mq receive posixmq88
} -result {can not find channel named "posixmq88"} -returnCodes error
```

```
<<mq tests>>=
test mq-receive-2.3 {
    Receive a message -- bad priority variable
} -setup {
    set queueName /mq-receive-2.3
} -cleanup {
    chan close $mqchan
    mq unlink $queueName
} -body {
    set mqchan [mq open $queueName {CREAT EXCL RDWR NONBLOCK}]
    mq send $mqchan "Hello World" 20
    mq receive $mqchan ::foobar::priority
} -result {can't set "::foobar::priority": parent namespace doesn't exist}\
    -returnCodes error
```

mq unlink command

```
<<mq forward function declarations>>=
static int mqUnlinkCmd(ClientData clientData, Tcl_Interp *interp, int objc,
        Tcl_Obj *const objv[]) ;
```

```
posixipc mq unlink mqname
```

mqname
: The name of message queue. Message queue names must start with a slash (/) character and may **not** contain any other slash characters.

The `unlink` command removes the message queue named, `mqname`, from the system. The queue is actually destroyed when all processes that have open descriptors for the queue close those descriptors referring to the queue.

```
<<mq static function definitions>>=
static int
mqUnlinkCmd(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const objv[])
{
    static char const usage[] = "queuename" ;
    if (objc != 2) {
        Tcl_WrongNumArgs(interp, 1, objv, usage) ;
        return TCL_ERROR ;
    }

    char const *mqname = Tcl_GetString(objv[1]) ;
    int syserr = mq_unlink(mqname) ;
    if (syserr == -1) {
        return mqFailure(interp, mqname) ;
    }

    return TCL_OK ;
}
```

Tests

```
<<mq tests>>=
test mq-unlink-1.0 {
```

```
    Unlink a message queue
} -setup {
    set queueName /mq-unlink-1.0
} -cleanup {
    chan close $mqchan
} -body {
    set mqchan [mq open $queueName w+]
    log::debug "mq channel = $mqchan"
    mq unlink $queueName
    file exists [file join $mqFSMount $queueName]
} -result {0}
```

```
<<mq tests>>=
test mq-unlink-2.0 {
    Unlink a message queue -- bad argument
} -setup {
} -cleanup {
} -body {
    set mqchan [mq unlink]
} -result {wrong # args: should be "mq unlink queuename"} -returnCodes error
```

```
<<mq tests>>=
test mq-unlink-2.0 {
    Unlink a message queue -- unknown queue
} -setup {
} -cleanup {
} -body {
    set mqchan [mq unlink /foobar]
} -result {posixipc mq: /foobar: no such file or directory} -returnCodes error
```

Shared Memory

POSIX shared memory mechanisms are implemented in Linux using a virtual file system. Opening a shared memory object creates a file at a specfic location in the file system. The object exists until either the system is rebooted or the shared memory is explicitly unlinked. To use shared memory, it is mapped into the address space of the process. At that point, it just looks like any other data memory.

The Tcl channel metaphor extends well to shared memory. However, since the memory is shared between multiple processes, some form of exclusive access must be implemented to insure the sharing processes do not interfer with each other. We use a POSIX semaphore for that purpose. The **wait** and **post** operations on the semaphore are kept as part of the channel operations and are not exposed to the script level. The semaphore is only locked for a long as necessary to copy memory to and from the shared area.

In what follows, we present the Tcl channel driver functions first, followed by the script level Tcl commands.

Shared Memory Channel Driver

In this section we present the channel driver for the POSIX shared memory type channel.

Channel Driver Components

In this section, we describe the functions supplied for the Tcl channel driver that is used for POSIX shared memory. This is similar to what was supplied for the message queue driver.

The following is an initialized variable of the proper structure that is given to `Tcl_CreateChannel()` to define the shared memory operations.

```
<<shm static data>>=
static Tcl_ChannelType ShmChannelType = {
    .typeName = "shm",          /* The name of the channel type in Tcl
                                 * commands. This storage is owned by channel
                                 * type. */
    .version = TCL_CHANNEL_VERSION_5,
                                /* Version of the channel type. */
    .closeProc = shmCloseProc,
                                /* Function to call to close the channel, or
                                 * TCL_CLOSE2PROC if the close2Proc should be
                                 * used instead. */
    .inputProc = shmInputProc,
                                /* Function to call for input on channel. */
    .outputProc = shmOutputProc,
                                /* Function to call for output on channel. */
    .seekProc = shmSeekProc,
                                /* Function to call to seek on the channel.
                                 * May be NULL. */
    .setOptionProc = NULL,
                                /* Set an option on a channel. */
    .getOptionProc = NULL,
                                /* Get an option from a channel. */
    .watchProc = shmWatchProc,
                                /* Set up the notifier to watch for events on
                                 * this channel. */
    .getHandleProc = shmGetHandleProc,
                                /* Get an OS handle from the channel or NULL
                                 * if not supported. */
    .close2Proc = NULL,
                                /* Function to call to close the channel if
                                 * the device supports closing the read &
                                 * write sides independently. */
    .blockModeProc = shmBlockModeProc,
                                /* Set blocking mode for the raw channel. May
                                 * be NULL. */
    /*
     * Only valid in TCL_CHANNEL_VERSION_2 channels or later
     */
    .flushProc = NULL,
                                /* Function to call to flush a channel. May be
                                 * NULL. */
    .handlerProc = NULL,
                                /* Function to call to handle a channel event.
                                 * This will be passed up the stacked channel
                                 * chain. */
    /*
     * Only valid in TCL_CHANNEL_VERSION_3 channels or later
     */
    .wideSeekProc = NULL,
                                /* Function to call to seek on the channel
                                 * which can handle 64-bit offsets. May be
                                 * NULL, and must be NULL if seekProc is
                                 * NULL. */
    /*
     * Only valid in TCL_CHANNEL_VERSION_4 channels or later
     * TIP #218, Channel Thread Actions
     */
    .threadActionProc = NULL,
                                /* Function to call to notify the driver of
                                 * thread specific activity for a channel. May
                                 * be NULL. */
```

