# Kbus Documentation

***Release 0.4 alpha***

**Tony Ibbs**

2016-11-16

Contents


**1 A simple introduction to using KBUS** **3**

**2 KBUS – Lightweight kernel-mediated messaging** **9**
2.1 Summary . . . 10
2.2 Intentions . . . 11
2.3 The basics . . . 11
2.4 Messages . . . 11
2.5 Types of message . . . 17
2.6 KBUS end points - Ksocks . . . 20
2.7 More information . . . 23

**3 KBUS Python bindings for Messages** **31**
3.1 Message . . . 31
3.2 Announcement . . . 37
3.3 Request and stateful_request . . . 39
3.4 Reply and reply_to . . . 41
3.5 MessageId . . . 43
3.6 Status . . . 44
3.7 OrigFrom . . . 45
3.8 split_replier_bind_event_data . . . 46

**4 KBUS Python bindings for Ksocks** **47**
4.1 Ksock . . . 47
4.2 Ksock/C datastructures . . . 51
4.3 Other functions . . . 52

**5 KBUS C bindings** **55**
5.1 kbus/linux/kbus_defns.h . . . 55
5.2 libkbus/kbus.h . . . 68
5.3 libkbus/limpet.h . . . 84

**6 Utilities** **89**
6.1 errno.py . . . 89
6.2 kmsg . . . 89
6.3 runlimpet and runlimpet.py . . . 90

**7 KBUS Limpets - an introduction with goldfish** **91**
7.1 A few more technical details . . . 94

**8 KBUS Limpets - an introduction** **97**
8.1 The problem, in brief . . . 97
8.2 Summary . . . 97
8.3 Restrictions and caveats . . . 97
8.4 With goldfish bowls . . . 98
8.5 Python and C implementations . . . 100
8.6 Network protocol . . . 101

**9 KBUS Python bindings for Limpets** **103**
9.1 Limpets . . . 103

**10 The KBUS documentation and sphinx** **107**
10.1 Pre-built documentation . . . 107
10.2 Building the documentation . . . 107
10.3 The Python bindings . . . 107
10.4 Mime type magic . . . 108
10.5 Mercurial gotchas . . . 108

**11 Indices and tables** **109**

**Python Module Index** **111**

KBUS is a lightweight messaging system for Linux, particularly aimed at embedded platforms. Message passing is managed by a kernel module, via reading/writing `/dev/kbus0` style devices. Python bindings are provided, as is a C library.

Contents:

CHAPTER 1

# A simple introduction to using KBUS

This is intended as a very simple introduction to the basics of how to use KBUS. The examples are not realistic, but should give some flavour of the way that KBUS works.

We shall start with a single "actor" in our virtual playlet:

*Terminal 1: Rosencrantz*

```
$ python
Python 2.6.4 (r264:75706, Dec  7 2009, 18:45:15)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from kbus import *
```

I'm generally against doing an import of `*`, but it's reasonably safe with the KBUS python module, and it makes the tutorial shorter.

First our actor needs to connect to KBUS itself, by opening a Ksock:

*Terminal 1: Rosencrantz*

```
>>> rosencrantz = Ksock(0)
```

This specifies which KBUS device to connect to. If KBUS is installed, then device `0` will always exist, so it is a safe choice. The default is to open the device for read and write - this makes sense since we will want to write messages to it.

Once we've done that, we can try sending a message:

*Terminal 1: Rosencrantz*

```
>>> ahem = Message('$.Actor.Speak', 'Ahem')
>>> rosencrantz.send_msg(ahem)
MessageId(0, 1)
```

The first line creates a new message named `$.Actor.Speak`, with the message data `"Ahem"`.

*(All message names are composed of ''$'' followed by a series of dot-separated parts.)*

The second line sends it. For convenience, the `send_msg` method also returns the *message id* assigned to the message by KBUS - this can be used to identify a specific message.

This will succeed, but doesn't do anything very useful, because no-one is listening. So, we shall need a second process, which we shall start in a new terminal.

*Terminal 2: Audience*

```
$ python
Python 2.6.4 (r264:75706, Dec  7 2009, 18:45:15)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from kbus import *
>>> audience = Ksock(0)
>>> audience.bind('$.Actor.Speak')
```

Here, the audience has opened the same KBUS device (messages cannot be sent between different KBUS devices). We've still opened it for write, since they might, for instance, want to be able to send `$.Applause` messages later on. They've then 'bound to' the `$.Actor.Speak` message, which means they will receive any messages that are sent with that name.

(In fact, all messages with that name sent by anyone, not just by rosencrantz.)

Now, if rosencrantz speaks:

*Terminal 1: Rosencrantz*

```
>>> rosencrantz.send_msg(ahem)
MessageId(0, 2)
```

the audience can listen:

*Terminal 2: Audience*

```
>>> audience.read_next_msg()
Message('$.Actor.Speak', data='Ahem', from_=1L, id=MessageId(0,2))
```

A friendlier representation of the message is given if one prints it:

*Terminal 2: Audience*

```
 >>> print _
<Announcement '$.Actor.Speak', id=[0:2], from=1, data='Ahem'>
```

"Plain" messages are also termed "announcements", since they are just being broadcast to whoever might be listening.

Note that this shows that the message received has the same `MessageId` as the message sent (which is good!).

Of course, if the audience tries to listen again, they're not going to "hear" anything new:

*Terminal 2: Audience*

```
>>> message = audience.read_next_msg()
>>> print message
None
```

and so they really need to set up a loop to wait for messages, something like:

*Terminal 2: Audience*

```
>>> import select
>>> while 1:
...     (r,w,x) = select.select([audience], [], [])
...     # At this point, r should contain audience
...     message = audience.read_next_msg()
...     print 'We heard', message.name, message.data
...
```

(although perhaps with more error checking, and maybe even a timeout, in a real example).

So if rosencrantz speaks again:

*Terminal 1: Rosencrantz*

```
>>> rosencrantz.send_msg(Message('$.Actor.Speak', 'Hello there'))
MessageId(0, 3)
>>> rosencrantz.send_msg(Message('$.Actor.Speak', 'Can you hear me?'))
MessageId(0, 4)
```

the audience should be able to hear him:

*Terminal 2: Audience*

```
We heard $.Actor.Speak Hello there
We heard $.Actor.Speak Can you hear me?
```

So now we'll introduce another participant:

*Terminal 3: Guildenstern*

```
$ python
Python 2.6.4 (r264:75706, Dec  7 2009, 18:45:15)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from kbus import *
>>> guildenstern = Ksock(0)
>>> guildenstern.bind('$.Actor.*')
```

Here, guildenstern is binding to any message whose name starts with `$.Actor.`. In retrospect this, of course, makes sense for the audience, too - let's fix that:

*Terminal 2: Audience*

```
<CTRL-C>
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
KeyboardInterrupt
>>> audience.bind('$.Actor.*')
>>> while 1:
...     msg = audience.wait_for_msg()
...     print 'We heard', msg.name, msg.data
...
```

(as a convenience, the Ksock class provides the `wait_for_msg()` wrapper around `select.select`, which is shorter to type...).

And maybe rosencrantz will want to hear his colleague:

*Terminal 1: Rosencrantz*

```
>>> rosencrantz.bind('$.Actor.*')
```

So let guildenstern speak:

*Terminal 3: Guildenstern*

```
>>> guildenstern.send_msg(Message('$.Actor.Speak', 'Pssst!'))
MessageId(0, 5)
>>> # Remember guildenstern is also listening to '$.Actor.*'
>>> print guildenstern.read_next_msg()
<Announcement '$.Actor.Speak', id=[0:5], from=3, data='Pssst!'>
```

and rosencrantz hears:

*Terminal 1: Rosencrantz*

```
>>> print rosencrantz.read_next_msg()
<Announcement '$.Actor.Speak', id=[0:5], from=3, data='Pssst!'>
```

However, when we look to the audience, we see:

*Terminal 2: Audience*

```
We heard $.Actor.Speak Pssst!
We heard $.Actor.Speak Pssst!
```

This is because the audience has bound to the message twice - it is hearing it once because it asked to receive every `$.Actor.Speak` message, and again because it asked to hear any message matching `$.Actor.*`.

The solution is simple - ask not to hear the more specific version:

*Terminal 2: Audience*

```
<CTRL-C>
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
KeyboardInterrupt
>>> audience.unbind('$.Actor.Speak')
>>> while 1:
...     msg = audience.wait_for_msg()
...     print 'We heard', msg.from_, 'say', msg.name, msg.data
...
```

Note that we've also amended the printout to say who the message was from. Each Ksock connection has an id associated with it - for instance:

*Terminal 1: Rosencrantz*

```
>>> rosencrantz.ksock_id()
1L
```

and every message indicates who sent it, so:

*Terminal 1: Rosencrantz*

```
>>> print 'I heard', message.from_, 'say', message.name, message.data
I heard 3 say $.Actor.Speak Pssst!
```

We've shown that KBUS allows one to "announce" (or, less politely, "shout") messages, but KBUS also supports asking questions. Thus:

*Terminal 3: Guildenstern*

```
>>> guildenstern.bind('$.Actor.Guildenstern.query', True)
```

allows Guildenstern to bind to this new message name as a Replier.

*(Only one person may be bound as Replier for a particular message name at any one time, so that it is unambiguous who is expected to do the replying.*

*Also, if a Sender tries to send a Request, but no-one has bound to that message name as a Replier, then an error is raised (contrast that with ordinary messages, where if no-one is listening, the message just gets ignored).)*

If Rosencrantz then sends a Request of that name:

*Terminal 1: Rosencrantz*

```
>>> req = Request('$.Actor.Guildenstern.query', 'Were you speaking to me?')
>>> rosencrantz.send_msg(req)
MessageId(0, 6)
```

Guildenstern can receive it:

*Terminal 3: Guildenstern*

```
>>> msg2 = guildenstern.read_next_msg()
>>> print 'I heard', msg2
I heard <Request '$.Actor.Guildenstern.query', id=[0:6], from=1, flags=0x3 (REQ,YOU), data='Were
>>> msg3 = guildenstern.read_next_msg()
>>> print msg3
<Request '$.Actor.Guildenstern.query', id=[0:6], from=1, flags=0x1 (REQ), data='Were you speakin
```

As we should expect, guildenstern is getting the message twice, once because he has bound as a listener to '$.Actor.*', and once because he is bound as a Replier to this specific message.

> *(There is, in fact, a way to ask KBUS to only deliver one copy of a given message, and if guildenstern had used that, he would only have received the Request that was marked for him to answer. I'm still a little undecided how often this mechanism should be used, though.)*

Looking at the two messages, the first is the Request specifically to guildenstern, which he is meant to answer:

*Terminal 3: Guildenstern*

```
>>> print msg2.wants_us_to_reply()
True
```

(and that is what the `YOU` in the flags means).

And rosencrantz himself will also have received a copy:

*Terminal 1: Rosencrantz*

```
>>> print rosencrantz.read_next_msg()
<Request '$.Actor.Guildenstern.query', id=[0:6], from=1, flags=0x1 (REQ), data='Were you speakin
```

Guildenstern can then reply:

*Terminal 3: Guildenstern*

```
>>> reply = reply_to(msg2, 'Yes, I was')
>>> print reply
<Reply '$.Actor.Guildenstern.query', to=1, in_reply_to=[0:6], data='Yes, I was'>
>>> guildenstern.send_msg(reply)
MessageId(0, 7)
```

The `reply_to` convenience function crafts a new `Reply` message, with the various message parts set in an appropriate manner. And thus:

*Terminal 1: Rosencrantz*

```
>>> rep = rosencrantz.read_next_msg()
>>> print 'I heard', rep.from_, 'say', rep.name, rep.data
I heard 3 say $.Actor.Guildenstern.query Yes, I was
```

Note that Rosencrantz didn't need to bind to this message to receive it - he will always get a Reply to any Request he sends (KBUS goes to some lengths to guarantee this, so that even if Guildenstern closes his Ksock, it will generate a "gone away" message for him).

And, of course:

*Terminal 2: Audience*

```
We heard 1 say $.Actor.Guildenstern.query Were you speaking to me?
We heard 3 say $.Actor.Guildenstern.query Yes, I was
```

So, in summary:

- To send or receive messages, a process opens a Ksock.
- A process can send messages (be a Sender).
- A process can bind to receive messages (be a Listener) by message name.
- When binding to a message name, wildcards can be used.
- When binding to a message name, a process can say it wants to receive Requests with that name (be a Replier)
- It is not an error to send an ordinary message if no-one is listening.
- It is an error to send a Request if there is no Replier.
- There can only be one Replier for a given message name.
- There can be any number of Listeners for a given message name.

---

**Note:** Running the examples in this introduction requires having the KBUS kernel module installed. If this is not already done, and you have the KBUS sources, then `cd` to the kernel module directory (i.e., `kbus` in the sources) and do:

```
make
make rules
sudo insmod kbus.ko
```

When you've finished the examples, you can remove the kernel module again with:

```
sudo rmmod kbus.ko
```

The message ids shown in the examples are correct if you've just installed the kernel module - the second number in each message id will be different (although always ascending) otherwise.

---

CHAPTER 2

# KBUS – Lightweight kernel-mediated messaging

**Contents**


- KBUS – Lightweight kernel-mediated messaging
  - Summary
  - Intentions
  - The basics
    - Python and C
  - Messages
    - Message names
    - Message ids
    - Message content
      - Unset
      - The message header
      - Message implementation
      - Limits
    - Message flags
      - Send flags
    - Things KBUS changes in a message
  - Types of message
    - Announcements
    - Request message
    - Reply message
    - Status message
    - Requests and Replies
  - KBUS end points - Ksocks
    - The KBUS devices
    - Ksocks
    - Senders
    - Listeners
    - Repliers
  - More information
    - Stateful transactions
    - Queues filling up
    - Urgent messages
    - Select, write/send and "next message", blocking
    - Receiving messages once only
    - IOCTLS
    - /proc/kbus/bindings
    - /proc/kbus/stats
    - Error numbers


## 2.1 Summary

KBUS provides lightweight kernel-mediated messaging for Linux.

- "lightweight" means that there is no intent to provide complex or sophisticated mechanisms - if you need something more, consider DBUS or other alternatives.
- "kernel-mediated" means that the actual business of message passing and message synchronisation is handled by a kernel module.
- "for Linux" means what it says, since the Linux kernel is required.

Initial use is expected to be in embedded systems.

There is (at least initially) no intent to aim for a "fast" system - this is not aimed at real-time systems.

Although the implementation is kernel-mediated, there is a mechanism ("Limpets") for commnicating KBUS messages between buses and/or systems.

## 2.2 Intentions

KBUS is intended:

- To be simple to use and simple to understand.
- To have a small codebase, written in C.
- To provide predictable message delivery.
- To give deterministic message ordering.
- To guarantee a reply to every request.

It needs to be simple to use and understand because the expected users are typically busy with other matters, and do not have time to spend learning a complex messaging system.

It needs to have a small codebase, written in C, because embedded systems often lack resources, and may not have enough space for C++ libraries, or messaging systems supporting more complex protocol stacks.

Our own experience on embedded systems of various sizes indicates that the last three points are especially important.

Predictable message delivery means the user can know whether they can tell in what circumstances messages will or will not be received.

Deterministic message ordering means that all recipients of a given set of messages will receive them in the same order as all other recpients (and this will be the order in which the messages were sent). This is important when several part of (for instance) an audio/video stack are interoperating.

Guaranteeing that a request will always result in a reply means that the user will be told if the intended replier has (for instance) crashed. This again allows for simpler use of the system.

## 2.3 The basics

### 2.3.1 Python and C

Although the KBUS kernel module is written in C, the module tests are written in Python, and there is a Python module providing useful interfaces, which is expected to be the normal way of using KBUS from Python.

There is also a C library (libkbus) which provides a similar level of abstraction, so that C programmers can use KBUS without having to handle the low level details of sockets and message datastructures. Note that the C programer using KBUS does need to have some awareness of how KBUS messages work in order to get memory management right.

## 2.4 Messages

### 2.4.1 Message names

All messages have names - for instance "$.Sensors.Kitchen".

All message names start with "$.", followed by one or more alphanumeric words separated by dots. There are two wildcard characters, "*" and "%", which can be the last word of a name.

Thus (in some notation or other):

```
name := '$.'  [ word '.' ]+  ( word  | '*' | '%' )
word := alphanumerics
```

Case is significant. There is probably a limit on the maximum size of a subname, and also on the maximum length of a message name.

Names form a name hierarchy or tree - so "$.Sensors" might have children "$.Sensors.Kitchen" and "$.Sensors.Bedroom".

If the last word of a name is "*", then this is a wildcard name that also includes all the child names at that level and below – i.e., all the names that start with the name up to the "*". So "$.Sensors.*" includes "$.Sensors.Kitchen", "$.Sensors.Bedroom", "$.Sensors.Kitchen.FireAlarm", "$.Sensors.Kitchen.Toaster", "$.Sensors.Bedroom.FireAlarm", and so on.

