kucomms Reference Manual

Table of Contents

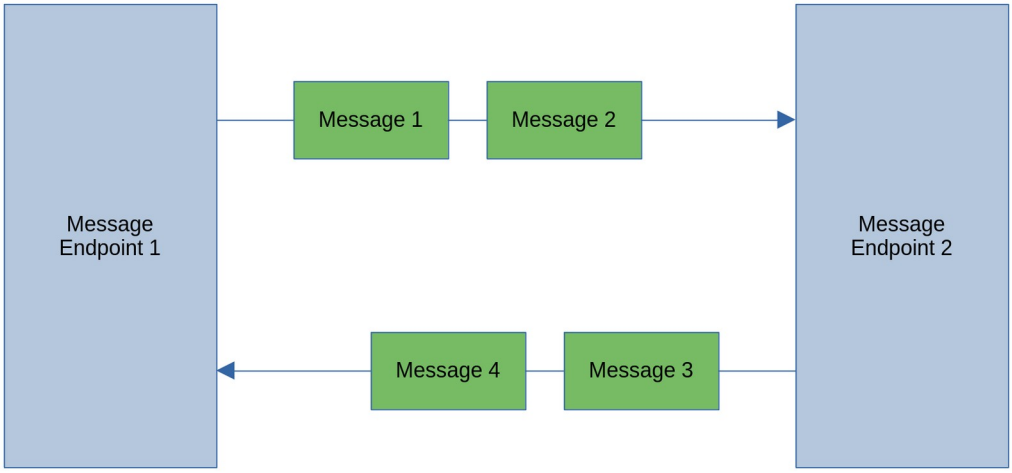
Overview....1
  Asynchronous message passing architecture....1
  Definition of the message....2
  The message endpoint....3
Message endpoints communicate using memory....3
Message endpoints in userspace and kernel....3
The programmers model for using a message endpoint....4
  The user callbacks....4
  The scheduler....4
  Message handlers....4
  Work handlers....4
  Timer handlers....5
  Locking....5
  The API function to send a message....5
The kucomms kernel module and user module....5
The naming scheme used by userspace and user module....5
Creating and removing kucomms character devices....5
Security Considerations....6

Overview

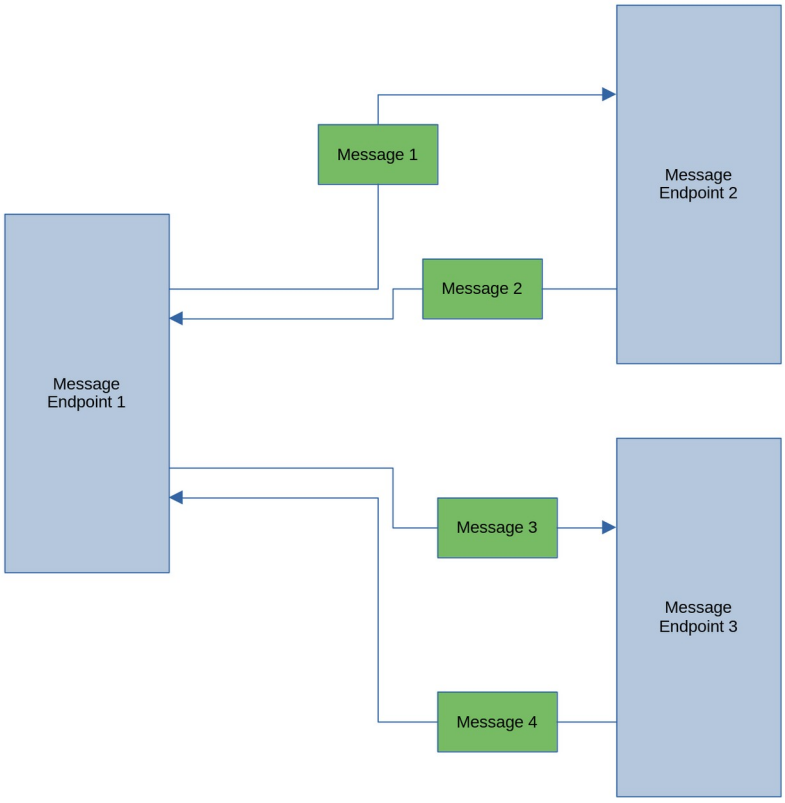
Asynchronous message passing architecture

The kucomms project is built on top of a asynchronous message passing architecture.

A message endpoint is an entity that can send messages and receive messages. Two message endpoints can communicate with each other by sending messages as shown in the diagram below.