```
    /*
     * Only valid in TCL_CHANNEL_VERSION_5 channels or later
     * TIP #208, File Truncation
     */
    .truncateProc = NULL
                                /* Function to call to truncate the underlying
                                 * file to a particular length. May be NULL if
                                 * the channel does not support truncation. */
} ;
```

Notice that not all functions are provided (*i.e.* some members of the structure are set to `NULL`). Those members are:

setOptionProc
: There are no channel type specific options that can be written.

getOptionProc
: There are no channel type specific options that can be read.

close2Proc
: For shared memory, the idea of closing read or write separately does not apply.

flushProc
: This is reserved.

handlerProc
: Used only for stacked drivers.

wideSeekProc
: We assume shared memory areas are less than 2 GiBytes so wide seeking is not necessary.

threadActionProc
: There is no thread specific driver state.

truncateProc
: Once created, the size of a shared memory object underlying the channel is not changed.

When a new shared memory Tcl channel is created, some instance data for the channel may also be supplied. That instance data is then provided as an argument to each of the functions for the driver. This makes things much more convenient for the driver function code to have access to the state of the channel. The following data structure defines the information that constitutes the state of an individual shared memory channel.

```
<<shm data types>>=
typedef struct {
    Tcl_Channel channel ;       // ❶
    int shmfd ;                 // ❷
    sem_t *semaphore ;          // ❸
    int tclMask ;               // ❹
    int blockingMode ;          // ❺
    uint8_t *address ;          // ❻
    size_t length ;             // ❼
    size_t position ;           // ❽
} ShmState ;
```

❶ The Tcl channel handle as returned from `Tcl_CreateChannel()`.

❷ The shared memory descriptor. In reality on Linux this is an ordinary file descriptor.

❸ The semaphore used to control exclusive access to the shared memory.

❹ A mask of allowed operations on the channel. This is combination of TCL_READABLE or TCL_WRITABLE.

5. The current state of blocking mode for the channel.
6. The address in memory where the shared memory area is located.
7. The length in bytes of the shared memory area.
8. The current position in the shared memory area where the next read or write will occur.

The following sections describe each of the functions supplied as part of the shared memory channel driver.

Close procedure

```
<<shm forward function declarations>>=
static int shmCloseProc(ClientData instanceData, Tcl_Interp *interp) ;
```

To close a share memory object, we must unmap it, close the underlying file, and close the exclusion semaphore. Should an error occur in this process, we want to continue to finish all the operations. We report the last error that occurred.

```
<<shm static function definitions>>=
static int
shmCloseProc(
    ClientData instanceData,
    Tcl_Interp *interp)
{
    ShmState *shmData = instanceData ;

    Tcl_DeleteFileHandler(shmData->shmfd) ;

    int resultCode = 0 ;

    int syserr = munmap(shmData->address, shmData->length) ;
    if (syserr < 0) {
        resultCode = errno ;
    }

    syserr = close(shmData->shmfd) ;
    if (syserr < 0) {
        resultCode = errno ;
    }

    syserr = sem_close(shmData->semaphore) ;
    if (syserr < 0) {
        resultCode = errno ;
    }

    ckfree(shmData) ;
    return resultCode ;
}
```

Input procedure

```
<<shm forward function declarations>>=
static int shmInputProc(ClientData instanceData, char *buf, int bufSize,
        int *errorCodePtr) ;
```

Input from the share memory is just copying from memory. There is necessary book keeping to make sure we don't run off the end of the memory area. Also, we must acquire the semaphore used to insure exclusive access to the shared memory.

```
<<shm static function definitions>>=
static int
shmInputProc(
    ClientData instanceData,
    char *buf,
    int bufSize,
    int *errorCodePtr)
{
    ShmState *shmData = instanceData ;

    <<shm: lock semaphore>>

    size_t remaining = shmData->length - shmData->position ;
    int toRead = bufSize < remaining ? bufSize : remaining ;

    memcpy(buf, shmData->address + shmData->position, toRead) ;
    shmData->position += toRead ;

    <<shm: unlock semaphore>>

    return toRead ;
}
```

We interpret non-blocking mode as not wanting to wait on the semaphore for access to the shared memory. We don't anticipate this to be a common situation. The semaphore is *hidden* in the channel driver and is only locked during the memory copy operations. So there seems little reason to use non-blocking mode for the shared memory.

lock semaphore