If the last word of a name is "%", then this is a wildcard name that also includes all the child names at that level – i.e., all the names obtained by replacing the "%" by another word. So "$.Sensors.%" includes "$.Sensors.Kitchen" and "$.Sensors.Bedroom", but not "$.Sensors.Kitchen.Toaster".

### 2.4.2 Message ids

Every message is expected to have a unique id.

A message id is made up of two parts, a network id and a serial number.

The network id is used to carry useful information when a message is transferred from one KBUS system to another (for instance, over a bridge). By default (for local messages) it is 0.

A serial number is used to identify the particular message within a network.

If a message is sent via KBUS with a network id of 0, then KBUS itself will assign a new message id to the message, with the network id (still) 0, and with the serial number one more than the last serial number assigned. Thus for local messages, message ids ascend, and their order is deterministic.

If a message is sent via KBUS with a non-zero network id, then KBUS does not touch its message id.

Network ids are represented textually as `{n,s}`, where `n` is the network id and `s` is the serial number.

> Message id {0,0} is reserved for use as an invalid message id. Both network id and serial number are unsigned 32-bit integers. Note that this means that local serial numbers will eventually wrap.

### 2.4.3 Message content

Messages are made of the following parts:

**start and end guards** These are unsigned 32-bit words. 'start_guard' is notionally "Kbus", and 'end_guard' (the 32 bit word after the rest of the message) is notionally "subK". Obviously that depends on how one looks at the 32-bit word. Every message shall start with a start guard and end with an end guard (but see *Message implementation* for details).

These provide some help in checking that a message is well formed, and in particular the end guard helps to check for broken length fields.

If the message layout changes in an incompatible manner (this has happened once, and is strongly discouraged), then the start and end guards change.

### Unset

Unset values are 0, or have zero length (as appropriate).

It is not possible for a message name to be unset.

### The message header

**message id** identifies this particular message. This is made up of a network id and a serial number, and is discussed in *Message ids*.

When replying to a message, copy this value into the 'In reply to' field.

**in_reply_to** is the message id of the message that this is a reply to.

This shall be set to 0 unless this message *is* a reply to a previous message. In other words, if this value is non-0, then the message *is* a reply.

**to** is the Ksock id identifying who the message is to be sent to.

When writing a new message, this should normally be set to 0, meaning "anyone listening" (but see below if "state" is being maintained).

When replying to a message, it shall be set to the 'from' value of the orginal message.

When constructing a request message (a message wanting a reply), then it can be set to a specific replier's Ksock id. When such a message is sent, if the replier bound (at that time) does not have that specific Ksock id, then the send will fail.

**from** indicates the Ksock id of the message's sender.

When writing a new message, set this to 0, since KBUS will set it.

When reading a message, this will have been set by KBUS.

**orig_from** this indicates the original sender of a message, when being transported via Limpet. This will be documented in more detail in the future.

**final_to** this indicates the final target of a message, when being transported via Limpet. This will be documented in more detail in the future.

**extra** this is a zero field, for future expansion. KBUS will always set this field to zero.

**flags** indicates extra information about the message. See *Message Flags* for detailed information.

When writing a message, typical uses include:

- the message is URGENT
- a reply is wanted

When reading a message, typical uses include:

- the message is URGENT
- a reply is wanted
- a reply is wanted from the specific reader

The top 16 bits of the flags field is reserved for use by the user - KBUS will not touch it.

**name_length** is the length of the message name in bytes. This will always be non-zero, as a message name must always be given.

**data_length** is the length of the message data in bytes. It may be zero if there is no data associated with this message.

**name** identifies the message. It must be terminated with a zero byte (as is normal for C - in the Python binding a normal Python string can be used, and the this will be done for you). Byte ordering is according to that of the platform.

In an "entire" message (see *Message implementation* below) the name shall be padded out to a multiple of 4 bytes. Neither the terminating zero byte nor the padding are included in the name length. Padding should be with zero bytes.

**data** is optional. KBUS does not touch the content of the data, but just copies it. Byte ordering is according to that of the platform.

In an "entire" message (see *Message implementation* below) the data shall, if present, be padded out to a multiple of 4 bytes. This padding is not included in the data length, and the padding bytes may be whatever byte values are convenient to the user. KBUS does not guarantee to copy the exact given padding bytes (in fact, current implementations just ignore them).

### Message implementation

There are two ways in which a message may be constructed, "pointy" and "entire". See the `kbus_defns.h` header file for details.

---

**Note:** The Python binding hides most of the detail of the message implementation from the user, so if you are using Python you may be able to skip this section.

---

In a "pointy" message, the `name` and `data` fields in the message header are C pointers to the actual name and data. If there is no data, then the `data` field is NULL. This is probably the simplest form of message for a C programmer to create. This might be represented as:

```
start_guard: 'Kbus'
id:          (0,0)
in_reply_to: (0,0)
to:          0
from:        0
name_len:    6
data_len:    0
name:        ---------------------------> "$.Fred"
data:        NULL
end_guard:   'subK'
```

or (with data):

```
start_guard: 'Kbus'
id:          (0,0)
in_reply_to: (0,0)
to:          0
from:        0
name_len:    6
data_len:    7
name:        ---------------------------> "$.Fred"
data:        ---------------------------> "abc1234"
end_guard:   'subK'
```

**Warning:** When writing a "pointy" message in C, be very careful not to free the name and data between the `write` and the SEND, as it is only when the message is sent that KBUS actually follows the `name` and `data` pointers.
*After* the SEND, KBUS will have taken its own copies of the name and (any) data.

In an "entire" message, both `name` and `data` fields are required to be NULL. The message header is followed by the message name (padded as described above), any message data (also padded), and another end guard. This might be represented as:

```
start_guard: 'Kbus'
id:          (0,0)
in_reply_to: (0,0)
to:          0
from:        0
name_len:    6
data_len:    0
name:        NULL
data:        NULL
end_guard:   'subK'
name_data:   '$.Fred\x0\x0'
end_guard:   'subK'
```

or (again with data):

```
start_guard: 'Kbus'
id:          (0,0)
in_reply_to: (0,0)
to:          0
from:        0
name_len:    6
data_len:    7
name:        NULL
data:        NULL
end_guard:   'subK'
name_data:   '$.Fred\x0\x0'
data_data:   'abc1234\x0'
end_guard:   'subK'
```

Note that in these examples:

1. The message name is padded out to 6 bytes of name, plus one of terminating zero byte, plus another zero byte to make 8, but the message's `name_len` is still 6.
2. When there is no data, there is no "data data" after the name data.
3. When there is data, the data is presented after the name, and is padded out to a multiple of 4 bytes (but without the necessity for a terminating zero byte, so it is possible to have no pad bytes if the data length is already a multiple of 4). Again, the `data_len` always reflects the "real" data length.
4. Although the data shown is presented as ASCII strings for these examples, it really is just bytes, with no assumption of its content/meaning.

When writing/sending messages, either form may be used (again, the "pointy" form may be simpler for C programmers).

When reading messages, however, the "entire" form is always returned - this removes questions about needing to free multiple returned datastructures (for instance, what to do if the user were to ask for the NEXTMSG, read a few bytes, and then DISCARD the rest).

### Limits

Message names may not be shorter than 3 characters (since they must be at least "$." plus another character). An arbitrary limit is also placed on the maximum message length - this is currently 1000 characters, but may be reviewed

in the future.

Message data may, of course, be of zero length.

When reading a message, an "entire" message is always returned.

---

**Note:** When using C to work with KBUS messages, it is generally ill-advised to reference the message name and data "directly":

```
char    *name = msg->name;
uint8_t *data = msg->data;
```

since this will work for "pointy" messages, but not for "entire" messages (where the `name` field will be NULL). Instead, it is always better to do:

```
char    *name = kbus_msg_name_ptr(msg);
uint8_t *data = kbus_msg_data_ptr(msg);
```

regardless of the message type.

---

### 2.4.4 Message flags

KBUS reserves the bottom 16 bits of the flags word for predefined purposes (although not all of those bits are yet used), and guarantees not to touch the top 16 bits, which are available for use by the programmer as a particular application may wish.

The WANT_A_REPLY bit is set by the sender to indicate that a reply is wanted. This makes the message into a request.

> Note that setting the WANT_A_REPLY bit (i.e., a request) and setting 'in_reply_to' (i.e., a reply) is bound to lead to confusion, and the results are undefined (i.e., don't do it).

The WANT_YOU_TO_REPLY bit is set by KBUS on a particular message to indicate that the particular recipient is responsible for replying to (this instance of the) message. Otherwise, KBUS clears it.

The SYNTHETIC bit is set by KBUS when it generates a Status message, for instance when a replier has gone away and will therefore not be sending a reply to a request that has already been queued.

> Note that KBUS does not check that a sender has not set this flag on a message, but doing so may lead to confusion.

The URGENT bit is set by the sender if this message is to be treated as urgent - i.e., it should be added to the *front* of the recipient's message queue, not the back.

#### Send flags

There are two "send" flags, ALL_OR_WAIT and ALL_OR_FAIL. Either one may be set, or both may be unset.

> If both are set, the message will be rejected as invalid.
>
> Both flags are ignored in reply messages (i.e., messages with the 'in_reply_to' field set).

If a message has ALL_OR_FAIL set, then a SEND will only succeed if the message could be added to all the (intended) recipient's message queues. Otherwise, SEND returns -EBUSY.

If a message has ALL_OR_WAIT set, then a SEND will only succeed if the message could be added to all the (intended) recipient's message queues. Otherwise SEND returns -EAGAIN. In this case, the message is still being

sent, and the caller should either call DISCARD (to drop it), or else use poll/select to wait for the send to finish. It will not be possible to call "write" until the send has completed or been discarded.

These are primarily intended for use in debugging systems. In particular, note that the mechanisms dealing with ALL_OR_WAIT internally are unlikely to be very efficient.

---

**Note:** The send flags will be less effective when messages are being mediated via Limpets, as remote systems are involved.

---

### 2.4.5 Things KBUS changes in a message

In general, KBUS leaves the content of a message alone - mostly so that an individual KBUS module can "pass through" messages from another domain. However, it does change:

- the message id's serial number (but only if its network id is unset)
- the 'from' id (to indicate the Ksock this message was sent from)
- the WANT_YOU_TO_REPLY bit in the flags (set or cleared as appropriate)
- the SYNTHETIC bit, which will always be unset in a message sent by a Sender

KBUS will always set the 'extra' field to zero.

Limpets will change:

- the network id in any field that has one.
- the 'orig_from' and 'final_to' fields (which in general should only be manipulated by Limpets).

## 2.5 Types of message

There are four basic message types:

- Announcement – a message aimed at any listeners, expecting no reply
- Request – a message aimed at a replier, who is expected to reply
- Reply – a reply to a request
- Status – a message generated by KBUS

The Python interface provides a Message base class, and subclasses thereof for each of the "user" message types (but not currently for Status).

### 2.5.1 Announcements

An announcement is the "plain" message type. It is a message that is being sent for all bound listeners to "hear".

When creating a new announcement message, it has:

**message id** see *Message ids*

**in reply to** unset (it's not a reply)

**to** unset (all announcements are broadcast to any listeners)

**from** unset (KBUS will set it)

**flags** typically unset, see *Message flags*

**message name** as appropriate

**message data** as appropriate

The Python interface provides an `Announcement` class to help in creating an announcement message.

### 2.5.2 Request message

A request message is a message that wants a reply.

Since only one Ksock may bind as a replier for a given message name, a request message wants a reply from a single Ksock. By default, this is whichever Ksock has bound to the message name at the moment of sending, but see *Stateful transactions*.

When creating a new request message, it has:

**message id** see *Message ids*

**in reply to** unset (it's not a reply)

**to** either unset, or a specific Ksock id if the request should fail if that Ksock is (no longer) the replier for this message name

**from** unset (KBUS will set it)

**flags** the "needs a reply" flag should be set. KBUS will set the "you need to reply" flag in the copy of the message delivered to its replier.

**message name** as appropriate

**message data** as appropriate

When receiving a request message, the WANT_YOU_TO_REPLY flag will be set if it is this recipient's responsibility to reply.

The Python interface provides a `Request` class to help in creating a request message.

When a request message is sent, it is an error if there is no replier bound to that message name.

The message will, as normal, be delivered to all listeners, and will have the "needs a reply" flag set wherever it is received. However, only the copy of the message received by the replier will be marked with the WANT_YOU_TO_REPLY flag.

> So, if a particular file descriptor is bound as listener and replier for '$.Fred', it will receive two copies of the original message (one marked as needing reply from that file descriptor). However, when the reply is sent, only the "plain" listener will receive a copy of the reply message.

### 2.5.3 Reply message

A reply message is the expected response after reading a request message.

A reply message is distinguished by having a non-zero 'in reply to' value.

Each reply message is in response to a specific request, as indicated by the 'in reply to' field in the message.

The replier is helped to remember that it needs to reply to a request, because the request has the WANT_YOU_TO_REPLY flag set.

When a reply is sent, all listeners for that message name will receive it. However, the original replier will not.

When creating a new reply message, it has:

**message id** see *Message ids*

**in reply to** the request message's 'message id'

**to** the request message's 'from' id

**from** unset (KBUS will set it)

**flags** typically unset, see *Message flags*

**message name** the request message's 'message name'

**message data** as appropriate

The Python interface provides a `Reply` class to help in creating a reply message, but more usefully there is also a `reply_to` function that creates a Reply Message from the original Request.

### 2.5.4 Status message

KBUS generates Status messages (also sometimes referred to as "synthetic" messages) when a request message has been successfully sent, but the replier is unable to reply (for instance, because it has closed its Ksock). KBUS thus uses a Status message to provide the "reply" that it guarantees the sender will get.

As you might expect, a KBUS status message is thus (technically) a reply message.

A status message looks like:

**message id** as normal

**in reply to** the 'message id' of the message whose sending or processing caused this message.

**to** the Ksock id of the recipient of the message

**from** the Ksock id of the sender of the message - this will be 0 if the sender is KBUS itself (which is assumed for most exceptions)

**flags** typically unset, see *Message flags*

**message name** for KBUS exceptions, a message name in '$.KBUS.*'

**message data** for KBUS exceptions, normally absent

KBUS status messages always have '$.KBUS.<something>' names (this may be a multi-level <something>), and are always in response to a previous message, so always have an 'in reply to'.

### 2.5.5 Requests and Replies

KBUS guarantees that each Request will (eventually) be matched by a consequent Reply (or Status [1]) message, and only one such.

The "normal" case is when the replier reads the request, and sends its own reply back.

If a Request message has been successfully SENT, there are the following other cases to consider:

1. The replier unbinds from that message name before reading the request message from its queue. In this case, KBUS removes the message from the repliers queue, and issues a "$.KBUS.Replier.Unbound" message.
2. The replier closes itself (close the Ksock), but has not yet read the message. In this case, KBUS issues a "$.KBUS.Replier.GoneAway" message.
3. The replier closes itself (closes the Ksock), has read the message, but has not yet (and now cannot) replied to it. In this case, KBUS issues a "$.KBUS.Replier.Ignored" message.

[1] Remember that a Status message is essentially a specialisation of a Reply message.

4. SEND did not complete, and the replier closes itself before the message can be added to its message queue (by the POLL mechanism). In this case, KBUS issues a "$.KBUS.Replier.Disappeared" message.
5. SEND did not complete, and an error occurs when the POLL mechanims tries to send the message. In this case, KBUS issues a "$.KBUS.ErrorSending" message.

In all these cases, the 'in_reply_to' field is set to the original request's message id. In the first three cases, the 'from' field will be set to the Ksock id of the (originally intended) replier. In the last two cases, that information is not available, and a 'from' of 0 (indicating KBUS itself) is used.

---

**Note:** Limpets introduce some extra messages, which will be documented when the proper Limpet documentation is written.

---

## 2.6 KBUS end points - Ksocks

### 2.6.1 The KBUS devices

Message interactions happen via the KBUS devices. Installing the KBUS kernel module always creates `/dev/kbus0`, it may also create `/dev/kbus1`, and so on.

The number of devices to create is indicated by an argument at module installation, for instance:

```
# insmod kbus.ko num_kbus_devices=10
```

Messages are sent by writing to a KBUS device, and received by reading from the same device. A variety of useful ioctls are also provided. Each KBUS device is independent - messages cannot be sent from `/dev/kbus0` to `/dev/kbus1`, since there is no shared information.

### 2.6.2 Ksocks

Specifically, messages are written to and read from KBUS device file descriptors. Each such is termed a *Ksock* - this is a simpler term than "file descriptor", and has some resonance with "socket".