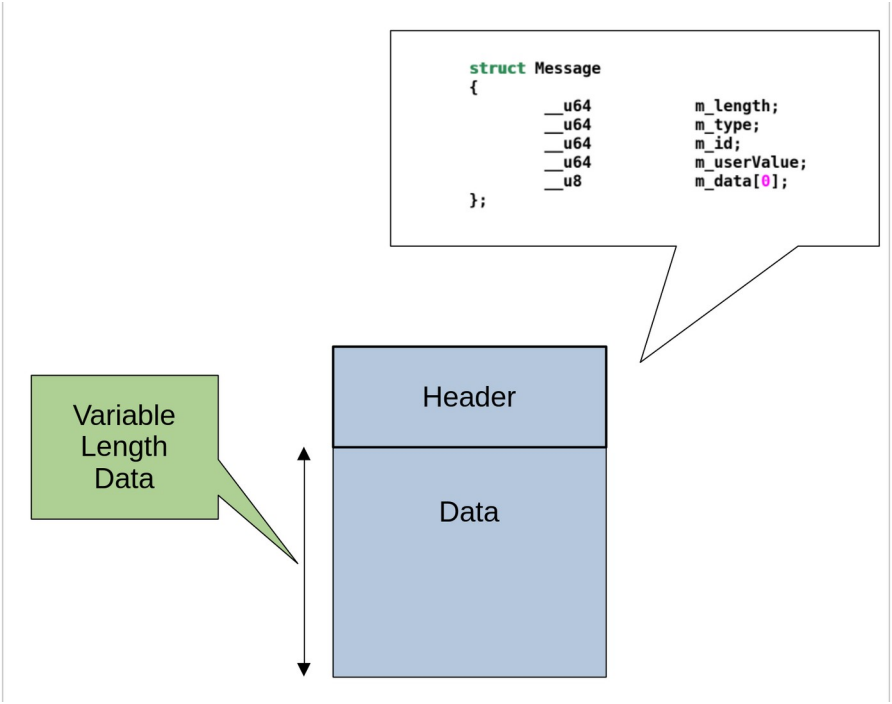


The kucomms project uses the one to one model shown in the diagram above but the messaging framework supports one endpoint connected to multiple endpoints as shown in the diagram below.



Definition of the message

The messages that are sent between endpoints are variable length datagrams defined by the user. The messages have a header and a data section. The layout of a message is shown below.



The fields in the header can be used the by the user in any way as there is no reserved values for the header fields. The user may use the variable length data section in any way. The generic nature of the message allows the two endpoints to define their own protocol for communication which includes a command/response protocol or a notification protocol or any combination of those protocols.

The message endpoint

Each message endpoint provides two important functions. One of the functions is that it provides a C and C++ API that is used to send a message. The other function is that a message endpoint has an internal thread which is responsible for receiving messages and then calling a user callback when a message arrives. The user of a message endpoint has to register a callback which will be called when a message arrives. When a user callback function is called, the message received will be provided to the callback function.

Message endpoints communicate using memory

A message endpoint requires two blocks of memory for it to be able to communicate with another message endpoint. One of the blocks of memory is used for sending messages and the other block of memory is used for receiving messages. Each block of memory is associated with two message endpoints such that one endpoint sends message to the block and the other endpoint receives messages from the block. This arrangement creates a bidirectional communication path between the endpoints.

Message endpoints in userspace and kernel

There are a number of ways message endpoints can be configured, so here is a description of some of the ways they can be configured :

1. Two or more message endpoints inside the same process. In this case, local memory allocated using malloc() is used to connect the message endpoints together.
2. Two or more message endpoints in different processes. In this case, the message endpoints communicate using shared memory allocated using shm_open().
3. One message endpoint in userspace and one message endpoint in the kernel. In this case message endpoints communicate using memory allocated using mmap() which is visible to user space and the kernel. The kucomms project uses this approach.

The programmers model for using a message endpoint

The user callbacks

There are three callbacks that need to be registered with a message endpoint. The three callbacks are known as the message handler, the work handler and the timer handler. As mentioned earlier, the message handler is called when a message arrives. The other two callbacks are optional, if the user does not wish to use the scheduler built into the message endpoint then the user does not need to provide an implementation for those callbacks. All three of the callback functions can call the API function that sends a message.

The scheduler

The message endpoint has an internal thread that is responsible for receiving messages and calling user callbacks. All three user callbacks are called from the context of this thread. The message handler is called when a message arrives and the other two callbacks are called periodically. If one of the callbacks takes a lot of time to execute then the other two callbacks will not be called until the executing callback returns, for this reason it is encouraged to keep callback execution time as short as possible.

Message handlers

The message handler is called when a message arrives as soon as possible.