```
<<shm: lock semaphore>>=
int syserr ;
if (shmData->blockingMode == TCL_MODE_NONBLOCKING) {
    syserr = sem_trywait(shmData->semaphore) ;
    if (syserr == -1) {
        *errorCodePtr = errno ;
        return -1 ;
    }
} else {
    syserr = sem_wait(shmData->semaphore) ;
    if (syserr == -1) {
        *errorCodePtr = errno ;
        return -1 ;
    }
}
```

unlock semaphore

```
<<shm: unlock semaphore>>=
syserr = sem_post(shmData->semaphore) ;
if (syserr == -1) {
    *errorCodePtr = errno ;
    return -1 ;
}
```

Output procedure

```
<<shm forward function declarations>>=
static int shmOutputProc(ClientData instanceData, char const *buf, int toWrite,
        int *errorCodePtr) ;
```

Like input, output to the share memory is nothing more than copying data to the shared memory area. Again, there is some book keeping to account for boundaries and we must acquire the semaphore to have exclusive access to the underlying data.

```
<<shm static function definitions>>=
static int
shmOutputProc(
    ClientData instanceData,
    char const *buf,
    int toWrite,
    int *errorCodePtr)
{
    ShmState *shmData = instanceData ;

    <<shm: lock semaphore>>

    size_t available = shmData->length - shmData->position ;
    int canWrite = toWrite < available ? toWrite : available ;

    memcpy(shmData->address + shmData->position, buf, canWrite) ;
    shmData->position += canWrite ;

    <<shm: unlock semaphore>>

    return canWrite ;
}
```

Seek procedure

```
<<shm forward function declarations>>=
static int shmSeekProc(ClientData instanceData, long offset, int seekMode,
        int *errorCodePtr) ;
```

The seeking operation in shared memory is just moving the notion of the current position to a new place in memory. Of course, we must account for boundary conditions and the way relative offsets are specified for the seek operation.

```
<<shm static function definitions>>=
static int
shmSeekProc(
    ClientData instanceData,
    long offset,
    int seekMode,
    int *errorCodePtr)
{
    ShmState *shmData = instanceData ;

    off_t start = 0 ;                   // ❶
    switch (seekMode) {
    case SEEK_SET:
        start = 0 ;
        break ;

    case SEEK_CUR:
        start = shmData->position ;
        break ;

    case SEEK_END:
        start = shmData->length - 1 ;
        break ;

    default:
```

```
        Tcl_Panic("unknown seek mode") ;
        break ;
    }

    off_t newPosition = start + offset ;
    if (newPosition < 0 || newPosition >= shmData->length) {
        *errorCodePtr = EINVAL ;
        return -1 ;
    }

    shmData->position = newPosition ;
    return newPosition ;
}
```

❶ This prevents some compiler warnings.

Watch procedure

```
<<shm forward function declarations>>=
static void shmWatchProc(ClientData instanceData, int mask) ;
```

We set the watch on the file descriptor for the shared memory, but it is not clear that this is a useful thing.

```
<<shm static function definitions>>=
static void
shmWatchProc(
    ClientData instanceData,
    int mask)
{
    ShmState *shmData = instanceData ;

    mask &= shmData->tclMask ;
    if (mask != 0) {
        Tcl_CreateFileHandler(shmData->shmfd, mask,
                (Tcl_FileProc *)Tcl_NotifyChannel, shmData->channel) ;
    } else {
        Tcl_DeleteFileHandler(shmData->shmfd) ;
    }
}
```

Get handle procedure

```
<<shm forward function declarations>>=
static int shmGetHandleProc(ClientData instanceData, int direction,
        ClientData *handlePtr) ;
```

Again, the underlying shared memory identifier is, in reality, an ordinary file descriptor.

```
<<shm static function definitions>>=
static int
shmGetHandleProc(
    ClientData instanceData,
    int direction,
    ClientData *handlePtr)
{
    ShmState *shmData = instanceData ;

    if ((direction & shmData->tclMask) == 0) {
```

```
        return TCL_ERROR ;
    }

    *handlePtr = (ClientData)(intptr_t)(shmData->shmfd) ;
    return TCL_OK ;
}
```

Block mode procedure

```
<<shm forward function declarations>>=
static int shmBlockModeProc(ClientData instanceData, int mode) ;
```

We need only to record the blocking mode so that it may be used to decide how to wait for the exclusion semaphore.

```
<<shm static function definitions>>=
static int
shmBlockModeProc(
    ClientData instanceData,
    int mode)
{
    ShmState *shmData = instanceData ;

    shmData->blockingMode = mode ;

    return 0 ;
}
```

Opening a Shared Memory Channel

Just as for message queues, we provide an external function that may be called by a “C” based extension to open an message queue channel. We don’t expect that to be a common use case, but it is convenient when necessary and we will use the same function to provide the Tcl script level command to open a message queue channel.