Each Ksock may be any (one or more) of:

- a Sender (opening the device for read/write)
- a Listener (only needing to open the device for read)
- a Replier (opening the device for read/write)

Every Ksock has an id. This is a 32-bit unsigned number assigned by KBUS when the device is opened. The value 0 is reserved for KBUS itself.

The terms "listener id", "sender id", "replier id", etc., thus all refer to a Ksock id, depending on what it is being used for.

### 2.6.3 Senders

Message senders are called "senders". A sender should open a Ksock for read and write, as it may need to read replies and error/status messages.

A message is sent by:

1. Writing the message to the Ksock (using the standard `write` function)

2. Calling the SEND ioctl on the Ksock, to actually send the message. This returns (via its arguments) the message id of the message sent. It also returns status information about the send

    The status information is to be documented.

The DISCARD ioctl can be used to "throw away" a partially written message, before SEND has been called on it.

If there are no listeners (of any type) bound to that message name, then the message will be ignored.

If the message is flagged as needing a reply, and there are no repliers bound to that message name, then an error message will be sent to the sender, by KBUS.

It is not possible to send a message with a wildcard message name.

> As a restriction this makes the life of the implementor and documentor easier. I believe it would also be confusing if provided.

The sender does not need to bind to any message names in order to receive error and status messages from KBUS.

When a sender sends a Request, an internal note is made that it expects a corresponding Reply (or possible a Status message from KBUS if the Replier goes away or unbinds from that message name, before replying). A place for that Reply is reserved in the sender's message queue. If the message queue fills up (either with messages waiting to be read, or with reserved slots for Replies), then the sender will not be able to send another Request until there is room on the message queue again.

> Hopefully, this can be resolved by the sender reading a message off its queue. However, if there are no messages to be read, and the queue is all reserved for replies, the only solution is for the sender to wait for a replier to send it something that it can then read.

---

**Note:** What order do we describe things in? Don't forget:

If the message being sent is a request, then the replier bound to that message name will (presumably) write a reply to the request. Thus the normal sequence for a request is likely to be:

1. write the request message
2. read the reply

The sender does *not* need to bind to anything in order to receive a reply to a request it has sent.

> Of course, if a sender binds to listen to the name it uses for its request, then it will get a copy of the request as sent, and it will also get (an extra) copy of the reply. But see *Receiving messages once only*.

---

### 2.6.4 Listeners

Message recipients are called "listeners".

Listeners indicate that they want to receive particular messages, by using the BIND ioctl on a Ksock to specify the name of the message that is to be listened for. If the binding is to a wildcarded message name, then the listener will receive all messages with names that match the wildcard.

An ordinary listener will receive all messages with that name (sent to the relevant Ksock). A listener may make more than one binding on the same Ksock (indeed, it is allowed to bind to the same name more than once).

Messages are received by:

1. Using the NEXTMSG ioctl to request the next message (this also returns the messages length in bytes)
2. Calling the standard `read` function to read the message data.

If NEXTMSG is called again, the next message will be readied for reading, whether the previous message has been read (or partially read) or not.

If a listener no longer wants to receive a particular message name, then they can unbind from it, using the UNBIND ioctl. The message name and flags used in an UNBIND must match those in the corresponding BIND. Any messages in the listener's message queue which match that unbinding will be removed from the queue (i.e., the listener will not actually receive them). This does *not* affect the message currently being read.

> Note that this has implication for binding and unbinding wildcards, which must also match.

Closing the Ksock also unbinds all the message bindings made on it. It does not affect message bindings made on other Ksocks.

### 2.6.5 Repliers

Repliers are a special sort of listener.

For each message name, there may be a single "replier". A replier binds to a message name in the same way as any other listener, but sets the "replier" flag. If someone else has already bound to the same Ksock as a replier for that message name, the request will fail.

Repliers only receive Requests (messages that are marked as wanting a reply).

A replier may (should? must?) reply to the request - this is done by sending a Reply message through the Ksock from which the Request was read.

It is perfectly legitimate to bind to a message as both replier and listener, in which case two copies of the message will be read, once as replier, and once as (just) listener (but see *Receiving messages once only*).

When a request message is read by the appropriate replier, KBUS will mark *that particular message* with the "you must reply" flag. This will not be set on copies of that message read by any (non-replier) listeners.

> So, in the case where a Ksock is bound as replier and listener for the same message name, only one of the two copies of the message received will be marked as "you must reply".

If a replier binds to a wildcarded message name, then they are the *default* replier for any message names satisfying that wildcard. If another replier binds to a more specific message name (matching that wildcard), then the specific message name binding "wins" - the wildcard replier will no longer receive that message name.

> In particular '$.Fred.Jim' is more specific than '$.Fred.%' which in turn is more specific than '$.Fred.*'

This means that if a wildcard replier wants to guarantee to see all the messages matching their wildcard, they also need to bind as a listener for the same wildcarded name.

For example:

> Assume message names are of the form '$.Sensors.<Room>' or '$.Sensors.<Room>.<Measurement>'.
>
> Replier 1 binds to '$.Sensors.*'. They will be the default replier for all sensor requests.
>
> Replier 2 binds to '$.Sensors.%'. They will take over as the default replier for any room specific requests.
>
> Replier 3 binds to '$.Sensors.Kitchen.Temperature'. They will take over as the replier for the kitchen temperature.
>
> So:
>
> - A message named '$.Sensors.Kitchen.Temperature' will go to replier 3.
> - A message named '$.Sensors.Kitchen' or '$.Sensors.LivingRoom' will go to replier 2.
> - A message named '$.Sensors.LivingRoom.Temperature' will go to replier 1.

When a Replier is closed (technically, when its `release` function is called by the kernel) KBUS traverses its outstanding message queue, and for each Request that has not been answered, generates a Status message saying that the Replier has "GoneAway".

Similarly, if a Replier unbinds from replying to a mesage, KBUS traverses its outstanding message queue, and for each Request that has not been answered, it generates a Status message saying that it has "Unbound" from being a replier for that message name. It also forgets the message, which it is now not going to reply to.

Lastly, when a Replier is closed, if it has read any Requests (technically, called NEXTMSG to pop them from the message queue), but not actually replied to them, then KBUS will send an "Ignored" Status message for each such Request.

## 2.7 More information

### 2.7.1 Stateful transactions

It is possible to make stateful message transactions, by:

1. sending a Request
2. receiving the Reply, and noting the Ksock id of the replier
3. sending another Request to that specific replier
4. and so on

Sending a request to a particular Ksock will fail if that Ksock is no longer bound as replier to the relevant message name. This allows a sender to guarantee that it is communicating with a particular instance of the replier for a message name.

### 2.7.2 Queues filling up

Messages are sent by a mechanism which:

1. Checks the message is plausible (it has a plausible message name, and the right sort of "shape")
2. If the message is a Request, checks that the sender has room on its message queue for the (eventual) Reply.
3. Finds the Ksock ids of all the listeners and repliers bound to that messages name
4. Adds the message to the queue for each such listener/replier

This can cause problems if one of the queues is already full (allowing infinite expansion of queues would also cause problems, of couse).

If a *sender* attempts to send a Request, but does not have room on its message queue for the (corresponding) Reply, then the message will not be sent, and the send will fail. Note that the message id will not be set, and the blocking behaviours defined below do not occur.

If a *replier* cannot receive a particular message, because its queue is full, then the message will not be sent, and the send will fail with an error. This does, however, set the message id (and thus the "last message id" on the sender).

Moreover, a sender can indicate if it wants a message to be:

1. Added to all the listener queues, regardless, in which case it will block until that can be done (ALL_OR_WAIT, sender blocks)
2. Added to all the listener queues, and fail if that can't be done (ALL_OR_FAIL)
3. Added to all the listener queues that have room (the default)

See *Message flags* for more details.

### 2.7.3 Urgent messages

Messages may be flagged urgent. In this case they will be added to the front of the destination message queue, rather than the end - in other words, they will be the next message to be "popped" by NEXTMSG.

Note that this means that if two urgent messages are sent to the same target, and *then* a NEXTMSG/read occurs, the second urgent message will be popped and read first.

### 2.7.4 Select, write/send and "next message", blocking

**Warning:** At the moment, `read` and `write` are always non-blocking.

`read` returns more of the currently selected message, or EOF if there is no more of that message to read (and thus also if there is no currently selected message). The NEXTMSG ioctl is used to select ("pop") the next message.

`write` writes to the end of the currently-being-written message. The DISCARD ioctl can be used to discard the data written so far, and the SEND ioctl to send the (presumably completed message). Whilst the message is being sent, it is not possible to use `write`.

Note that if SEND is used to send a Request, then KBUS ensures that there will always be either a Reply or a Status message in response to that request.

Specifically, if:

1. The Replier "goes away" (and its "release" function is called) before reading the Request (specifically, before calling NEXTMSG to pop it from the message queue)
2. The Replier "goes away" (and its "release" function is called) before replying to a Request that it has already read (i.e., used NEXTMSG to pop from the message queue)
3. The Replier unbinds from that Request message name before reading the Request (with the same caveat on what that means)
4. Select/poll attempts to send the Request, and discovers that the Replier has disappeared since the initial SEND
5. Select/poll attempts to send the Request, and some other error occurs

then KBUS will "reply" with an appropriate Status message.

---

KBUS support its own particular variation on blocking of message sending.

First of all, it supports use of "select" to determine if there are any messages waiting to be read. So, for instance (in Python):

```
with Ksock(0,'rw') as sender:
    with Ksock(0,'r') as listener:
        (r,w,x) = select.select([listener],[],[],0)
        assert r == []

        listener.bind('$.Fred')
        msg = Announcement('$.Fred','data')
        sender.send_msg(msg)

        (r,w,x) = select.select([listener],[],[],0)
        assert r == [listener]
```

This simply checks if there is a message in the Ksock's message list, waiting to be "popped" with NEXTMSG.

Secondly, `write`, SEND and DISCARD interact in what is hoped to be a sensible manner. Specifically:

- When SEND (i.e., the SEND ioctl) is called, KBUS can either:
  1. Succeed in sending the message. The Ksock is now ready for `write` to be called on it again.
  2. Failed in sending the message (possibly, if the message was a Request, with EADDRNOTAVAIL, indicating that there is no Replier for that Request). The Ksock is now ready for `write` to be called on it again.
  3. If the message was marked ALL_OR_WAIT, then it may fail with EAGAIN. In this case, the Ksock is still in sending state, and an attempt to call `write` will fail (with EALREADY). The caller can either use DISCARD to discard the message, or use select/poll to wait for the message to finish sending.

Thus "select" for the write case checks whether it is allowed to call "write" - for instance:

```
with Ksock(0,'rw') as sender:
    write_list = [sender]
    with Ksock(0,'r') as listener1:
        write_list= [sender,listener1]
        read_list = [listener1]

        (r,w,x) = select.select(read_list,write_list,[],0)
        assert r == []
        assert w == [sender]
        assert x == []

        with Ksock(0,'rw') as listener2:
            write_list.append(listener2)
            read_list.append(listener2)

            (r,w,x) = select.select(read_list,write_list,[],0)
            assert r == []
            assert len(w) == 2
            assert sender in w
            assert listener2 in w
            assert x == []
```

### 2.7.5 Receiving messages once only

In normal usage (and by default), if a Ksock binds to a message name multiple times, it will receive multiple copies of a message. This can happen:

- explicitly (the Ksock deliberately and explicitly binds to the same name more than once, seeking this effect).
- as a result of binding to a message name and a wildcard that includes the same name, or two overlapping wildcards.
- as a result of binding as Replier to a name, and also as Listener to the same name (possibly via a wildcard). In this case, multiple copies will only be received when a Request with that name is made.

Several programmers have complained that the last case, in particular, is very inconvenient, and thus the "receive a message once only" facility has been added.

Using the MSGONCEONLY IOCTL, it is possible to tell a Ksock that only one copy of a particular message should be received, even if multiple are "due". In the case of the Replier/Listener copies, it will always be the message to which the Replier should reply (the one with WANT_YOU_TO_REPLY set) that will be received.

Please use this facility with care, and only if you really need it.

### 2.7.6 IOCTLS

The KBUS ioctls are defined (with explanatory comments) in the kernel module header file (`kbus_defns.h`). They are:

**RESET** Currently has no effect

**BIND** Bind to a particular message name (possibly as replier).

**UNBIND** Unbind from a binding - must match exactly.

**KSOCKID** Determine the Ksock id of the Ksock used

**REPLIER** Determine who is bound as replier to a particular message name. This returns 0 or the Ksock id of the replier.

**NEXTMSG** Pop the next message from the Ksock's message queue, ready for reading (with `read`), and return its length (in bytes). If there is no next message, return a length of 0. The length is always the length of an "entire" message (see *Message implementation*).

**LENLEFT** Determine how many bytes of the message currently being read are still to read.

**SEND** Send the current outstanding message for this Ksock (i.e., the bytes written to the Ksock since the last SEND or DISCARD). Return the message id of the message, and maybe other status information.

**DISCARD** Discard (throw away) the current outstanding message for this Ksock (i.e., any bytes written to the Ksock since the last SEND or DISCARD).

**LASTSENT** Determine the message id of the last message SENT on this Ksock.

**MAXMSGS** Set the maximum length of the (read) message queue for this KSOCK, and return the actual length that is set. An attempt to set the queue length to 0 will just return the current queue length.

**NUMMSGS** Determine how many messages are outstanding in this Ksock's read queue.

**UNREPLIEDTO** Determines how many Requests (marked "WANT_YOU_TO_REPLY") this Ksock still needs to reply to. This is primarily a development tool.

**MSGONLYONCE** Determines whether only one copy of a message will be received, even if the message name is bound to multiple times. May also be used to query the current state.

**VERBOSE** Determines whether verbose kernel messages should be output or not. Affects the *device* (the entire Ksock). May also be used to query the current state.

**NEWDEVICE** Requests another KBUS device (`/dev/kbus/<n>`). The next KBUS device number (up to a maximum of 255) will be allocated. Returns the new device number.

**REPORTREPLIERBINDS** Request synthetic messages announcing Replier BIND/UNBIND events. These are messages named "$.KBUS.ReplierBindEvent", and are the only predefined messages with data. Both Python and C bindings provide a useful function to extract the `is_bind`, `binder` and `name` values from the data.

**MAXMSGSIZE** Set the maximum size of a KBUS message for this KBUS device, and return the value that is set. This is the size of the largest message that may be written to a KBUS Ksock. Trying to write a longer message will result in an -EMSGSIZE error. An attempt to set this value of 0 will just return the current maximum size. Otherwise, the size requested may not be less than 100, or more than the kernel configuration value KBUS_ABS_MAX_MESSAGE_SIZE. The default maximum size is set by the kernel configuration value KBUS_DEF_MAX_MESSAGE_SIZE, and is typically 1024. The size being tested is that returned by the KBUS_ENTIRE_MESSAGE_LEN macro - i.e., the size of an equivalent "entire" message.

### 2.7.7 /proc/kbus/bindings

`/proc/kbus/bindings` is a debugging aid for reporting the listener id, exclusive flag and message name for each binding, for each kbus device.

An example might be:

```
$ cat /proc/kbus/bindings
# <device> is bound to <Ksock-ID> in <process-PID> as <Replier|Listener> for <message-name>
  1:        1    22158  R  $.Sensors.*
  1:        2    22158  R  $.Sensors.Kitchen.Temperature
  1:        3    22158  L  $.Sensors.*
 13:        4    22159  L  $.Jim.*
 13:        1    22159  R  $.Fred
 13:        1    22159  L  $.Jim
 13:       14    23021  L  $.Jim.*
```

This describes two KBUS devices (`/dev/kbus1` and `/dev/kbus13`).

The first has bindings on Ksock ids 1, 2 and 3, for the given message names. The "R" indicates a replier binding, the "L" indicates a listener (non-replier) binding.

The second has bindings on Ksock ids 4, 1 and 14. The order of the bindings reported is *not* particularly significant.

Note that there is no communication between the two devices, so Ksock id 1 on device 1 is not related to (and has no commonality with) Ksock id 1 on device 13.

### 2.7.8 /proc/kbus/stats

`/proc/kbus/stats` is a debugging aid for reporting various statistics about the KBUS devices and the Ksocks open on them.

An example might be:

```
$ cat /proc/kbus/stats
dev  0: next file 5 next msg 8 unsent unbindings 0
        ksock 4 last msg 0:7 queue 1 of 100
            read byte 0 of 0, wrote byte 52 (max 60), sending
            outstanding requests 0 (size 16, max 0), unsent replies 0 (max 0)
        ksock 3 last msg 0:5 queue 0 of 1
            read byte 0 of 0, wrote byte 0 (max 0), not sending
            outstanding requests 1 (size 16, max 0), unsent replies 0 (max 0)
```

or:

```
$ cat /proc/kbus/stats
dev  0: next file 4 next msg 101 unsent unbindings 0
        ksock 3 last msg 0:0 queue 100 of 100
              read byte 0 of 0, wrote byte 0 (max 0), not sending
              outstanding requests 0 (size 16, max 0), unsent replies 0 (max 0)
        ksock 2 last msg 0:100 queue 0 of 100
              read byte 0 of 0, wrote byte 0 (max 0), not sending
              outstanding requests 100 (size 102, max 92), unsent replies 0 (max 0)
```

### 2.7.9 Error numbers

The following error numbers get special use. In Python, they are all returned as values inside the IOError exception.

