kucomms userspace programmers guide

Table of Contents

Introduction

In order to create a userspace application that can communicate with a kernel module, it is necessary to define three callback functions and then run the main loop of the message endpoint.

Defining and registering callbacks

The first step is to define 3 classes, the user callbacks will be members of these classes.

```
class MyMessageHandler : public MessageHandler
{
public:
        bool hlr(const struct Message * message,
                MessageQueueWriter & tx_msgq,
                std::vector<MessageQueueWriter> & tx_msgq_list);
};

class MyWorkHandler : public WorkHandler
{
public:
        bool hlr(std::vector<MessageQueueWriter> & tx_msgq_list);
};

class MyTimerHandler : public TimerHandler
{
public:
        void hlr(const __u64 time,
                std::vector<MessageQueueWriter> & tx_msgq_list);
};
```

The next step is to declare the handler methods. The methods shown below have no implementation and are examples only.

```
bool
MyMessageHandler::hlr(const struct Message * message,
                MessageQueueWriter & tx_msgq,
                std::vector<MessageQueueWriter> & tx_msgq_list)
{
        return true;
}

bool
MyWorkHandler::hlr(std::vector<MessageQueueWriter> & tx_msgq_list)
{
        return false;
}

void
MyTimerHandler::hlr(const __u64 time,
                std::vector<MessageQueueWriter> & tx_msgq_list)
{
}
```

The last step is to run the main loop of the message endpoint.

```
static bool g_stopped = false;

void
terminate_signal_hanlder(int sig)
{
        g_stopped = true;
}

int main(int argc, char ** argv)
{
        signal(SIGTERM, terminate_signal_hanlder);

        MyMessageHandler msghlr;
        MyWorkHandler workhlr;
        MyTimerHandler timerhlr;

        bool ok = MessageManager::run(
                                "/dev/kucomms_myname"
                                1024*1024,
                                g_stopped,
                                msghlr,
                                workhlr,
                                timerhlr);

        return 0;
}
```

Sending a message

A message is sent by calling the add() method of the object passed to the caller.

Below are two examples of sending a message.

```
bool
MyMessageHandler::hlr(const struct Message * message,
                MessageQueueWriter & tx_msgq,
                std::vector<MessageQueueWriter> & tx_msgq_list)
{
        // Send the message received back to the sender
        bool ok = tx_msgq_list[0].add(message);
        return true;
}

bool
MyWorkHandler::hlr(std::vector<MessageQueueWriter> & tx_msgq_list)
{
        DataMessage msg(10); // Build a message
        Message * message = msg.get();
        __u8 * data = msg.get_data();
        message->m_type = 0;
        message->m_id = id;
        message->m_userValue = 0;
        for (__u32 u0=0; u0<msg.get_data_length(); u0++) data[u0] = 0;

        bool ok = tx_msgq_list[0].add(msg.get()); // Send message

        return false;
}
```

Sending a message and locking

The function to send a message has two variants, a locked version and a unlocked version. If messages are to be sent only from callback handlers then the unlocked version can be called. If messages are to be sent from multiple thread contexts then the locked version must be used.

The unlocked version is called *MessageQueueWriter::add()* and the locked version is called *MessageQueueWriter::add_locked()*.

Writing work handlers

Work handlers are called at a rate of approximately 100 times a second. Work handlers should execute as quickly as possible. The average execution time of a work handler should not exceed 1/100 of a second. If a work handler does not sleep then it should return false. If a work handler sleeps for >= 1/100 of a second then it should return true.