```
<<shm external functions definitions>>=
Tcl_Channel
PosixIPC_OpenSHMChannel(
    Tcl_Interp *interp,
    Tcl_Obj *name,
    char const *mode,
    mode_t permissions,
    off_t size)
{
    int flags = 0 ;
    int tclMask = 0 ;
    bool bintrans = false ;

    <<OpenSHMChannel: process arguments>>

    int syserr ;
    char const *shmname = Tcl_GetString(name) ;

    <<OpenSHMChannel: open semaphore>>
    <<OpenSHMChannel: open shared memory>>
    <<OpenSHMChannel: size shared memory>>
    <<OpenSHMChannel: map shared memory>>

    Tcl_Channel shmChan ;
    <<OpenSHMChannel: create shared memory channel>>
```

```
    <<OpenSHMChannel: set translation option>>

    return shmChan ;

release_chan:
    Tcl_Close(interp, shmChan) ;
    ckfree(shmdata) ;

release_shm:
    close(shmfd) ;
    shm_unlink(shmname) ;

release_sem:
    sem_close(shmsem) ;
    sem_unlink(shmname) ;

    return NULL ;
}
```

Like message queues, shared memory also uses the same argument conventions for mode and access.

processing arguments

```
<<OpenSHMChannel: process arguments>>=
if (islower(*mode)) {
    if (processFopenModeOption(interp, mode, &flags, &tclMask, &bintrans) !=
            TCL_OK) {
        return NULL ;
    }
} else {
    if (processOpenModeOption(interp, mode, &flags, &tclMask, &bintrans) !=
            TCL_OK) {
        return NULL ;
    }
}

if ((flags & O_CREAT) != 0 && size <= 0) {              // ❶
    if (interp != NULL) {
        Tcl_SetObjResult(interp,
                Tcl_ObjPrintf("shared memory size must be positive")) ;
    }
    return NULL ;
}
```

❶ A non-positive size is not meaningful.

opening the semaphore

```
<<OpenSHMChannel: open semaphore>>=
int semFlags = flags & (O_CREAT | O_EXCL) ;                    // ❶
sem_t *shmsem = sem_open(shmname, semFlags, permissions, 1) ;
if (shmsem == SEM_FAILED) {
    shmFailure(interp, "cannot open shared memory exclusive access semaphore") ;
    return NULL ;
}
```

❶ The only flags that make sense for a semaphore deal with creation. So we mask off any others. Linux is more forgiving of passing other flags in. FreeBSD is not.

Since it is possible to have system calls related to shared memory fail, we specialize `ipcFailure()` for shared memory use.

```
<<shm forward function declarations>>=
static int shmFailure(Tcl_Interp *interp, char const *msg) ;
```

```
<<shm static function definitions>>=
static int
shmFailure(
    Tcl_Interp *interp,
    char const *msg)
{
    return ipcFailure(interp, "shm", msg) ;
}
```

We need a check on the arguments to make sure don't end up with shared memory of zero length.

```
<<OpenSHMChannel: open shared memory>>=
int shmfd = shm_open(shmname, flags, permissions) ;
if (shmfd < 0) {
    shmFailure(interp, shmname) ;
    goto release_sem ;
}
```

When a shared memory object is created, it has a zero size and we need to `ftruncate()` the underlying file to give it the requested size. The complication here is knowing whether we have actually created the shared memory object or whether it was already there when we tried to open it. The answer is to look at the size of the shared memory object. A zero size indicates we actually created the share memory object and need to expand it to the requested size. Since, the shared memory object is represented by a file descriptor, `fstat()` will give the required information.

```
<<OpenSHMChannel: size shared memory>>=
struct stat shmStatus ;
syserr = fstat(shmfd, &shmStatus) ;
if (syserr < 0) {
    shmFailure(interp, "cannot get status of shared memory") ;
    goto release_shm ;
}

if (shmStatus.st_size == 0) {
    int syserr = ftruncate(shmfd, size) ;
    if (syserr < 0) {
        shmFailure(interp,
                "cannot set the size of shared memory using ftruncate()") ;
        goto release_shm ;
    }
} else {
    size = shmStatus.st_size ;
}
```

The last bit of systems programming is to map the shared memory object into the process address space. Since the shared memory is really a file, then `mmap()` is used to perform the mapping. It's just a matter of computing the flag values based on the direction of data flow.

```
<<OpenSHMChannel: map shared memory>>=
int mapProt = 0 ;
mapProt |= (tclMask & TCL_READABLE) != 0 ? PROT_READ : 0 ;
mapProt |= (tclMask & TCL_WRITABLE) != 0 ? PROT_WRITE : 0 ;

void *mapaddr = mmap(NULL, size, mapProt, MAP_SHARED, shmfd, 0) ;
if (mapaddr == MAP_FAILED) {
    shmFailure(interp, shmname) ;
    goto release_shm ;
}
```

Once all the required system programming is done, we can now turn the shared memory and its associated semaphore into a Tcl channel. We allocate the data object used to deal with the state of the channel and call Tcl to create it.

```
<<OpenSHMChannel: create shared memory channel>>=
char chanName[TCL_INTEGER_SPACE + 16] ;
snprintf(chanName, sizeof(chanName), "posixshm%d", shmfd) ;

ShmState *shmdata = ckalloc(sizeof(ShmState)) ;
shmdata->shmfd = shmfd ;
shmdata->semaphore = shmsem ;
shmdata->tclMask = tclMask ;
shmdata->blockingMode = (flags & O_NONBLOCK) != 0 ?
        TCL_MODE_NONBLOCKING : TCL_MODE_BLOCKING ;
shmdata->address = mapaddr ;
shmdata->length = size ;
shmdata->position = 0 ;
shmdata->channel = shmChan =
        Tcl_CreateChannel(&ShmChannelType, chanName, shmdata, tclMask) ;
```

If the channel was opened in binary mode, we need to instruct the Tcl channel system to set the translation properly.