Since we're trying to fit into the normal Un*x convention that negative values are error numbers, and since Un*x defines many of these for us, it is natural to make use of the relevant definitions. However, this also means that we are often using them in an unnatural sense. I've tried to make the error numbers used bear at least a vague relationship to their (mis)use in KBUS.

**EADDRINUSE** On attempting to bind a message name as replier: There is already a replier bound for this message

**EADDRNOTAVAIL** On attempting to send a Request message: There is no replier bound for this message's name.

On attempting to send a Reply message: The sender of the original request (i.e., the Ksock mentioned as the `to` in the Reply) is no longer connected.

**EALREADY** On attempting to write to a Ksock, when a previous send has returned EAGAIN. Either DISCARD the message, or use select/poll to wait for the send to complete, and write to be allowed.

**EBADMSG** On attempting to bind, unbind or send a message: The message name is not valid. On sending, this can also be because the message name is a wildcard.

**EBUSY** On attempting to send, then:

1. For a request, the replier's message queue is full.
2. For any message, with ALL_OR_FAIL set, one of the targetted listener/replier queues was full.

**ECONNREFUSED** On attempting to send a Reply, the intended recipient (the notional original sender of the Request) is not expecting a Reply with that message id in its 'in_reply_to'. Or, in other words, this appears to be an attempt to reply to the wrong message id or the wrong Ksock.

**EINVAL** Something went wrong (generic error).

**EMSGSIZE** On attempting to write a message: Data was written after the end of the message (i.e., after the final end guard of the message), or an attempt was made to write a message that is too long (see the MAXMSGSIZE ioctl).

**ENAMETOOLONG** On attempting to bind, unbind or send a message: The message name is too long.

**ENOENT** On attempting to open a Ksock: There is no such device (normally because one has tried to open, for instance, '/dev/kbus9' when there are only 3 KBUS devices).

**ENOLCK** On attempting to send a Request, when there is not enough room in the sender's message queue to guarantee that it can receive a reply for every Request already sent, *plus* this one. If there are oustanding messages in the sender's message queue, then the solution is to read some of them. Otherwise, the sender will have to wait until one of the Repliers replies to a previous Request (or goes away and KBUS replies for it).

When this error is received, the send has failed (just as if the message was invalid). The sender is not left in "sending" state, nor has the message been assigned a message id.

Note that this is *not* EAGAIN, since we do not want to block the sender (in the SEND) if it is up to the sender to perform a read to sort things out.

**ENOMSG** On attempting to send, when there is no message waiting to be sent (either because there has been no write since the last send, or because the message being written has been discarded).

**EPIPE** On attempting to send 'to' a specific replier, the replier with that id is no longer bound to the given message's name.

**EFAULT** Memory allocation, copy from user space, or other such failed. This is normally very bad, it should not happen, UNLESS it is the result of calling an ioctl, when it indicates that the ioctl argument cannot be accessed.

**ENOMEM** Memory allocation failed (return NULL). This is normally very bad, it should not happen.

**EAGAIN** On attempting to send, the message being sent had ALL_OR_WAIT set, and one of the targetted listener/replier queues was full.

On attempting to unbind when Replier Bind Events have been requested, one or more of the KSocks bound to receive "$.KBUS.ReplierBindEvent" messages has a full message queue, and thus cannot receive the unbind event. The unbind has not been done.

In the `utils` directory of the KBUS sources, there is a script called `errno.py` which takes an `errno` integer or name and prints out both the "normal" meaning of that error number, and also (if there is one) the KBUS use of it. For instance:

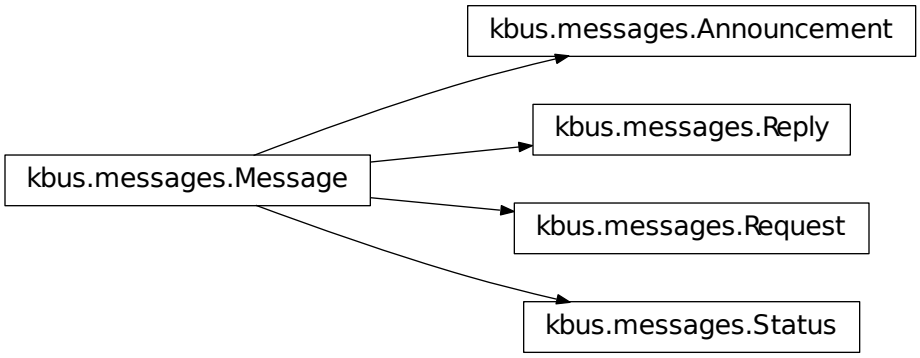
```
$ errno.py 1
Error 1 (0x1) is EPERM: Operation not permitted
$
$ errno.py EPIPE
EPIPE is error 32 (0x20): Broken pipe

KBUS:
On attempting to send 'to' a specific replier, the replier with that id
is no longer bound to the given message's name.
```

CHAPTER 3

# KBUS Python bindings for Messages

KBUS lightweight message system.




## 3.1 Message

**class** kbus.**Message**(*name*, *data=None*, *to=None*, *from_=None*, *orig_from=None*, *final_to=None*, *in_reply_to=None*, *flags=None*, *id=None*)

Bases: object

A wrapper for a KBUS message

A Message can be created in a variety of ways. Perhaps most obviously:

```
>>> msg = Message('$.Fred')
>>> msg
Message('$.Fred')
```

```
>>> msg = Message('$.Fred', '1234')
>>> msg
Message('$.Fred', data='1234')
```

```
>>> msg = Message('$.Fred', '12345678')
>>> msg
Message('$.Fred', data='12345678')
```

```
>>> msg1 = Message('$.Fred', data='1234')
>>> msg1
Message('$.Fred', data='1234')
```

A *Message* can be constructed from another message directly:

```
>>> msg2 = Message.from_message(msg1)
>>> msg2 == msg1
True
```

or from the tuple returned by *extract()*:

```
>>> msg3 = Message.from_sequence(msg1.extract())
>>> msg3 == msg1
True
```

or from an equivalent list:

```
>>> msg3 = Message.from_sequence(list(msg1.extract()))
>>> msg3 == msg1
True
```

or one can use a "string" – for instance, as returned by the `Ksock.read()` method:

```
>>> msg_as_string = msg1.to_bytes()
>>> msg4 = Message.from_bytes(msg_as_string)
>>> msg4 == msg1
True
```

Some testing is made on the first argument - a printable string must start with `$.` (KBUS itself will make a more stringent test when the message is sent):

```
>>> Message('Fred')
Traceback (most recent call last):
...
ValueError: Message name "Fred" does not start "$."
```

and a data "string" must be plausible - that is, long enough for the minimal message header:

```
>>> Message.from_bytes(msg_as_string[:8])
Traceback (most recent call last):
...
ValueError: Cannot form entire message from string "Kbus\x00\x00\x00\x00" of length 8
```

and starting with a message start guard:

```
>>> Message.from_bytes('1234'+msg_as_string)
Traceback (most recent call last):
...
ValueError: Cannot form entire message from string "1234Kbus..1234subK" which does not start wit
```

When constructing a message from another message, one may override particular values (but not the name):

```
>>> msg5 = Message.from_message(msg1, to=9, in_reply_to=MessageId(0, 3))
>>> msg5
Message('$.Fred', data='1234', to=9L, in_reply_to=MessageId(0, 3))
```

```
>>> msg5a = Message.from_message(msg1, to=9, in_reply_to=MessageId(0, 3))
>>> msg5a == msg5
True
```

However, whilst it is possible to set (for instance) *to* back to 0 by this method:

```
>>> msg6 = Message.from_message(msg5, to=0)
>>> msg6
Message('$.Fred', data='1234', in_reply_to=MessageId(0, 3))
```

(and the same for any of the integer fields), it is not possible to set any of the message id fields to None:

```
>>> msg6 = Message.from_message(msg5, in_reply_to=None)
>>> msg6
Message('$.Fred', data='1234', to=9L, in_reply_to=MessageId(0, 3))
```

If you need to do that, go via the *extract()* method:

```
>>> (id, in_reply_to, to, from_, orig_from, final_to, flags, name, data) = msg5.extract()
>>> msg6 = Message(name, data, to, from_, None, None, flags, id)
>>> msg6
Message('$.Fred', data='1234', to=9L)
```

For convenience, the parts of a Message may be retrieved as properties:

```
>>> print msg1.id
None
>>> msg1.name
'$.Fred'
>>> msg1.to
0L
>>> msg1.from_
0L
>>> print msg1.in_reply_to
None
>>> msg1.flags
0L
>>> msg1.data
'1234'
```

Message ids are objects if set:

```
>>> msg1 = Message('$.Fred', data='1234', id=MessageId(0, 33))
>>> msg1
Message('$.Fred', data='1234', id=MessageId(0, 33))
>>> msg1.id
MessageId(0, 33)
```

The arguments to Message() are:

- *arg* – this is the initial argument, and is a message name (a string that starts `$.`), a Message, or a string representing an "entire" message.

  If *arg* is a message name, or another Message then the keyword arguments may be used (for another Message, they override the values in that Message). If *arg* is a message-as-a-string, any keyword arguments will be ignored.

- *data* is data for the Message, either `None` or a Python string.

- *to* is the Ksock id for the destination, for use in replies or in stateful messaging. Normally it should be left 0.

- *from_* is the Ksock id of the sender. Normally this should be left 0, as it is assigned by KBUS.
- if *in_reply_to* is non-zero, then it is the Ksock id to which the reply shall go (taken from the *from_* field in the original message). Setting *in_reply_to* non-zero indicates that the Message *is* a reply. See also the Reply class, and especially the *reply_to* function, which makes constructing replies simpler.
- *flags* can be used to set the flags for the message. If all that is wanted is to set the `Message.WANT_A_REPLY` flag, it is simpler to use the `Request` class to construct the message.
- *id* may be used to set the message id, although unless the *network_id* field is set, KBUS will ignore this and set the id internally (this can be useful when constructing a message to compare received messages against).

Our internal values are:

- *msg*, which is the actual message datastructure.

---

**Note:** Message data is always held as the appropriate C datastructure (via ctypes), mainly to try to minimise copying of data in and out of that form. A "pointy" or "entire" form is used as appropriate.

The message fields (inside *msg*) are readable directly (as properties of `Message`), but are not directly writable. Setter methods (`set_urgent()` and `set_want_reply()`) are provided for those which are likely to be sensible to alter in normal use.

If you need to alter the Message contents, beyond use of the setter methods, then you will need to do so via the internal *msg* datastructure, with a clear understanding of the KBUS datastructure. If you need an example of doing this, see the Limpet codebase (which changes the *id*, *orig_from* and *final_to* fields, not something normal code should need or want to do).

---

**ALL_OR_FAIL = 512L**

**ALL_OR_WAIT = 256L**

**END_GUARD = 1264743795**

**START_GUARD = 1937072715**

**SYNTHETIC = 4L**

**URGENT = 8L**

**WANT_A_REPLY = 1L**

**WANT_YOU_TO_REPLY = 2L**

**cast**()
Return (a copy of) ourselves as an appropriate subclass of `Message`.

Reading from a `Ksock` returns a `Message`, whatever the actual message type. Normally, this is OK, but sometimes it would be nice to have an actual message of the correct class.

**data**
Returns the payload of this KBUS message as a Python string, or None if it is not present.

**equivalent**(*other*)
Returns true if the two messages are mostly the same.

For purposes of this comparison, we ignore *id*, *flags*, *in_reply_to* and *from_*.

**extract**()
Return our parts as a tuple.

The values are returned in something approximating the order within the message itself:

(*id*, *in_reply_to*, *to*, *from_*, *orig_from*, *final_to*, *flags*, *name*, *data*)

This is not the same order as the keyword arguments to *Message()*.

**final_to**
The *final_to* field of the message header. This is of type *OrigFrom*.

**flags**
The *flags* field of the message header.

**from_**
The *from* field of the message header.

**static from_bytes**(*arg*)
Construct a *Message* from bytes, as read by *Ksock.read_data()*.

For instance:

```
>>> msg1 = Message('$.Fred', '12345678')
>>> msg1
Message('$.Fred', data='12345678')
>>> msg2 = Message.from_bytes(msg1.to_bytes())
>>> msg2
Message('$.Fred', data='12345678')
```

**static from_message**(*msg*, *data=None*, *to=None*, *from_=None*, *orig_from=None*, *final_to=None*, *in_reply_to=None*, *flags=None*, *id=None*)
Construct a *Message* from another message.

All the values in the old message, except the name, may be changed by specifying new values in the argument list.

For instance:

```
>>> msg1 = Message('$.Fred', '12345678')
>>> msg1
Message('$.Fred', data='12345678')
>>> msg2 = Message.from_message(msg1, flags=1)
>>> msg2
Message('$.Fred', data='12345678', flags=0x00000001)
```

**static from_sequence**(*seq*, *data=None*, *to=None*, *from_=None*, *orig_from=None*, *final_to=None*, *in_reply_to=None*, *flags=None*, *id=None*)
Construct a *Message* from a sequence, as returned by *extract*.

All the values in the old message, except the name, may be changed by specifying new values in the argument list.

For instance:

```
>>> msg1 = Message('$.Fred', '12345678')
>>> msg1
Message('$.Fred', data='12345678')
>>> msg2 = Message.from_sequence(msg1.extract(), flags=1)
>>> msg2
Message('$.Fred', data='12345678', flags=0x00000001)
```

**id**
The *id* field of the message header.

**in_reply_to**
The *in_reply_to* field of the message header.

**is_reply**()
A convenience method - are we a Reply?

**is_request**()
A convenience method - are we a Request?

**is_stateful_request**()
A convenience method - are we a Stateful Request?

**is_synthetic**()
Return `True` if this is a synthetic message - one generated by KBUS.

**is_urgent**()
Return `True` if this is an urgent message.

**name**
The *name* field of the message header. Any padding bytes are removed.

**orig_from**
The *orig_from* field of the message header. This is of type *OrigFrom*.

**set_urgent**(*value=True*)
Set or unset the 'urgent message' flag.

**set_want_reply**(*value=True*)
Set or unset the 'we want a reply' flag.

**to**
The *to* field of the message header.

**to_bytes**()
Return the message as a string.

This returns the entirety of the message as a Python string.

In order to do this, it first coerces the mesage to an "entire" message (so that we don't have any dangling "pointers" to the name or data).

See the *total_length()* method for how to determine the "correct" length of this string.

**total_length**()
Return the total length of this message.

A Message may be held in one of two ways:

- "pointy" - this is a message header, with references to the message name and data.
- "entire" - this is a message header with the message name and data (and an extra end guard) appended to it.

Message construction may produce either of these (although construction of a message from a string will always produce an "entire" message). Reading a message from a *Ksock* returns an "entire" message string.

The actual "pointy" or "entire" message data is held in the *msg* value of the *Message* instance.

The *to_bytes()* method returns the data for an "entire" message. In certain circumstances (typically, on a 64-byte system) the actual length of data returned by *to_bytes()* may be slightly too long (due to extra padding at the end).

This method calculates the correct length of the equivalent "entire" message for this Message, without any such padding. If you want to write the data returned by *to_bytes()* into a *Ksock*, only use the number of bytes indicated by this method.

**wants_us_to_reply**()
Return `True` if we (*specifically* us) are should reply to this message.

## 3.2 Announcement

**class** kbus.**Announcement**(*name*, *data=None*, *to=None*, *from_=None*, *flags=None*, *id=None*)
Bases: kbus.messages.Message

A "plain" message, needing no reply

This is intended to be a convenient way of constructing a message that is just aimed at any listeners.

It's also a terminological convenience - all of the "message" things are clearly messages, so we need a special name for "plain" messages... There's an argument for just factory functions to create these things, but a class feels a little cleaner to me.

An Announcement can be created in a variety of ways. Perhaps most obviously:

```
>>> ann1 = Announcement('$.Fred', data='1234')
>>> ann1
Announcement('$.Fred', data='1234')
```

Since Announcement is a "plain" Message, we expect to be able to use the normal ways of instantiating a Message for an Announcement.