Work handlers

The work handler is called 100 times a second. This handler is used for implementing a state machine. This handler should execute as quickly as possible. The average execution time of this handler should not exceed 1/100 of a second if possible.

This handler could be used to poll some resource of interest to the user or it could be used to poll a state variable (i.e. a state machine).

Timer handlers

The timer handler is called one times a second. This handler is used for performing timer related functions.

This handler could be used to implement a timeout for a command/response pair.

Locking

All three callbacks are called from the same thread context so there is no need for locking of variables between these three callback functions. If variables are accessed by an external user thread and by the callback functions, then locking will be required for the variables.

The API function to send a message

The function provided by the message endpoint to send a message is not thread safe, that is, it can only be called by one thread at a time. The API provides a locked and unlocked version of the send message function. The locked version calls the unlocked version with a mutex held. The three callbacks are all called from the same thread so they can usually call the unlocked version of the send message function because no other thread is sending messages. If a user wants to send messages from callback functions and also from an external user thread then the locked version of the send message function must be used.

The kucomms kernel module and user module

The kucomms architecture provides a communication path between a user kernel module and a userspace application. The kucomms kernel module provides this communication pathway. The user kernel module must register its three callbacks with the kucomms kernel module. Hence the kucomms kernel module must be inserted first before the user kernel module.

The naming scheme used by userspace and user module

In order for a userspace application to communicate with a user kernel module they must agree on a name. The name must start with the prefix **kucomms_** and a character device having that name must exist in the */dev* directory. For example the name chosen could be **kucomms_mymodule** and a character device */dev/kucomms_mymodule* must exist. When the userspace application starts up it will open the character device. When the user kernel module starts up and registers its three callbacks, it will also specify the name agreed upon.

Creating and removing kucomms character devices

A mechanism has been provided to create and remove kucomms character devices in the */dev* directory.

When the kucomms kernel module is inserted, two new files will appear in the */sys* directory.

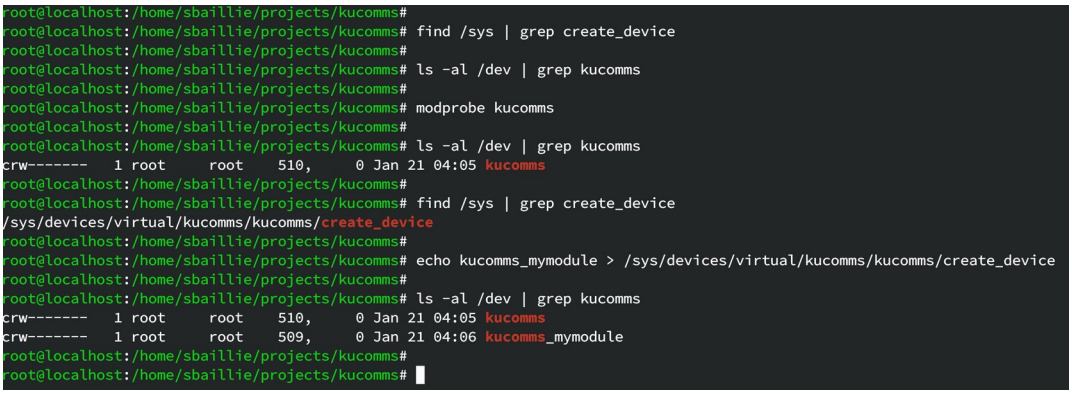
The two new files are :

/sys/devices/virtual/kucomms/kucomms/create_device
/sys/devices/virtual/kucomms/kucomms/remove_device
These files are used to create and remove kucomms character devices.

For example, a character device having the name **kucomms_mymodule** can be created using the shell command :

echo kucomms_mymodule > /sys/devices/virtual/kucomms/kucomms/create_device

See image below for a more detailed example :



Security Considerations

The kucomms API provides an arbitrary message passing mechanism between userspace and kernel modules and hence provides the potential for a security vulnerability to the host machine. A malicious user space application could intentionally try to crash the kernel.

The kucomms infrastructure has been coded with this in mind and the infrastructure can guarantee that no memory overflows will occur (i.e. pointer access beyond the end of an array). The infrastructure can guarantee that when a message handler is called, the message passed to the handler will have a valid length field, nothing else about the message is validated.

It is up to the message handler to validate the message. In the case of a message handler executing in the kernel, it is critical to validate the message before acting on it. The kernel module author should assume that a malicious user space application is sending the messages and perform strict checking to protect the host machine.

In order for a userspace application to send messages to the kernel, it must first open a character device in the /dev directory. A mechanism that will improve security is to set the file permissions of the character device such that only trusted users can send messages to the kernel.