```
<<OpenSHMChannel: set translation option>>=
if (bintrans) {
    if (Tcl_SetChannelOption(interp, shmChan, "-translation", "binary") !=
            TCL_OK) {
        goto release_chan ;
    }
}
```

Shared Memory Commands

In this section, we present the "C" code that implements the Tcl script level commands for shared memory. Per the usual conventions, we provide a command to open a shared memory channel. The usual Tcl channel commands can then be used to read or write to the channel. Like, message queues, shared memory is special in that it can be unlinked and a separate command is provided for that (*i.e.* unlinking is different than closing).

shm open command

```
posixipc shm open ?options? shmname
posixipc shm open ?options? shmname ?access?
posixipc shm open ?options? shmname ?access? ?permissions?
```

options

-size *bytes*
The size of the shared memory in bytes. This option must be supplied if the shared memory object is to be created. This option is not required if an existing shared memory object is being opened nor can the size of an existing shared memory object be changed as part of opening it.

shmname
The name of shared memory object. Shared memory names must start with a slash (/) character and may **not** contain any other slash characters.

access
The *access* argument, if present, indicates the way in which the shared memory is to be accessed. In the first form, *access* may have any of the following forms:

r
Open the shared memory for reading only. The shared memory must already exist. This is the default if *access* is not specified.

r+
Open the shared memory for reading and writing. The shared memory must already exist.

w
Open the shared memory for writing only. Create the shared memory if it does not already exist.

w+
Open the shared memory for reading and writing. Create the shared memory if it does not already exist.

All legal *access* values above may have the character **b** added as the second or third character in the value to indicate that the opened channel should be configured as if with the **fconfigure -translation binary** option, making the channel suitable for reading or writing of binary data.

In the second form, *access* consists of a list of any of the following flags, all of which have the standard POSIX meanings. Exactly one flags must be either **RDONLY**, **WRONLY**, or **RDWR**.

RDONLY
Open the shared memory for reading only.

WRONLY
Open the shared memory for writing only.

RDWR
Open the shared memory for both reading and writing.

BINARY
Configure the opened channel with the **-translation binary** option.

CREAT
Create the shared memory if it does not already exist.

EXCL
If **CREAT** is also specified, an error is returned if the shared memory already exists.

NONBLOCK
Prevent the process from blocking in subsequent I/O operations.

permissions
If the shared memory is created as part of opening it, *permissions* is used to set the permissions for the new message queue in conjunction withthe process's file mode creation mask. *Permissions* defaults to 0666.

```
<<shm forward function declarations>>=
static int shmOpenCmd(ClientData clientData, Tcl_Interp *interp, int objc,
        Tcl_Obj *const objv[]) ;
```

The following function is executed for the `shm open` command. Like most Tcl command functions, much of its code deals with marshalling and validating arguments.

```
<<shm static function definitions>>=
static int
shmOpenCmd(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const objv[])
{
    static char const usage[] = "?options? sharedname ?access? ?permissions?" ;
    if (objc < 2) {
        Tcl_WrongNumArgs(interp, 1, objv, usage) ;
        return TCL_ERROR ;
    }

    <<shmOpenCmd: establish default option values>>

    <<shmOpenCmd: parse options arguments>>

    <<shmOpenCmd: gather remaining arguments>>

    <<shmOpenCmd: create channel>>

    return TCL_OK ;
}
```

default shared memory options

```
<<shmOpenCmd: establish default option values>>=
char const *mode = "r" ;
mode_t permissions = 0666 ;
off_t shmSize = 0 ;                // ❶
```

❶ Start at zero and expect it to be overridden if we intend to create the shared memory object.

The only argument allowed for shared memory is to set the size for newly created shared memory objects.

```
<<shmOpenCmd: parse options arguments>>=
int objindex = 1 ;
for ( ; objindex < objc ; objindex++) {
    char const *optionName = Tcl_GetString(objv[objindex]) ;
    if (strcmp(optionName, "--") == 0) {
        objindex++ ;
        break ;
    } else if (strcmp(optionName, "-size") == 0) {
        if (++objindex >= objc) {
            Tcl_WrongNumArgs(interp, 1, objv, usage) ;
            return TCL_ERROR ;
        }
        int size ;
        if (Tcl_GetIntFromObj(interp, objv[objindex], &size) != TCL_OK) {
            return TCL_ERROR ;
        }
        shmSize = size ;
```

```
    } else {
        break ;
    }
}
```

The access and permissions arguments are optional.

```
<<shmOpenCmd: gather remaining arguments>>=
int remain = objc - objindex ;
if (remain < 1 || remain > 3) {
    Tcl_WrongNumArgs(interp, 1, objv, usage) ;
    return TCL_ERROR ;
}

if (remain > 1) {
    mode = Tcl_GetString(objv[objindex + 1]) ;
}

if (remain == 3) {
    int permArgValue ;
    if (Tcl_GetIntFromObj(interp, objv[objindex + 2], &permArgValue)
            != TCL_OK) {
        return TCL_ERROR ;
    }
    permissions = permArgValue ;
}
```

To create the channel we invoke the external function supplied for that purpose.