So, an Announcement can be constructed from another message directly:

```
>>> ann2 = Announcement.from_message(ann1)
>>> ann2 == ann1
True
```

```
>>> msg = Announcement.from_message(ann1)
>>> ann2a = Announcement.from_message(msg)
>>> ann2 == ann2a
True
```

Since it's an Announcement, there's no *in_reply_to* argument

```
>>> fail = Announcement('$.Fred', in_reply_to=None)
Traceback (most recent call last):
...
TypeError: __init__() got an unexpected keyword argument 'in_reply_to'
```

and the *in_reply_to* value in Message objects is ignored:

```
>>> msg = Message('$.Fred', data='1234', in_reply_to=MessageId(1, 2))
>>> ann = Announcement.from_message(msg)
>>> ann
Announcement('$.Fred', data='1234')
>>> print ann.in_reply_to
None
```

or from the *extract()* tuple - again, *reply_to* will be ignored:

```
>>> ann3 = Announcement.from_sequence(ann1.extract())
>>> ann3 == ann1
True
```

or from an equivalent list (and as above for *reply_to*):

```
>>> ann3 = Announcement.from_sequence(list(ann1.extract()))
>>> ann3 == ann1
True
```

Or one can use the same thing represented as a string:

```
>>> ann_as_string = ann1.to_bytes()
>>> ann4 = Announcement.from_bytes(ann_as_string)
>>> ann4 == ann1
True
```

For convenience, the parts of an Announcement may be retrieved as properties:

```
>>> print ann1.id
None
>>> ann1.name
'$.Fred'
>>> ann1.to
0L
>>> ann1.from_
0L
>>> print ann1.in_reply_to # always expected to be None
None
>>> ann1.flags
0L
>>> ann1.data
'1234'
```

Note that:

1. An Announcement message is such because it is not a message of another type. There is nothing else special about it.

**static `from_bytes`** (*arg*)
Construct an Announcement from bytes, as read by *`Ksock.read_data()`*.

For instance:

```
>>> msg1 = Message('$.Fred', '12345678')
>>> msg1
Message('$.Fred', data='12345678')
>>> msg2 = Announcement.from_bytes(msg1.to_bytes())
>>> msg2
Announcement('$.Fred', data='12345678')
```

**static `from_message`** (*msg*, *data=None*, *to=None*, *from_=None*, *flags=None*, *id=None*)
Construct an Announcement from another message.

The optional arguments allow changing the named fields in the new Announcement.

For instance:

```
>>> msg1 = Message('$.Fred', '12345678')
>>> msg1
Message('$.Fred', data='12345678')
>>> msg2 = Announcement.from_message(msg1, flags=1)
>>> msg2
Announcement('$.Fred', data='12345678', flags=0x00000001)
```

**static `from_sequence`** (*seq*, *data=None*, *to=None*, *from_=None*, *flags=None*, *id=None*)
Construct an Announcement from a sequence, as returned by *`extract()`*.

The optional arguments allow changing the named fields in the new Announcement.

For instance:

```
>>> msg1 = Message('$.Fred', '12345678')
>>> msg1
Message('$.Fred', data='12345678')
>>> msg2 = Announcement.from_sequence(msg1.extract(), flags=1)
>>> msg2
Announcement('$.Fred', data='12345678', flags=0x00000001)
```

**set_want_reply**(*value=True*)
Announcements are not Requests.

## 3.3 Request and stateful_request

**class** kbus.**Request**(*name*, *data=None*, *to=None*, *from_=None*, *final_to=None*, *flags=None*, *id=None*)
Bases: kbus.messages.Message

A message that wants a reply.

This is intended to be a convenient way of constructing a message that wants a reply.

It doesn't take an *in_reply_to* initialisation argument:

```
>>> fail = Request('$.Fred', in_reply_to=None)
Traceback (most recent call last):
...
TypeError: __init__() got an unexpected keyword argument 'in_reply_to'
```

And it automatically sets the WANT_A_REPLY flag, but otherwise it behaves just like a Message.

For instance, consider:

```
>>> msg = Message('$.Fred', data='1234', flags=Message.WANT_A_REPLY)
>>> msg
Message('$.Fred', data='1234', flags=0x00000001)
>>> req = Request('$.Fred', data='1234')
>>> req
Request('$.Fred', data='1234', flags=0x00000001)
>>> req == msg
True
```

If it is given a *to* argument, then it is a Stateful Request - it will be an error if it cannot be delivered to that particular Replier (for instance, if the Replier had unbound and someone else had bound as Replier for this message name).

```
>>> req = Request('$.Fred', data='1234', to=1234)
>>> req
Request('$.Fred', data='1234', to=1234L, flags=0x00000001)
```

A Stateful Request may also need to supply a *final_to* argument, if the original Replier is over a (Limpet) network. This should be taken from an earlier Reply from that Replier – see the convenience function stateful_request(). However, it can be done by hand:

```
>>> req = Request('$.Fred', data='1234', to=1234, final_to=OrigFrom(12, 23), flags=0x00000001)
>>> req
Request('$.Fred', data='1234', to=1234L, final_to=OrigFrom(12, 23), flags=0x00000001)
```

Note that:

1. A request message is a request just because it has the *Message.WANT_A_REPLY* flag set. There is nothing else special about it.
2. A stateful request message is then a request that has its *to* flag set.

**static from_bytes**(*arg*)
Construct a Request from bytes, as read by *Ksock.read_data()*.

For instance:

```
>>> msg1 = Message('$.Fred', '12345678')
>>> msg1
Message('$.Fred', data='12345678')
>>> msg2 = Request.from_bytes(msg1.to_bytes())
>>> msg2
Request('$.Fred', data='12345678', flags=0x00000001)
```

**static from_message**(*msg*, *data=None*, *to=None*, *from_=None*, *final_to=None*, *flags=None*, *id=None*)
Construct a Request from another message.

The optional arguments allow changing the named fields in the new Request.

For instance:

```
>>> msg1 = Message('$.Fred', '12345678')
>>> msg1
Message('$.Fred', data='12345678')
>>> msg2 = Request.from_message(msg1, flags=2)
>>> msg2
Request('$.Fred', data='12345678', flags=0x00000003)
```

**static from_sequence**(*seq*, *data=None*, *to=None*, *from_=None*, *final_to=None*, *flags=None*, *id=None*)
Construct a Request from a sequence, as returned by *extract()*.

The optional arguments allow changing the named fields in the new Request.

For instance:

```
>>> msg1 = Message('$.Fred', '12345678')
>>> msg1
Message('$.Fred', data='12345678')
>>> msg2 = Request.from_sequence(msg1.extract(), flags=2)
>>> msg2
Request('$.Fred', data='12345678', flags=0x00000003)
```

**set_want_reply**()
Calling this method is an error in this subclass, as by definition a *Request* always has the WANT_A_REPLY flag set.

kbus.**stateful_request**(*earlier_msg*, *name*, *data=None*, *from_=None*, *flags=None*, *id=None*)
Construct a stateful Request, based on an earlier *Reply* or stateful *Request*.

This is intended to be the normal way of constructing a stateful request.

*earlier_msg* is either:

1. an earlier Reply, whose *from_* field will be used as the new Request's *to* field, and whose *orig_from* field will be used as the new Request's *final_to* field.

   Remember, a Reply is a message whose *in_reply_to* field is set.

2.an earlier Stateful Request, whose *to* and *orig_from* fields will be copied to the new Request.

Remember, a Stateful Request is a message with the *Message.WANT_A_REPLY* flag set (a Request), and whose *to* field is set (which is to a specific Replier).

The rest of the arguments are the same as for *Request()*, except that the *to* and *orig_from* initialiser arguments are missing.

For instance, in the normal (single network) case:

```
>>> reply = Reply('$.Fred', to=27, from_=39, in_reply_to=MessageId(0, 132))
>>> reply
Reply('$.Fred', to=27L, from_=39L, in_reply_to=MessageId(0, 132))
>>> request = stateful_request(reply, '$.SomethingElse')
>>> request
Request('$.SomethingElse', to=39L, flags=0x00000001)
```

or, with a Reply that has come from far away:

```
>>> reply = Reply('$.Fred', to=27, from_=39, in_reply_to=MessageId(0, 132), orig_from=OrigFrom(1
>>> reply
Reply('$.Fred', to=27L, from_=39L, orig_from=OrigFrom(19, 23), in_reply_to=MessageId(0, 132))
>>> request = stateful_request(reply, '$.SomethingElse')
>>> request
Request('$.SomethingElse', to=39L, final_to=OrigFrom(19, 23), flags=0x00000001)
```

or, reusing our stateful Request:

```
>>> request = stateful_request(request, '$.Again', data='Aha!')
>>> request
Request('$.Again', data='Aha!', to=39L, final_to=OrigFrom(19, 23), flags=0x00000001)
```

## 3.4 Reply and reply_to

**class** kbus.**Reply**(*name*, *data=None*, *to=None*, *from_=None*, *orig_from=None*, *in_reply_to=None*, *flags=None*, *id=None*)

Bases: kbus.messages.Message

A reply message.

(Note that the constructor for this class does *not* flip fields (such as *id* and *in_reply_to*, or *from_* and *to*) when building the Reply - if you want that behaviour (and you probably do), use the *reply_to()* function.)

Thus Reply can be used as, for instance:

```
>>> direct = Reply('$.Fred', to=27, in_reply_to=MessageId(0, 132))
>>> direct
Reply('$.Fred', to=27L, in_reply_to=MessageId(0, 132))
>>> reply = Reply.from_message(direct)
>>> direct == reply
True
```

Since a Reply is a Message with its *in_reply_to* set, this *must* be provided:

```
>>> msg = Message('$.Fred', data='1234', from_=27, to=99, id=MessageId(0, 132), flags=Message.WA
>>> msg
Message('$.Fred', data='1234', to=99L, from_=27L, flags=0x00000001, id=MessageId(0, 132))
>>> reply = Reply.from_message(msg)
```

```
Traceback (most recent call last):
...
ValueError: A Reply must specify in_reply_to
```

```
>>> reply = Reply.from_message(msg, in_reply_to=MessageId(0, 5))
>>> reply
Reply('$.Fred', data='1234', to=99L, from_=27L, in_reply_to=MessageId(0, 5), flags=0x00000001, i
```

When Limpet networks are in use, it may be necessary to construct a Reply with its *orig_from* field set (this should only really be done by a Limpet itself, though):

```
>>> reply = Reply.from_message(msg, in_reply_to=MessageId(0, 5), orig_from=OrigFrom(23, 92))
>>> reply
Reply('$.Fred', data='1234', to=99L, from_=27L, orig_from=OrigFrom(23, 92), in_reply_to=MessageI
```

It's also possible to construct a Reply in most of the other ways a *Message* can be constructed. For instance:

```
>>> rep2 = Reply.from_bytes(direct.to_bytes())
>>> rep2 == direct
True
>>> rep4 = Reply.from_sequence(direct.extract())
>>> rep4 == direct
True
```

**static `from_bytes`**(*arg*)

Construct a Message from bytes, as read by *Ksock.read_data()*.

*in_reply_to* must be set in the message data.

For instance:

```
>>> msg1 = Message('$.Fred', '12345678', in_reply_to=MessageId(0,5))
>>> msg1
Message('$.Fred', data='12345678', in_reply_to=MessageId(0, 5))
>>> msg2 = Message.from_bytes(msg1.to_bytes())
>>> msg2
Message('$.Fred', data='12345678', in_reply_to=MessageId(0, 5))
```

**static `from_message`**(*msg*, *data=None*, *to=None*, *from_=None*, *orig_from=None*, *in_reply_to=None*, *flags=None*, *id=None*)

Construct a Message from another message.

All the values in the old message, except the name, may be changed by specifying new values in the argument list.

*in_reply_to* must be specified explicitly, if it is not present in the old/template message.

For instance:

```
>>> msg1 = Message('$.Fred', '12345678')
>>> msg1
Message('$.Fred', data='12345678')
>>> msg2 = Reply.from_message(msg1, flags=2, in_reply_to=MessageId(0,5))
>>> msg2
Reply('$.Fred', data='12345678', in_reply_to=MessageId(0, 5), flags=0x00000002)
```

**static `from_sequence`**(*seq*, *data=None*, *to=None*, *from_=None*, *orig_from=None*, *in_reply_to=None*, *flags=None*, *id=None*)

Construct a Message from a sequence, as returned by *extract()*.

All the values in the old message, except the name, may be changed by specifying new values in the argument list.

*in_reply_to* must be specified explicitly, if it is not present in the sequence.

For instance:

```
>>> msg1 = Message('$.Fred', '12345678')
>>> msg1
Message('$.Fred', data='12345678')
>>> msg2 = Reply.from_sequence(msg1.extract(), flags=2, in_reply_to=MessageId(0,5))
>>> msg2
Reply('$.Fred', data='12345678', in_reply_to=MessageId(0, 5), flags=0x00000002)
```

kbus.**reply_to**(*original*, *data=None*, *flags=0*)

Return a *Reply* to the given *Message*.

This is intended to be the normal way of constructing a reply message.

For instance:

```
>>> msg = Message('$.Fred', data='1234', from_=27, to=99, id=MessageId(0, 132), flags=Message.WA
>>> msg
Message('$.Fred', data='1234', to=99L, from_=27L, flags=0x00000003, id=MessageId(0, 132))
>>> reply = reply_to(msg)
>>> reply
Reply('$.Fred', to=27L, in_reply_to=MessageId(0, 132))
```

Note that:

1. The message we're constructing a reply to must be a message that wants a reply. Specifically, this means that it must have the *Message.WANT_A_REPLY* flag set, and also the *Message.WANT_YOU_TO_REPLY* flag. This last is because anyone listening to a Request will "see" the *Message.WANT_A_REPLY* flag, but only the (single) replier will receive the message with *Message.WANT_YOU_TO_REPLY* set.
2. A reply message is a reply because it has the *in_reply_to* field set. This indicates the message id of the original message, the one we're replying to.
3. As normal, the Reply's own message id is unset - KBUS will set this, as for any message.
4. We give a specific *to* value, the id of the *Ksock* that sent the original message, and thus the *from* value in the original message.
5. We keep the same message name, but don't copy the original message's data. If we want to send data in a reply message, it will be our own data.

The other arguments available are *flags* (allowing the setting of flags such as *Message.ALL_OR_WAIT*, for instance), and *data*, allowing reply data to be added:

```
>>> rep4 = reply_to(msg, flags=Message.ALL_OR_WAIT, data='1234')
>>> rep4
Reply('$.Fred', data='1234', to=27L, in_reply_to=MessageId(0, 132), flags=0x00000100)
```

## 3.5 MessageId

**class** kbus.**MessageId**

Bases: _ctypes.Structure

A wrapper around a message id.

```
>>> a = MessageId(1, 2)
>>> a
MessageId(1, 2)
>>> a < MessageId(2, 2) and a < MessageId(1, 3)
True
>>> a == MessageId(1, 2)
True
>>> a > MessageId(0, 2) and a > MessageId(1, 1)
True
```

We support addition in a limited manner:

```
>>> a + 3
MessageId(1, 5)
```

simply to make it convenient to generate unique message ids. This returns a new MessageId - it doesn't amend the existing one.

**network_id**
Structure/Union member

**serial_num**
Structure/Union member

## 3.6 Status

**class** kbus.**Status**(*original*)
Bases: kbus.messages.Message

A status message, from KBUS.

This is provided as a sugar-coating around the messages KBUS sends us. As such, it is not expected that a normal user would want to construct one, and the initialisation mechanisms are correspondingly more restrictive.

For instance:

```
>>> msg = Message('$.KBUS.Dummy', from_=27, to=99, in_reply_to=MessageId(0, 132))
>>> msg
Message('$.KBUS.Dummy', to=99L, from_=27L, in_reply_to=MessageId(0, 132))
>>> status = Status.from_bytes(msg.to_bytes())
>>> status
Status('$.KBUS.Dummy', to=99L, from_=27L, in_reply_to=MessageId(0, 132))
```

At the moment it is not possible to construct a Status message in any other way - it is assumed to be strictly for "wrapping" a message read (as bytes) from KBUS. Thus:

```
>>> msg = Status('$.Fred')
Traceback (most recent call last):
...
NotImplementedError: Use the Status.from_bytes() method to construct a Status
```

Note that:

1. A status message is such because it is a (sort of) *Reply*, with the message name starting with $.KBUS..

**static from_bytes**(*arg*)
Construct a Status from bytes, as read by *Ksock.read_data()*.

For instance:

```
>>> msg1 = Message('$.Fred', '12345678')
>>> msg1
Message('$.Fred', data='12345678')
>>> msg2 = Status.from_bytes(msg1.to_bytes())
>>> msg2
Status('$.Fred', data='12345678')
```

**static from_message**(*msg*, *data=None*, *to=None*, *from_=None*, *orig_from=None*, *final_to=None*, *in_reply_to=None*, *flags=None*, *id=None*)

It is not meaningful to create a Status from another Message.

```
>>> msg1 = Message('$.Fred', '12345678')
>>> msg1
Message('$.Fred', data='12345678')
>>> msg2 = Status.from_message(msg1, in_reply_to=MessageId(0,5))
Traceback (most recent call last):
...
NotImplementedError: Status does not support the from_message() static method
```

**static from_sequence**(*seq*, *data=None*, *to=None*, *from_=None*, *orig_from=None*, *final_to=None*, *in_reply_to=None*, *flags=None*, *id=None*)

It is not meaningful to create a Status from a sequence.

```
>>> msg1 = Message('$.Fred', '12345678')
>>> msg1
Message('$.Fred', data='12345678')
>>> msg2 = Status.from_sequence(msg1.extract())
Traceback (most recent call last):
...
NotImplementedError: Status does not support the from_sequence() static method
```

## 3.7 OrigFrom

**class** kbus.**OrigFrom**

Bases: _ctypes.Structure

A wrapper to the underlying C *struct kbus_orig_from* type. This is the type of *Message.orig_from* and *Message.final_to*.

```
>>> a = OrigFrom(1, 2)
>>> a
OrigFrom(1, 2)
>>> a < OrigFrom(2, 2) and a < OrigFrom(1, 3)
True
>>> a == OrigFrom(1, 2)
True
>>> a > OrigFrom(0, 2) and a > OrigFrom(1, 1)
True
```

**local_id**

Structure/Union member

**network_id**

Structure/Union member

## 3.8 split_replier_bind_event_data

kbus.**split_replier_bind_event_data**(*data*)

Split the data from a `$.KBUS.ReplierBindEvent` message.