```
<<shmOpenCmd: create channel>>=
Tcl_Channel chan = PosixIPC_OpenSHMChannel(interp, objv[objindex], mode,
        permissions, shmSize) ;
if (chan == NULL) {
    return TCL_ERROR ;
}
Tcl_RegisterChannel(interp, chan) ;
Tcl_SetObjResult(interp, Tcl_NewStringObj(Tcl_GetChannelName(chan), -1)) ;
```

Tests

```
<<shm tests>>=
test shm-open-1.0 {
    Open a shared memory object
} -setup {
    set shmName /shm_open_1.0
} -cleanup {
    chan close $shmchan
    shm unlink $shmName
} -body {
    set shmchan [shm open -size 256 $shmName w+]
    log::debug "shm channel = $shmchan"
    return $shmchan
} -result {posixshm[0-9]*} -match glob
```

```
<<shm tests>>=
test shm-open-1.1 {
    Open a shared memory object with negative size
} -setup {
    set shmName /shm_open_1.1
} -cleanup {
} -body {
```

```
    shm open -size -20 $shmName w+
} -result {shared memory size must be positive} -returnCodes error
```

```
<<shm tests>>=
test shm-open-1.2 {
    Open a shared memory object with bad name
} -setup {
    set shmName /shm_open_1.2/foo
} -cleanup {
} -body {
    shm open -size 256 $shmName w+
} -result {posixipc shm: cannot open shared memory exclusive access\
        semaphore:*} -match glob -returnCodes error
```

```
<<shm tests>>=
test shm-open-1.3 {
    Open a shared memory object with bad access value
} -setup {
    set shmName /shm_open_1.3
} -cleanup {
} -body {
    shm open -size 256 $shmName a
} -result {unknown open mode, "a": should be one of r, r+ rb, rb+, w, w+, wb,\
        or wb+} -returnCodes error
```

```
<<shm tests>>=
test shm-open-1.4 {
    Open a shared memory object with bad access flag
} -setup {
    set shmName /shm_open_1.4
} -cleanup {
} -body {
    shm open -size 256 $shmName {FOOBAR}
} -result {invalid access mode "FOOBAR": must be BINARY, CREAT, EXCL,\
        NONBLOCK, RDONLY, RDWR, or WRONLY} -returnCodes error
```

```
<<shm tests>>=
test shm-open-1.5 {
    Open a shared memory object with bad size spec
} -setup {
    set shmName /shm_open_1.5
} -cleanup {
} -body {
    shm open -size
} -result {wrong # args: should be "shm open ?options? sharedname ?access?\
        ?permissions?"} -returnCodes error
```

```
<<shm tests>>=
test shm-open-1.6 {
    Open a shared memory object with no name
} -setup {
    set shmName /shm_open_1.6
} -cleanup {
} -body {
    shm open -size 256
} -result {wrong # args: should be "shm open ?options? sharedname ?access?\
        ?permissions?"} -returnCodes error
```

```
<<shm tests>>=
```

```
test shm-open-1.7 {
    Open a shared memory object with no name
} -setup {
    set shmName /shm_open_1.6
} -cleanup {
} -body {
    shm open $shmName r+ foobar
} -result {expected integer but got "foobar"} -returnCodes error
```

```
<<shm tests>>=
test shm-open-1.8 {
    Open a shared memory -- too few arguments
} -setup {
} -cleanup {
} -body {
    shm open
} -result {wrong # args: should be "shm open ?options? sharedname ?access?\
        ?permissions?"} -returnCodes error
```

```
<<shm tests>>=
test shm-open-1.9 {
    Open a shared memory -- bad size spec
} -setup {
    set shmName /shm_open_1.9
} -cleanup {
} -body {
    shm open -size foobar $shmName r+
} -result {expected integer but got "foobar"} -returnCodes error
```

```
<<shm tests>>=
test shm-open-1.10 {
    Open a shared memory with non-default permissions
} -setup {
    set shmName /shm_open_1.10
    set shmFile /dev/shm$shmName
} -cleanup {
    chan close $shmchan
    shm unlink $shmName
} -constraints linuxOnly -body {
    set shmchan [shm open -size 1024 $shmName w+ 0640]

    set fattrs [file attributes $shmFile]
    log::debug "shm file attributes = $fattrs"
    dict get $fattrs -permissions
} -result {00640}
```

```
<<shm tests>>=
test shm-open-2.0 {
    Open shared memory, write and read
} -setup {
    set shmName /shm_open_2.0
} -cleanup {
    chan close $shmchan
    shm unlink $shmName
} -body {
    set shmchan [shm open -size 256 -- $shmName w+]
    log::debug "shm channel = $shmchan"
    chan configure $shmchan -buffering line
    chan puts $shmchan "Hello World"
    chan seek $shmchan 0
```

```
    chan gets $shmchan
} -result {Hello World}
```

```
<<shm tests>>=
test shm-open-2.1 {
    Open shared memory, write and read -- non-blocking
} -setup {
    set shmName /shm_open_2.1
} -cleanup {
    chan close $shmchan
    shm unlink $shmName
} -body {
    set shmchan [shm open -size 256 -- $shmName w+]
    log::debug "shm channel = $shmchan"
    chan configure $shmchan -buffering line -blocking false
    chan puts $shmchan "Hello World"
    chan seek $shmchan 0
    chan gets $shmchan
} -result {Hello World}
```

```
<<shm tests>>=
test shm-open-3.0 {
    Read past EOF.
} -setup {
    set shmName /shm_open_3.0
} -cleanup {
    chan close $shmchan
    shm unlink $shmName
} -body {
    set shmchan [shm open -size 256 -- $shmName w+]
    log::debug "shm channel = $shmchan"
    chan configure $shmchan -buffering line
    chan seek $shmchan 0
    chan puts $shmchan [string repeat a 255]
    chan seek $shmchan 0
    set contents [chan gets $shmchan]
    log::debug "got [string length $contents] byte string"
    log::debug "string starts with \"[string index $contents 0]\""
    log::debug "at file location, [chan tell $shmchan]"
    set end [chan read $shmchan 1]
    log::debug "reading past EOF, \"$end\""
    set eof [chan eof $shmchan]
    log::debug "eof = $eof"
    return $eof
} -result {1}
```

shm unlink command

Like message queue, shared memory has a presence in the file system. Unlinking removes the name of the shared memory object and the shared memory is deleted when there are no open references to it. This operation can be done through the file system, but that requires knowledge that the share memory resides in a virtual file system and knowledge of where that file system is mounted. Further, using this call also cleans up any associated exclusion semaphore.

```
posixipc shm unlink shmname
```

shmname
The name of shared memory object. Shared memory object names must start with a slash (/) character and may **not** contain any other slash characters.

The `unlink` command removes the shared memory object named, `shmname`, from the system. The shared memory is actually destroyed when all processes that have open descriptors for the shared memory close those descriptors.

```
<<shm forward function declarations>>=
static int shmUnlinkCmd(ClientData clientData, Tcl_Interp *interp, int objc,
        Tcl_Obj *const objv[]) ;
```

```
<<shm static function definitions>>=
static int
shmUnlinkCmd(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const objv[])
{
    static char const usage[] = "shmname" ;
    if (objc != 2) {
        Tcl_WrongNumArgs(interp, 1, objv, usage) ;
        return TCL_ERROR ;
    }

    char const *shmname = Tcl_GetString(objv[1]) ;

    int shmerr = shm_unlink(shmname) ;
    int semerr = sem_unlink(shmname) ;      // ❶

    if (shmerr < 0 || semerr < 0) {
        return shmFailure(interp, shmname) ;
    }

    return TCL_OK ;
}
```

❶ The associated exclusion semaphore is also unlinked so as not to bleed semaphore names into the file system.

Tests

```
<<shm tests>>=
test shm-unlink-1.0 {
    unlink shared memory -- wrong arguments
} -setup {
} -cleanup {
} -body {
    shm unlink
} -result {wrong # args: should be "shm unlink shmname"} -returnCodes error
```

Code Organization

In this section we describe how the code files are composed from the literate program chunks.

There are two root chunks:

- The “C” source for the package.
- The `tcltest` source for testing the extension.

Package Source Code

The following root chunk shows the general layout of the “C” source code for the package. The ordering of the file is per the requirements of the “C” compiler. For example, since we prefer to place file static functions after the external functions, it is necessary to have forward references to the static functions. Generally, data types definitions must precede data definitions.

```
<<posixipc.c>>=
/*
<<edit warning>>
*/

/*
<<copyright info>>
*/

/*
 * Include Files
 */
#define _POSIX_C_SOURCE 201112L

<<include files>>

/*
 * Macros
 */
<<macros>>

/*
 * Data Types
 */
<<data types>>

/*
 * Forward References
 */
<<forward function declarations>>

/*
 * Static Inline Functions
 */
<<static inline function definitions>>

/*
 * Static Data
 */
<<static data definitions>>

/*
 * External Functions
 */
<<external function definitions>>

/*
 * Static Functions
 */
<<static function definitions>>
```

Include files are collected into a single chunk.

```
<<include files>>=
#include "tcl.h"
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <ctype.h>
#include <string.h>
#include <mqueue.h>
#include <semaphore.h>
```

```
<<macros>>=
#ifndef COUNTOF
#define COUNTOF(a)      (sizeof(a) / sizeof(a[0]))
#endif /* COUNTOF */
<<package macros>>
```

The following chunks define the composition of the chunks defined for the "C" source file root. This composition allows us to order commonly used definitions to appear before definitions for specific portions of the source. For example, we order the aspects dealing with the package as a whole to come before the message queue aspects of the source which come before the shared memory aspects.

```
<<data types>>=
<<common data types>>
<<mq data types>>
<<shm data types>>
```

```
<<static data definitions>>=
<<mq static data>>
<<shm static data>>
```

```
<<forward function declarations>>=
<<common forward function declarations>>
<<mq forward function declarations>>
<<shm forward function declarations>>
```

```
<<static function definitions>>=
<<common static function definitions>>
<<mq static function definitions>>
<<shm static function definitions>>
```

```
<<external function definitions>>=
<<package external function definitions>>
<<mq external functions definitions>>
<<shm external functions definitions>>
```

Unit Tests

Part of the literate program documentation is to show the test cases for the code. In the document, we have included the test cases near to the code they test. In this section, the root chunk for the tests is defined.

The package uses `tcltest` to run and tally the tests.