Returns a tuple of the form (is_bind, binder, name)

CHAPTER 4

# KBUS Python bindings for Ksocks

KBUS lightweight message system.

## 4.1 Ksock

**class** kbus.**Ksock**(*which=0*, *mode='rw'*)

Bases: object

A wrapper around a KBUS device, for purposes of message sending.

*which* is which KBUS device to open – so if *which* is 3, we open /dev/kbus3.

*mode* should be 'r' or 'rw' – i.e., whether to open the device for read or write (opening for write also allows reading, of course).

Ksock can act like an iterable container; it implements `__iter__()` and *`next()`* in the usual way. It also provides `__enter__()` and `__exit__()` methods to support the use of `with`.

I'm not really very keen on the name Ksock, but it's better than the original "File", which I think was actively misleading.

**IOC_BIND = 1074293506**

**IOC_DISCARD = 27401**

**IOC_KSOCKID = 2148035332**

**IOC_LASTSENT = 2148035338**

**IOC_LENLEFT = 2148035335**

**IOC_MAGIC = 'k'**

**IOC_MAXMSGS = 3221777163**

**IOC_MAXMSGSIZE = 3221777170**

**IOC_MSGONLYONCE = 3221777166**

**IOC_NEWDEVICE = 2148035344**

**IOC_NEXTMSG = 2148035334**

**IOC_NUMMSGS = 2148035340**

**IOC_REPLIER = 3221777157**

**IOC_REPORTREPLIERBINDS = 3221777169**

**IOC_RESET = 27393**

**IOC_SEND = 2148035336**

**IOC_UNBIND = 1074293507**

**IOC_UNREPLIEDTO = 2148035341**

**IOC_VERBOSE = 3221777167**

**bind**(*name*, *replier=False*)
Bind the given name to the file descriptor.

If *replier*, then we are binding as the only fd that can reply to this message name.

**close**()
Shuts down the socket. This implicitly unbinds the Ksock client object from every name it was bound to. Any further attempt to use the object will cause errors.

**discard**()
Discard the message being written.

Indicates that we have should throw away the message we've been writing. Has no effect if there is no current message being written (for instance, because *send()* has already been called). be sent.

**fileno**()
Return the integer file descriptor from our internal fd.

This allows a Ksock instance to be used in a call of `select.select()` - so, for instance, one should be able to do:

```
(r, w, x) = select.select([ksock1, ksock2, ksock3], None, None)
```

instead of the (less friendly, but also valid):

```
(r, w, x) = select.select([ksock1.fd, ksock2.fd, ksock3.fd], None, None)
```

**find_replier**(*name*)
Find the id of the replier (if any) for this message.

Returns None if there was no replier, otherwise the replier's id.

**kernel_module_verbose**(*verbose=True*, *just_ask=False*)
Determine whether the kernel module should output verbose messages.

Determine whether the kernel module should output verbose messages for this device (this Ksock). This will only have any effect if the kernel module was built with `CONFIG_KBUS_DEBUG` defined.

The default is False, i.e., not to output verbose messages (as this clutters up the kernel log).

- if *verbose* is true then we want verbose messages.
- if *just_ask* is true, then we just want to find out the current state of the flag, and *verbose* will be ignored.

Returns the previous value of the flag (i.e., what it used to be set to). Which, if *just_ask* is true, will also be the current state.

Beware that setting this flag affects the Ksock as a whole, so it is possible for several programs to open a Ksock and "disagree" about how this flag should be set.

**ksock_id**()
Return the internal 'Ksock id' for this file descriptor.

**last_msg_id**()
Return the id of the last message written on this file descriptor.

Returns 0 before any messages have been sent.

**len_left**()
Return how many bytes of the current message are still to be read.

Returns 0 if there is no current message (i.e., *next_msg()* has not been called), or if there are no bytes left.

**max_message_size**()
Return the maximum message size that can be written to this KBUS device.

**max_messages**()
Return the number of messages that can be queued on this Ksock.

**new_device**()
Request that KBUS set up a new device (/dev/kbus<n>).

Note that it can take a little while for the hotplugging mechanisms to set the new device up for user access.

Returns the new device number (<n>).

**next**()
This provides iteration support. Each iteration gives a whole message as returned by *read_next_msg()*. We stop when there is no next message to read.

**next_msg**()
Indicates that we want to start reading the next message.

Returns the length of the next message, or 0 if there is no next message at the present time.

**num_messages**()
Return the number of messages that are queued on this Ksock.

**num_unreplied_to**()
Return the number of replies we still have outstanding.

That is, the number of Requests that we have read, which had the *Message.WANT_YOU_TO_REPLY* flag set, but for which we have not yet sent a *Reply*.

**read_data**(*count*)
Read the next *count* bytes, and return them.

Returns '' (the empty string) if there was nothing to be read, which is consistent with now Python file reads normally behave at end-of-file.

**read_msg**(*length*)
Read a Message of length *length* bytes.

It is assumed that *length* was returned by a previous call of *next_msg()*. It must be large enough to cause the entire message to be read.

After the data has been read, it is passed to Message to construct a message instance, which is returned.

Returns None if there was nothing to be read.

**read_next_msg**()
Read the next Message.

Equivalent to a call of *next_msg()*, followed by reading the appropriate number of bytes and passing that to Message to construct a message instance, which is returned.

Returns None if there was nothing to be read.

**report_replier_binds**(*report_events=True*, *just_ask=False*)
Determine whether the kernel module should report Replier bind/unbind events.

Determine whether the kernel module should output a "synthetic" message to announce each Replier bind/unbind event.

When the flag is set, then each time a Replier binds or unbinds to a message (i.e., when `ksock.bind(name,True)` or `ksock.unbind(name,True` is called), a message will automatically be generated and sent.

The message generated is called '$.KBUS.ReplierBindEvent', and it has data:

- a 32-bit value, 1 if this is a bind, 0 if it is an unbind
- a 32-bit value, the Ksock id of the binder
- the name of the message being bound to by a Replier (terminated by a null byte, and then, if necessary, padded up to the next four-byte boundary with null bytes

The default is False, i.e., not to output report such events.

- if *report_events* is true then we want bind/unbind messages.
- if *just_ask* is true, then we just want to find out the current state of the flag, and *report_events* will be ignored.

Returns the previous value of the flag (i.e., what it used to be set to). Which, if *just_ask* is true, will also be the current state.

Beware that setting this flag affects the Ksock as a whole, so it is possible for several programs to open a Ksock and "disagree" about how this flag should be set.

**send**()
Indicates that we have finished writing a message, and it should be sent.

Returns the *MessageId* of the send message.

Raises `IOError` with errno `ENOMSG` if no message has been written, i.e. there is nothing to send.

**send_msg**(*message*)
Write a Message, and then send it.

Entirely equivalent to calling *write_msg()* and then *send()*, and returns the *MessageId* of the sent message, as *send()* does.

**set_max_message_size**(*count*)
Set the maximum message size that can be written to this KBUS device.

A *count* of 0 does not actually change the value - this may thus be used to query the Ksock for the current value of the maximum.

A 'count' of 1 also does not change the value, instead, it returns the absolute maximum size that the value may be set to.

The method *max_message_size()* method is provided as a possibly simpler to use alternative for a call of 'count' 0.

Returns the maximum size of message that may be written to this KBUS device, or a query result as described above.

**set_max_messages**(*count*)
Set the number of messages that can be queued on this Ksock.

A *count* of 0 does not actually change the value - this may thus be used to query the Ksock for the current value of the maximum. However, the "more Pythonic" `max_messages()` method is provided for use when such a query is wanted, which is just syntactic sugar around such a call.

Returns the number of messages that are allowed to be queued on this Ksock.

**unbind**(*name*, *replier=False*)
Unbind the given name from the file descriptor.

The arguments need to match the binding that we want to unbind.

**wait_for_msg**(*timeout=None*)
Wait for the next Message.

This is a simple wrapper around `select.select()`, waiting for the next Message on this Ksock.

If timeout is given, it is a floating point number of seconds, after which to timeout the select, otherwise this method will wait forever.

Returns the new Message, or None if the timeout expired.

**want_messages_once**(*only_once=False*, *just_ask=False*)
Determine whether multiply-bound messages are only received once.

Determine whether we should receive a particular message once, even if we are both a Replier and Listener for the message, or if we are registered more than once as a Listener for the message name.

Note that in the case of a `Request` that we should reply to, we will always get the Request, and it will be the Listener's version of the message that will be "dropped".

The default is False, i.e., to receive each message as many times as we are bound to its name.

- if *only_once* is true then we want to receive each message once only.
- if *just_ask* is true, then we just want to find out the current state of the flag, and *only_once* will be ignored.

Returns the previous value of the flag (i.e., what it used to be set to). Which, if *just_ask* is true, will also be the current state.

Beware that setting this flag affects how messages are added to the Ksock's message queue *as soon as it is set* - so changing it and then changing it back "at once" is not (necessarily) a null operation.

**write_data**(*data*)
Write out (and flush) some data.

This does not actually send the message and does not imply that what has been written is all of a message (although clearly it should form *some* of a message).

**write_msg**(*message*)
Write a Message. Doesn't send it.

## 4.2 Ksock/C datastructures

### 4.2.1 Bind/unbind argument

**class** `kbus.`**BindStruct**
Bases: `_ctypes.Structure`

The datastucture we need to describe an `IOC_BIND` argument

**is_replier**
Structure/Union member

**len**
Structure/Union member

**name**
Structure/Union member

### 4.2.2 Result of find_replier

**class** kbus.**ReplierStruct**
Bases: _ctypes.Structure

The datastucture we need to describe an IOC_REPLIER argument

**len**
Structure/Union member

**name**
Structure/Union member

**return_id**
Structure/Union member

### 4.2.3 Result of send

**class** kbus.**SendResultStruct**
Bases: _ctypes.Structure

The datastucture we need to describe an IOC_SEND argument/return

**msg_id**
Structure/Union member

**retval**
Structure/Union member

## 4.3 Other functions

### 4.3.1 read_bindings

kbus.**read_bindings**(*names*)
Read the bindings from /proc/kbus/bindings, and return a list

/proc/kbus/bindings gives us data like:

```
0: 10 16319 R $.Fred
0: 11 17420 L $.Fred.Bob
0: 12 17422 R $.William
```

(i.e., device, file descriptor id, PID of process, whether it is Replier or Listener, and the message name concerned).

*names* is a dictionary of file descriptor binding id to string (name) - for instance:

{ 10:'f1', 11:'f2' }

If there is no entry in the *names* dictionary for a given id, then the id will be used (as an integer).

Thus with the above we would return a list of the form:

```
[ ('f1', True, '$.Fred'), ('f2', False, '$.Fred.Bob'),
  (12, True, '$.William' ]
```

CHAPTER 5

# KBUS C bindings

*Proper documentation of the C bindings for KBUS (libkbus) will live here. For the moment, here are some header files:*

- `kbus/linux/kbus_defns.h` - the kernel header file, which defines the various datastructures, the IOCTLs, and some useful macros and functions. This header file is used by the kernel, and also by user space programs (although in this latter case normally by including the `libkbus` header file).
- `libkbus/kbus.h` - the main header file for `libkbus`. This include the kernel header file for you.
- `libkbux/limpet.h` - the `libkbus` header file for Limpet code. Since Limpets are more normally used via the `runlimpet` application, this may or may not be of direct use.

## 5.1 kbus/linux/kbus_defns.h

*Please be aware that the comments in this file are currently inaccurate, and scheduled to be rewritten. The datastructures are, of course, correct...*

```
/* Kbus kernel module external headers
 *
 * This file provides the definitions (datastructures and ioctls) needed to
 * communicate with the KBUS character device driver.
 */

/*
 * ***** BEGIN LICENSE BLOCK *****
 * Version: MPL 1.1
 *
 * The contents of this file are subject to the Mozilla Public License Version
 * 1.1 (the ``License''); you may not use this file except in compliance with
 * the License. You may obtain a copy of the License at
 * http://www.mozilla.org/MPL/
 *
 * Software distributed under the License is distributed on an ``AS IS'' basis,
 * WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License
 * for the specific language governing rights and limitations under the
 * License.
 *
 * The Original Code is the KBUS Lightweight Linux-kernel mediated
 * message system
 *
```

```
 * The Initial Developer of the Original Code is Kynesim, Cambridge UK.
 * Portions created by the Initial Developer are Copyright (C) 2009
 * the Initial Developer. All Rights Reserved.
 *
 * Contributor(s):
 *   Kynesim, Cambridge UK
 *   Tony Ibbs <tibs@tonyibbs.co.uk>
 *
 * Alternatively, the contents of this file may be used under the terms of the
 * GNU Public License version 2 (the ``GPL''), in which case the provisions of
 * the GPL are applicable instead of the above.  If you wish to allow the use
 * of your version of this file only under the terms of the GPL and not to
 * allow others to use your version of this file under the MPL, indicate your
 * decision by deleting the provisions above and replace them with the notice
 * and other provisions required by the GPL.  If you do not delete the
 * provisions above, a recipient may use your version of this file under either
 * the MPL or the GPL.
 *
 * ***** END LICENSE BLOCK *****
 */

#ifndef _kbus_defns
#define _kbus_defns

#if !__KERNEL__ && defined(__cplusplus)
extern ``C'' {
#endif

#include <linux/types.h>
#if __KERNEL__
#include <linux/kernel.h>
#include <linux/ioctl.h>
#else
#include <stdint.h>
#include <sys/ioctl.h>
#endif

/*
 * A message id is made up of two fields.
 *
 * If the network id is 0, then it is up to us (KBUS) to assign the
 * serial number. In other words, this is a local message.
 *
 * If the network id is non-zero, then this message is presumed to
 * have originated from another ``network'', and we preserve both the
 * network id and the serial number.
 *
 * The message id {0,0} is special and reserved (for use by KBUS).
 */
struct kbus_msg_id {
        __u32 network_id;
        __u32 serial_num;
};
```

```
/*
 * kbus_orig_from is used for the ``originally from'' and ``finally to'' ids
 * in the message header. These in turn are used when messages are
 * being sent between KBUS systems (via KBUS ``Limpets''). KBUS the kernel
 * module transmits them, unaltered, but does not use them (although
 * debug messages may report them).
 *
 * An ``originally from'' or ``finally to'' id is made up of two fields, the
 * network id (which indicates the Limpet, if any, that originally gated the
 * message), and a local id, which is the Ksock id of the original sender
 * of the message, on its local KBUS.
 *
 * If the network id is 0, then the ``originally from'' id is not being used.
 *
 * Limpets and these fields are discussed in more detail in the userspace
 * KBUS documentation - see http://kbus-messaging.org/ for pointers to
 * more information.
 */
struct kbus_orig_from {
        __u32 network_id;
        __u32 local_id;
};

/* When the user asks to bind a message name to an interface, they use: */
struct kbus_bind_request {
        __u32 is_replier;       /* are we a replier? */
        __u32 name_len;
        char *name;
};

/* When the user requests the id of the replier to a message, they use: */
struct kbus_bind_query {
        __u32 return_id;
        __u32 name_len;
        char *name;
};

/* When the user writes/reads a message, they use: */
struct kbus_message_header {
        /*
         * The guards
         * ----------
         *
         * * `start_guard' is notionally ``Kbus'', and `end_guard' (the 32 bit
         *   word after the rest of the message datastructure) is notionally
         *   ``subK''. Obviously that depends on how one looks at the 32-bit
         *   word. Every message datastructure shall start with a start guard
         *   and end with an end guard.
         *
         * These provide some help in checking that a message is well formed,
         * and in particular the end guard helps to check for broken length
         * fields.
         *
         * - `id' identifies this particular message.
```

```
*
*   When a user writes a new message, they should set this to {0,0}.
*   KBUS will then set a new message id for the message.
*
*   When a user reads a message, this will have been set by KBUS.
*
*   When a user replies to a message, they should copy this value
*   into the `in_reply_to' field, so that the recipient will know
*   what message this was a reply to.
*
* - `in_reply_to' identifies the message this is a reply to.
*
*   This shall be set to {0,0} unless this message *is* a reply to a
*   previous message. In other words, if this value is non-0, then
*   the message *is* a reply.
*
* - `to' is who the message is to be sent to.
*
*   When a user writes a new message, this should normally be set
*   to {0,0}, meaning ``anyone listening'' (but see below if ``state''
*   is being maintained).
*
*   When replying to a message, it shall be set to the `from' value
*   of the orginal message.
*
*   When constructing a request message (a message wanting a reply),
*   the user can set it to a specific replier id, to produce a stateful
*   request. This is normally done by copying the `from' of a previous
*   Reply from the appropriate replier. When such a message is sent,
*   if the replier bound (at that time) does not have that specific
*   id, then the send will fail.
*
*   Note that if `to' is set, then `orig_from' should also be set.
*
* - `from' indicates who sent the message.
*
*   When a user is writing a new message, they should set this
*   to {0,0}.
*
*   When a user is reading a message, this will have been set
*   by KBUS.
*
*   When a user replies to a message, the reply should have its
*   `to' set to the original messages `from', and its `from' set
*   to {0,0} (see the ``hmm'' caveat under `to' above, though).
*
* - `orig_from' and `final_to' are used when Limpets are mediating
*   KBUS messages between KBUS devices (possibly on different
*   machines). See the description by the datastructure definition
*   above. The KBUS kernel preserves and propagates their values,
*   but does not alter or use them.
*
* - `extra' is currently unused, and KBUS will set it to zero.
*   Future versions of KBUS may treat it differently.
```

```
 *
 * - `flags' indicates the type of message.
 *
 *   When a user writes a message, this can be used to indicate
 *   that:
 *
 *   * the message is URGENT
 *   * a reply is wanted
 *
 *   When a user reads a message, this indicates if:
 *
 *   * the message is URGENT
 *   * a reply is wanted
 *
 *   When a user writes a reply, this field should be set to 0.
 *
 *   The top half of the `flags' is not touched by KBUS, and may
 *   be used for any purpose the user wishes.
 *
 * - `name_len' is the length of the message name in bytes.
 *
 *   This must be non-zero.
 *
 * - `data_len' is the length of the message data in bytes. It may be
 *   zero if there is no data.
 *
 * - `name' is a pointer to the message name. This should be null
 *   terminated, as is the normal case for C strings.
 *
 *   NB: If this is zero, then the name will be present, but after
 *   the end of this datastructure, and padded out to a multiple of
 *   four bytes (see kbus_entire_message). When doing this padding,
 *   remember to allow for the terminating null byte. If this field is
 *   zero, then `data' shall also be zero.
 *
 * - `data' is a pointer to the data. If there is no data (if
 *   `data_len' is zero), then this shall also be zero. The data is
 *   not touched by KBUS, and may include any values.
 *
 *   NB: If this is zero, then the data will occur immediately
 *   after the message name, padded out to a multiple of four bytes.
 *   See the note for `name' above.
 *
 */
__u32 start_guard;
struct kbus_msg_id id;  /* Unique to this message */
struct kbus_msg_id in_reply_to; /* Which message this is a reply to */
__u32 to;               /* 0 (empty) or a replier id */
__u32 from;             /* 0 (KBUS) or the sender's id */
struct kbus_orig_from orig_from;/* Cross-network linkage */
struct kbus_orig_from final_to; /* Cross-network linkage */
__u32 extra;    /* ignored field - future proofing */
__u32 flags;    /* Message type/flags */
__u32 name_len; /* Message name's length, in bytes */
```

```
        __u32 data_len; /* Message length, also in bytes */
        char *name;
        void *data;
        __u32 end_guard;
};

#define KBUS_MSG_START_GUARD    0x7375624B
#define KBUS_MSG_END_GUARD      0x4B627573

/*
 * When a message is returned by `read', it is actually returned using the
 * following datastructure, in which:
 *
 * - `header.name' will point to `rest[0]'
 * - `header.data' will point to `rest[(header.name_len+3)/4]'
 *
 * followed by the name (padded to 4 bytes, remembering to allow for the
 * terminating null byte), followed by the data (padded to 4 bytes) followed by
 * (another) end_guard.
 */
struct kbus_entire_message {
        struct kbus_message_header header;
        __u32 rest[];
};

/*
 * We limit a message name to at most 1000 characters (some limit seems
 * sensible, after all)
 */
#define KBUS_MAX_NAME_LEN       1000

/*
 * The length (in bytes) of the name after padding, allowing for a terminating
 * null byte.
 */
#define KBUS_PADDED_NAME_LEN(name_len)   (4 * ((name_len + 1 + 3) / 4))

/*
 * The length (in bytes) of the data after padding
 */
#define KBUS_PADDED_DATA_LEN(data_len)   (4 * ((data_len + 3) / 4))

/*
 * Given name_len (in bytes) and data_len (in bytes), return the
 * length of the appropriate kbus_entire_message_struct, in bytes
 *
 * Note that we're allowing for a zero byte after the end of the message name.
 *
 * Remember that ``sizeof'' doesn't count the `rest' field in our message
 * structure.
 */
#define KBUS_ENTIRE_MSG_LEN(name_len, data_len)    \
        (sizeof(struct kbus_entire_message) + \
         KBUS_PADDED_NAME_LEN(name_len) + \
```

```
         KBUS_PADDED_DATA_LEN(data_len) + 4)

/*
 * The message name starts at entire->rest[0].
 * The message data starts after the message name - given the message
 * name's length (in bytes), that is at index:
 */
#define KBUS_ENTIRE_MSG_DATA_INDEX(name_len)     ((name_len+1+3)/4)
/*
 * Given the message name length (in bytes) and the message data length (also
 * in bytes), the index of the entire message end guard is thus:
 */
#define KBUS_ENTIRE_MSG_END_GUARD_INDEX(name_len, data_len)  \
        ((name_len+1+3)/4 + (data_len+3)/4)

/*
 * Find a pointer to the message's name.
 *
 * It's either the given name pointer, or just after the header (if the pointer
 * is NULL)
 */
static inline char *kbus_msg_name_ptr(const struct kbus_message_header
                                      *hdr)
{
        if (hdr->name) {
                return hdr->name;
        } else {
                struct kbus_entire_message *entire;
                entire = (struct kbus_entire_message *)hdr;
                return (char *)&entire->rest[0];
        }
}

/*
 * Find a pointer to the message's data.
 *
 * It's either the given data pointer, or just after the name (if the pointer
 * is NULL)
 */
static inline void *kbus_msg_data_ptr(const struct kbus_message_header
                                      *hdr)
{
        if (hdr->data) {
                return hdr->data;
        } else {
                struct kbus_entire_message *entire;
                __u32 data_idx;

                entire = (struct kbus_entire_message *)hdr;
                data_idx = KBUS_ENTIRE_MSG_DATA_INDEX(hdr->name_len);
                return (void *)&entire->rest[data_idx];
        }
}
```

```
/*
 * Find a pointer to the message's (second/final) end guard.
 */
static inline __u32 *kbus_msg_end_ptr(struct kbus_entire_message
                                      *entire)
{
        __u32 end_guard_idx =
                KBUS_ENTIRE_MSG_END_GUARD_INDEX(entire->header.name_len,
                                                entire->header.data_len);
        return (__u32 *) &entire->rest[end_guard_idx];
}

/*
 * Things KBUS changes in a message
 * --------------------------------
 * In general, KBUS leaves the content of a message alone. However, it does
 * change:
 *
 * - the message id (if id.network_id is unset - it assigns a new serial
 *   number unique to this message)
 * - the from id (if from.network_id is unset - it sets the local_id to
 *   indicate the Ksock this message was sent from)
 * - the KBUS_BIT_WANT_YOU_TO_REPLY bit in the flags (set or cleared
 *   as appropriate)
 * - the SYNTHETIC bit, which KBUS will always unset in a user message
 */

/*
 * Flags for the message `flags' word
 * ----------------------------------
 * The KBUS_BIT_WANT_A_REPLY bit is set by the sender to indicate that a
 * reply is wanted. This makes the message into a request.
 *
 *     Note that setting the WANT_A_REPLY bit (i.e., a request) and
 *     setting `in_reply_to' (i.e., a reply) is bound to lead to
 *     confusion, and the results are undefined (i.e., don't do it).
 *
 * The KBUS_BIT_WANT_YOU_TO_REPLY bit is set by KBUS on a particular message
 * to indicate that the particular recipient is responsible for replying
 * to (this instance of the) message. Otherwise, KBUS clears it.
 *
 * The KBUS_BIT_SYNTHETIC bit is set by KBUS when it generates a synthetic
 * message (an exception, if you will), for instance when a replier has
 * gone away and therefore a reply will never be generated for a request
 * that has already been queued.
 *
 *     Note that KBUS does not check that a sender has not set this
 *     on a message, but doing so may lead to confusion.
 *
 * The KBUS_BIT_URGENT bit is set by the sender if this message is to be
 * treated as urgent - i.e., it should be added to the *front* of the
 * recipient's message queue, not the back.
 *
 * Send flags
```

```
 * ==========
 * There are two ``send'' flags, KBUS_BIT_ALL_OR_WAIT and KBUS_BIT_ALL_OR_FAIL.
 * Either one may be set, or both may be unset.
 *
 *    If both bits are set, the message will be rejected as invalid.
 *
 *    Both flags are ignored in reply messages (i.e., messages with the
 *    `in_reply_to' field set).
 *
 * If both are unset, then a send will behave in the default manner. That is,
 * the message will be added to a listener's queue if there is room but
 * otherwise the listener will (silently) not receive the message.
 *
 *     (Obviously, if the listener is a replier, and the message is a request,
 *     then a KBUS message will be synthesised in the normal manner when a
 *     request is lost.)
 *
 * If the KBUS_BIT_ALL_OR_WAIT bit is set, then a send should block until
 * all recipients can be sent the message. Specifically, before the message is
 * sent, all recipients must have room on their message queues for this
 * message, and if they do not, the send will block until there is room for the
 * message on all the queues.
 *
 * If the KBUS_BIT_ALL_OR_FAIL bit is set, then a send should fail if all
 * recipients cannot be sent the message. Specifically, before the message is
 * sent, all recipients must have room on their message queues for this
 * message, and if they do not, the send will fail.
 */

/*
 * When a $.KBUS.ReplierBindEvent message is constructed, we use the
 * following to encapsulate its data.
 *
 * This indicates whether it is a bind or unbind event, who is doing the
 * bind or unbind, and for what message name. The message name is padded
 * out to a multiple of four bytes, allowing for a terminating null byte,
 * but the name length is the length without said padding (so, in C terms,
 * strlen(name)).
 *
 * As for the message header data structure, the actual data ``goes off the end''
 * of the datastructure.
 */
struct kbus_replier_bind_event_data {
        __u32 is_bind;  /* 1=bind, 0=unbind */
        __u32 binder;   /* Ksock id of binder */
        __u32 name_len; /* Length of name */
        __u32 rest[];   /* Message name */
};

#if !__KERNEL__
#define BIT(num)                (((unsigned)1) << (num))
#endif

#define KBUS_BIT_WANT_A_REPLY           BIT(0)
```

```
#define KBUS_BIT_WANT_YOU_TO_REPLY      BIT(1)
#define KBUS_BIT_SYNTHETIC              BIT(2)
#define KBUS_BIT_URGENT                 BIT(3)

#define KBUS_BIT_ALL_OR_WAIT            BIT(8)
#define KBUS_BIT_ALL_OR_FAIL            BIT(9)

/*
 * Standard message names
 * ======================
 * KBUS itself has some predefined message names.
 *
 * Synthetic Replies with no data
 * ------------------------------
 * These are sent to the original Sender of a Request when KBUS knows that the
 * Replier is not going to Reply. In all cases, you can identify which message
 * they concern by looking at the ``in_reply_to'' field:
 *
 * * Replier.GoneAway - the Replier has gone away before reading the Request.
 * * Replier.Ignored - the Replier has gone away after reading a Request, but
 *   before replying to it.
 * * Replier.Unbound - the Replier has unbound (as Replier) from the message
 *   name, and is thus not going to reply to this Request in its unread message
 *   queue.
 * * Replier.Disappeared - the Replier has disappeared when an attempt is made
 *   to send a Request whilst polling (i.e., after EAGAIN was returned from an
 *   earlier attempt to send a message). This typically means that the Ksock
 *   bound as Replier closed.
 * * ErrorSending - an unexpected error occurred when trying to send a Request
 *   to its Replier whilst polling.
 *
 * Synthetic Announcements with no data
 * ------------------------------------
 * * UnbindEventsLost - sent (instead of a Replier Bind Event) when the unbind
 *   events ``set aside'' list has filled up, and thus unbind events have been
 *   lost.
 */
#define KBUS_MSG_NAME_REPLIER_GONEAWAY          ``$.KBUS.Replier.GoneAway''
#define KBUS_MSG_NAME_REPLIER_IGNORED           ``$.KBUS.Replier.Ignored''
#define KBUS_MSG_NAME_REPLIER_UNBOUND           ``$.KBUS.Replier.Unbound''
#define KBUS_MSG_NAME_REPLIER_DISAPPEARED       ``$.KBUS.Replier.Disappeared''
#define KBUS_MSG_NAME_ERROR_SENDING             ``$.KBUS.ErrorSending''
#define KBUS_MSG_NAME_UNBIND_EVENTS_LOST        ``$.KBUS.UnbindEventsLost''

/*
 * Replier Bind Event
 * ------------------
 * This is the only message name for which KBUS generates data -- see
 * kbus_replier_bind_event_data. It is also the only message name which KBUS
 * does not allow binding to as a Replier.
 *
 * This is the message that is sent when a Replier binds or unbinds to another
 * message name, if the KBUS_IOC_REPORTREPLIERBINDS ioctl has been used to
 * request such notification.
```

```
 */
#define KBUS_MSG_NAME_REPLIER_BIND_EVENT        ``$.KBUS.ReplierBindEvent''

#define KBUS_IOC_MAGIC  `k'     /* 0x6b - which seems fair enough for now */
/*
 * RESET: reserved for future use
 */
#define KBUS_IOC_RESET      _IO(KBUS_IOC_MAGIC,  1)
/*
 * BIND - bind a Ksock to a message name
 * arg: struct kbus_bind_request, indicating what to bind to
 * retval: 0 for success, negative for failure
 */
#define KBUS_IOC_BIND      _IOW(KBUS_IOC_MAGIC,  2, char *)
/*
 * UNBIND - unbind a Ksock from a message id
 * arg: struct kbus_bind_request, indicating what to unbind from
 * retval: 0 for success, negative for failure
 */
#define KBUS_IOC_UNBIND    _IOW(KBUS_IOC_MAGIC,  3, char *)
/*
 * KSOCKID - determine a Ksock's Ksock id
 *
 * The network_id for the current Ksock is, by definition, 0, so we don't need
 * to return it.
 *
 * arg (out): __u32, indicating this Ksock's local_id
 * retval: 0 for success, negative for failure
 */
#define KBUS_IOC_KSOCKID   _IOR(KBUS_IOC_MAGIC,  4, char *)
/*
 * REPLIER - determine the Ksock id of the replier for a message name
 * arg: struct kbus_bind_query
 *
 *    - on input, specify the message name to ask about.
 *    - on output, KBUS fills in the relevant Ksock id in the return_value,
 *      or 0 if there is no bound replier
 *
 * retval: 0 for success, negative for failure
 */
#define KBUS_IOC_REPLIER  _IOWR(KBUS_IOC_MAGIC,  5, char *)
/*
 * NEXTMSG - pop the next message from the read queue
 * arg (out): __u32, number of bytes in the next message, 0 if there is no
 *            next message
 * retval: 0 for success, negative for failure
 */
#define KBUS_IOC_NEXTMSG   _IOR(KBUS_IOC_MAGIC,  6, char *)
/*
 * LENLEFT - determine how many bytes are left to read of the current message
 * arg (out): __u32, number of bytes left, 0 if there is no current read
 *            message
 * retval: 1 if there was a message, 0 if there wasn't, negative for failure
 */
```

```
#define KBUS_IOC_LENLEFT   _IOR(KBUS_IOC_MAGIC,  7, char *)
/*
 * SEND - send the current message
 * arg (out): struct kbus_msg_id, the message id of the sent message
 * retval: 0 for success, negative for failure
 */
#define KBUS_IOC_SEND      _IOR(KBUS_IOC_MAGIC,  8, char *)
/*
 * DISCARD - discard the message currently being written (if any)
 * arg: none
 * retval: 0 for success, negative for failure
 */
#define KBUS_IOC_DISCARD    _IO(KBUS_IOC_MAGIC,  9)
/*
 * LASTSENT - determine the message id of the last message SENT
 * arg (out): struct kbus_msg_id, {0,0} if there was no last message
 * retval: 0 for success, negative for failure
 */
#define KBUS_IOC_LASTSENT  _IOR(KBUS_IOC_MAGIC, 10, char *)
/*
 * MAXMSGS - set the maximum number of messages on a Ksock read queue
 * arg (in): __u32, the requested length of the read queue, or 0 to just
 *           request how many there are
 * arg (out): __u32, the length of the read queue after this call has
 *            succeeded
 * retval: 0 for success, negative for failure
 */
#define KBUS_IOC_MAXMSGS  _IOWR(KBUS_IOC_MAGIC, 11, char *)
/*
 * NUMMSGS - determine how many messages are in the read queue for this Ksock
 * arg (out): __u32, the number of messages in the read queue.
 * retval: 0 for success, negative for failure
 */
#define KBUS_IOC_NUMMSGS   _IOR(KBUS_IOC_MAGIC, 12, char *)
/*
 * UNREPLIEDTO - determine the number of requests (marked ``WANT_YOU_TO_REPLY'')
 * which we still need to reply to.
 * arg(out): __u32, said number
 * retval: 0 for success, negative for failure
 */
#define KBUS_IOC_UNREPLIEDTO _IOR(KBUS_IOC_MAGIC, 13, char *)
/*
 * MSGONLYONCE - should we receive a message only once?
 *
 * This IOCTL tells a Ksock whether it should only receive a particular message
 * once, even if it is both a Replier and Listener for the message (in which
 * case it will always get the message as Replier, if appropriate), or if it is
 * registered as multiple Listeners for the message.
 *
 * arg(in): __u32, 1 to change to ``only once'', 0 to change to the default,
 * 0xFFFFFFFF to just return the current/previous state.
 * arg(out): __u32, the previous state.
 * retval: 0 for success, negative for failure (-EINVAL if arg in was not one
 * of the specified values)
```

```
 */
#define KBUS_IOC_MSGONLYONCE  _IOWR(KBUS_IOC_MAGIC, 14, char *)
/*
 * VERBOSE - should KBUS output verbose ``printk'' messages (for this device)?
 *
 * This IOCTL tells a Ksock whether it should output debugging messages. It is
 * only effective if the kernel module has been built with the VERBOSE_DEBUGGING
 * flag set.
 *
 * arg(in): __u32, 1 to change to ``verbose'', 0 to change to ``quiet'',
 * 0xFFFFFFFF to just return the current/previous state.
 * arg(out): __u32, the previous state.
 * retval: 0 for success, negative for failure (-EINVAL if arg in was not one
 * of the specified values)
 */
#define KBUS_IOC_VERBOSE  _IOWR(KBUS_IOC_MAGIC, 15, char *)

/*
 * NEWDEVICE - request another KBUS device (/dev/kbus<n>).
 *
 * The next device number (up to a maximum of 255) will be allocated.
 *
 * arg(out): __u32, the new device number (<n>)
 * retval: 0 for success, negative for failure
 */
#define KBUS_IOC_NEWDEVICE _IOR(KBUS_IOC_MAGIC, 16, char *)

/*
 * REPORTREPLIERBINDS - request synthetic messages announcing Replier
 * bind/unbind events.
 *
 * If this flag is set, then when someone binds or unbinds to a message name as
 * a Replier, KBUS will send out a synthetic Announcement of this fact.
 *
 * arg(in): __u32, 1 to change to ``report'', 0 to change to ``do not report'',
 * 0xFFFFFFFF to just return the current/previous state.
 * arg(out): __u32, the previous state.
 * retval: 0 for success, negative for failure (-EINVAL if arg in was not one
 * of the specified values)
 */
#define KBUS_IOC_REPORTREPLIERBINDS  _IOWR(KBUS_IOC_MAGIC, 17, char *)
/*
 * MAXMSGSIZE - set the maximum size of a KBUS message for this KBUS device.
 * This may not be set to less than 100, or more than
 * CONFIG_KBUS_ABS_MAX_MESSAGE_SIZE.
 * arg (in): __u32, the requested maximum message size, or 0 to just
 *           request what the current limit is, 1 to request the absolute
 *           maximum size.
 * arg (out): __u32, the maximum essage size after this call has
 *            succeeded
 * retval: 0 for success, negative for failure
 */
#define KBUS_IOC_MAXMSGSIZE _IOWR(KBUS_IOC_MAGIC, 18, char *)
```

```
/* If adding another IOCTL, remember to increment the next number! */
#define KBUS_IOC_MAXNR  18

#if !__KERNEL__ && defined(__cplusplus)
}
#endif

#endif /* _kbus_defns */
```