```
<<posixipc.test>>=
<<edit warning>>
<<copyright info>>

<<required packages>>

# Add custom arguments here.
set optlist {
    {level.arg warn {Logging level}}
}
array set options [::cmdline::getKnownOptions argv $optlist]
tcltest::configure {*}$argv

logger::setlevel $options(level)

if {$::tcl_platform(os) eq "Linux"} {
    load ../tea/build/x86_64-linux-tcl8.6/libposixipc1.0.so
} else {
    package require posixipc
}

namespace eval ::posixipc::test {
    namespace import ::tcltest::*

    testConstraint linuxOnly [expr {$::tcl_platform(os) eq "Linux" ?\
            "true" : "false"}]

    <<set up logger>>
    log::info "testing posixipc version: [package require posixipc]"

    variable mqFSMount
    if {[file exists /dev/mqueue]} {
        set mqFSMount /dev/mqueue
    } elseif {[file exists /mnt/mqueue]} {
        set mqFSMount /mnt/mqueue
    }

    <<package initialization tests>>

    namespace import ::posixipc::mq
    <<mq tests>>
    <<mq examples>>

    namespace import ::posixipc::shm
    <<shm tests>>

    cleanupTests
}
```

We collect all the additional packages required for the tests into one place.

```
<<required packages>>=
package require Tcl 8.6
package require cmdline
package require logger
package require logger::utils
package require logger::appender
package require tcltest
```

The following incantation sets up the logger to use color to output log messages and then imports the logger commands into a child namespace.

```
<<set up logger>>=
set logger [::logger::init posixipc]
::logger::utils::applyAppender -appender colorConsole -serviceCmd $logger\
        -appenderArgs {-conversionPattern {\[%c\] \[%p\] '%m'}}
::logger::import -all -force -namespace log posixipc
```

Copyright Information

The following is copyright and licensing information.

```
<<copyright info>>=
# This software is copyrighted 2018 by G. Andrew Mangogna.
# The following terms apply to all files associated with the software unless
# explicitly disclaimed in individual files.
#
# The authors hereby grant permission to use, copy, modify, distribute,
# and license this software and its documentation for any purpose, provided
# that existing copyright notices are retained in all copies and that this
# notice is included verbatim in any distributions. No written agreement,
# license, or royalty fee is required for any of the authorized uses.
# Modifications to this software may be copyrighted by their authors and
# need not follow the licensing terms described here, provided that the
# new terms are clearly indicated on the first page of each file where
# they apply.
#
# IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR
# DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING
# OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES
# THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF
# SUCH DAMAGE.
#
# THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES,
# INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT.  THIS SOFTWARE
# IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE
# NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS,
# OR MODIFICATIONS.
#
# GOVERNMENT USE: If you are acquiring this software on behalf of the
# U.S. government, the Government shall have only "Restricted Rights"
# in the software and related documentation as defined in the Federal
# Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2).  If you
# are acquiring the software on behalf of the Department of Defense,
# the software shall be classified as "Commercial Computer Software"
# and the Government shall have only "Restricted Rights" as defined in
# Clause 252.227-7013 (c) (1) of DFARs.  Notwithstanding the foregoing,
# the authors grant the U.S. Government and others acting in its behalf
# permission to use and distribute the software in accordance with the
# terms specified in this license.
```

Edit Warning

We want to make sure to warn readers that the source code is generated and not manually written.

```
<<edit warning>>=
#
# DO NOT EDIT THIS FILE!
# THIS FILE IS AUTOMATICALLY GENERATED FROM A LITERATE PROGRAM SOURCE FILE.
```

```
#
```

Literate Programming

The source for this document conforms to asciidoc syntax. This document is also a literate program. The source code for the implementation is included directly in the document source and the build process extracts the source that is then given to the Tcl interpreter. This process is known as *tangle*ing. The program, `atangle`, is available to extract source code from the document source and the `asciidoc` tool chain can be used to produce a variety of different output formats, although PDF is the intended choice.

The goal of a literate program is to explain the logic of the program in an order and fashion that facilitates human understanding of the program and then *tangle* the document source to obtain the Tcl code in an order suitable for the Tcl interpreter. Briefly, code is extracted from the literate source by defining a series of *chunks* that contain the source. A chunk is *defined* by including its name as:

```
<<chunk name>>=
```

The trailing = sign denotes a definition. A chunk definition ends at the end of the source block or at the beginning of another chunk definition. A chunk may be *referenced* from within a chunk definition by using its name without the trailing = sign, as in:

```
<<chunk definition>>=
    <<chunk reference>>
```

Chunk names are arbitrary strings. Multiple definitions with the same name are simply concatenated in the order they are encountered. There are one or more *root chunks* which form the conceptual tree for the source files that are contained in the literate source. By convention, root chunks are named the same as the file name to which they will be tangled. Tangling is then the operation of starting at a root chunk and recursively substituting the definition for the chunk references that are encountered.

For readers that are not familiar with the literate style and who are adept at reading source code directly, the chunks definitions and reordering provided by the tangle operation can be a bit disconcerting at first. You can, of course, examine the tangled source output, but if you read the literate program as a document, you will have to trust that the author managed to arrange the chunk definitions and references in a manner so that the tangled output is acceptable to a language compiler or interpreter.

Index