## 5.2 libkbus/kbus.h

```
/*
 * ***** BEGIN LICENSE BLOCK *****
 * Version: MPL 1.1
 *
 * The contents of this file are subject to the Mozilla Public License Version
 * 1.1 (the "License"); you may not use this file except in compliance with
 * the License. You may obtain a copy of the License at
 * http://www.mozilla.org/MPL/
 *
 * Software distributed under the License is distributed on an "AS IS" basis,
 * WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License
 * for the specific language governing rights and limitations under the
 * License.
 *
 * The Original Code is the KBUS Lightweight Linux-kernel mediated
 * message system
 *
 * The Initial Developer of the Original Code is Kynesim, Cambridge UK.
 * Portions created by the Initial Developer are Copyright (C) 2009
 * the Initial Developer. All Rights Reserved.
 *
 * Contributor(s):
 *   Kynesim, Cambridge UK
 *   Gareth Bailey <gb@kynesim.co.uk>
 *   Tony Ibbs <tibs@tonyibbs.co.uk>
 *
 * ***** END LICENSE BLOCK *****
 */

#ifndef _LKBUS_H_INCLUDED_
#define _LKBUS_H_INCLUDED_

#ifdef __cplusplus
extern "C" {
#endif

#include "linux/kbus_defns.h"

#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdint.h>
#include <errno.h>
#include <string.h>
```

```
#include <stdlib.h>
#include <assert.h>

// NOTE that the middle portion of this file is autogenerated from libkbus.c
// so that the function header comments and function prototypes may be
// automatically kept in-step. This allows me to treat the C file as the main
// specification of the functions it defines, and also to keep C header
// comments in the C file, which I find easier when keeping the comments
// correct as the code is edited.
//
// The Python script extract_hdrs.py is used to perform this autogeneration.
// It should transfer any C function marked as 'extern' and with a header
// comment (of the '/*...*...*/' form).

/*
 * In kernel modules (and thus in the kbus_defns.h header, which is used by the
 * KBUS kernel module) ``typedef`` is strongly discouraged. Therefore the KBUS
 * kernel module header does not provide a typedef for, well, anything.
 * However, in the outside C programming world, typedefs are often a good thing,
 * allowing simpler programming, so we provide some here.
 */

typedef struct kbus_msg_id              kbus_msg_id_t;
typedef struct kbus_orig_from           kbus_orig_from_t;
typedef struct kbus_bind_request        kbus_bind_request_t;
typedef struct kbus_bind_query          kbus_bind_query_t;

typedef struct kbus_message_header      kbus_message_t;

typedef struct kbus_entire_message      kbus_entire_message_t;

typedef struct kbus_replier_bind_event_data     kbus_replier_bind_event_data_t;

/** A Ksock is just a file descriptor, an integer, as returned by 'open'.
 */
typedef int kbus_ksock_t;

/*
 * Please, however, do consult the kbus_defns.h header file for many useful
 * definitions, and also some key functions, such as:
 *
 * * kbus_msg_name_ptr(msg)
 * * kbus_msg_data_ptr(msg)
 *
 * which are really what you want for extracting KBUS message name and data
 * from the message datastructures (regardless of whether they are pointy or
 * not).
 *
 * If you haven't read kbus_defns.h, you *are* missing important information.
 */

// The following are used in kbus_wait_for_message.
#define KBUS_KSOCK_READABLE 1
#define KBUS_KSOCK_WRITABLE 2

/* Ksock Functions */

/** @file
```

```
 *
 * Note that all of the functions here are non-blocking: there is no such
 * thing as a synchronous kbus socket (though there are wait() functions here
 * to emulate one).
 */

// -------- TEXT AFTER THIS AUTOGENERATED - DO NOT EDIT --------
// Autogenerated by extract_hdrs.py on 2013-01-16 (Wed 16 Jan 2013) at 11:47

/*
 * Open a Ksock.
 *
 * `device_number` indicates which KSock device to open, as
 * "/dev/kbus<device_number>".
 *
 * Which device numbers are available depends upon how many KBUS devices have
 * been initialised, either when the KBUS kernel module was installed, or by
 * use of `kbus_new_device()`.
 *
 * `flags` may be one of ``O_RDONLY``, ``O_WRONLY`` or ``O_RDWR``.
 *
 * Returns the file descriptor for the new Ksock, or a negative value on error.
 * The negative value will be ``-errno``.
 */
extern kbus_ksock_t kbus_ksock_open(uint32_t device_number,
                                    int      flags);

/*
 * Open a Ksock by device name. Since KBUS currrently only supports devices
 * of the for ``/dev/kbus<device_number>``, this function has no advantage
 * over `kbus_ksock_open``.
 *
 * `device_name` indicates which KSock device to open, as "/dev/kbus<device_number>",
 * where ``<device_number>`` is zero or more, depending on how many KBUS
 * devices are initialised.
 *
 * `flags` may be one of ``O_RDONLY``, ``O_WRONLY`` or ``O_RDWR``.
 *
 * Returns the file descriptor for the new Ksock, or a negative value on error.
 * The negative value will be ``-errno``.
 */
extern kbus_ksock_t kbus_ksock_open_by_name(const char *device_name,
                                            int         flags);

/*
 * Close a Ksock.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_ksock_close(kbus_ksock_t ksock);

/*
 * Bind the given message name to the specified Ksock.
 *
 * If `is_replier`, then bind as a Replier, otherwise as a Listener.
 *
 * Only one KSock at a time may be bound to a particular message as a Replier.
 *
```

```
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_ksock_bind(kbus_ksock_t         ksock,
                           const char          *name,
                           uint32_t             is_replier);

/*
 * Unbind the given message name to the specified Ksock.
 *
 * If `is_replier`, then unbind as a Replier, otherwise as a Listener.
 *
 * The unbinding must exactly match a previous binding (i.e., both message name
 * and `is_replier` must match).
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_ksock_unbind(kbus_ksock_t         ksock,
                             const char          *name,
                             uint32_t             is_replier);

/*
 * Return the internal (to KBUS) Ksock id for this Ksock.
 *
 * The Ksock id is a positive, non-zero number. It is used in message ``to``
 * and ``from`` fields.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_ksock_id(kbus_ksock_t   ksock,
                         uint32_t      *ksock_id);

/*
 * Indicate that we wish to start reading the next message.
 *
 * Each Ksock has an (internal to KBUS) "next message" list. This function
 * pops the next message from that list, and makes it the "being read" message.
 * If there was still data for an earlier "being read" message, this will be
 * thrown away.
 *
 * `message_length` is set to the length of the message - that is, the value
 * to be passed to a subsequent call of ``kbus_ksock_next_msg()`` - or 0 if
 * there is no next message.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_ksock_next_msg(kbus_ksock_t     ksock,
                               uint32_t        *message_length);

/*
 * Find out how many bytes of the "being read" message are still to be read.
 *
 * `len_left` is set to the remaining number of bytes, or 0 if there are no
 * more bytes in the "being read" message, or if there is no "being read"
 * message (i.e., ``kbus_ksock_next_msg()`` has not been called since the
 * last message was finished or discarded).
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
```

```
extern int kbus_ksock_len_left(kbus_ksock_t   ksock,
                               uint32_t      *len_left);

/*
 * Determine the message id of the last message written on this Ksock.
 *
 * This will be {0,0} if there was no previous message.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_ksock_last_msg_id(kbus_ksock_t          ksock,
                                  kbus_msg_id_t        *msg_id);

/*
 * Find the Ksock id of the Replier for the given message name.
 *
 * `replier_ksock_id` will either be the Replier's Ksock id, or 0 if there
 * is no Replier bound for this message name.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_ksock_find_replier(kbus_ksock_t   ksock,
                                   const char    *name,
                                   uint32_t      *replier_ksock_id);

/*
 * Determine the number of (unread) messages that can be queued for this Ksock.
 *
 * If 'max_messages' is greater than 0, then KBUS will be asked to adjust the
 * read queue for this Ksock to length 'max_messages'.
 *
 * If 'max_messages' is 0, then it will be set to the current length of the
 * queue.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_ksock_max_messages(kbus_ksock_t   ksock,
                                   uint32_t      *max_messages);

/*
 * Find out how many (unread) messages are in the read queue for this Ksock.
 *
 * 'num_messages' will be set to the number of messages in the read queue.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_ksock_num_messages(kbus_ksock_t   ksock,
                                   uint32_t      *num_messages);

/*
 * Determine the number of (unread) messages queued for this Ksock.
 *
 * Returns the current (unread) message count for this Ksock, or a negative
 * number (``-errno``) for failure.
 */
extern int kbus_ksock_num_unreplied_to(kbus_ksock_t   ksock,
                                       uint32_t      *num_messages);
```

```
/*
 * Send the last written message.
 *
 * Used to send a message when all of it has been written.
 *
 * Once the messge has been sent, the message and any name/data pointed to may
 * be freed.
 *
 * `msg_id` returns the message id assigned to the message by KBUS.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_ksock_send(kbus_ksock_t         ksock,
                           kbus_msg_id_t       *msg_id);

/*
 * Discard the message being written.
 *
 * Indicates that KBUS should throw away the (partial) message that has been
 * written. If there is no current message being written (for instance, because
 * ``kbus_ksock_send()`` has just been called), then this function has no
 * effect.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_ksock_discard(kbus_ksock_t         ksock);

/*
 * Determine whether multiply-bound messages are only received once.
 *
 * Determine whether this Ksock should receive a particular message once, even
 * if it is both a Replier and Listener for the message, or if it is registered
 * more than once as a Listener for the message name.
 *
 * Note that in the case of a Request that the Ksock should reply to, it will
 * always get the Request, and it will be the Listener's version of the message
 * that will be "dropped".
 *
 * If `request` is 1, then only one copy of the message is wanted.
 *
 * If `request` is 0, then as many copies as implied by the bindings are wanted.
 *
 * If `request` is 0xFFFFFFFF, then the number of copies is not to be changed.
 * This may be used to query the current state of the "only once" flag for this
 * Ksock.
 *
 * Beware that setting this flag affects how messages are added to the Ksock's
 * message queue *as soon as it is set* - so changing it and then changing it
 * back "at once" is not (necessarily) a null operation.
 *
 * Returns 0 or 1, according to the state of the "only once" flag *before* this
 * function was called, or a negative number (``-errno``) for failure.
 */
extern int kbus_ksock_only_once(kbus_ksock_t   ksock,
                                uint32_t       request);

/*
 * Determine whether Replier bind/unbind events should be reported.
```

```
 *
 * If `request` is 1, then each time a Ksock binds or unbinds as a Replier,
 * a Replier bind/unbind event should be sent (a "$.KBUS.ReplierBindEvent"
 * message).
 *
 * If `request` is 0, then Replier bind/unbind events should not be sent.
 *
 * If `request` is 0xFFFFFFFF, then the current state should not be changed.
 * This may be used to query the current state of the "send Replier bind event"
 * flag.
 *
 * Note that although this call is made via an individual Ksock, it affects the
 * behaviour of the entire KBUS device to which this Ksock is attached.
 *
 * Returns 0 or 1, according to the state of the "send Replier bind event" flag
 * *before* this function was called, or a negative number (``-errno``) for
 * failure.
 */
extern int kbus_ksock_report_replier_binds(kbus_ksock_t       ksock,
                                           uint32_t           request);

/*
 * Request verbose kernel module messages.
 *
 * KBUS writes message via the normal kernel module mechanisms (which may be
 * inspected, for instance, via the ``dmesg`` command). Normal output is meant
 * to be reasonably minimal. Verbose messages can be useful for debugging the
 * kernel module.
 *
 * If `request` is 1, then verbose kernel messages are wanted.
 *
 * If `request` is 0, then verbose kernel messages are not wanted.
 *
 * If `request` is 0xFFFFFFFF, then the current state should not be changed.
 * This may be used to query the current state of the "verbose" flag.
 *
 * Note that although this call is made via an individual Ksock, it affects the
 * behaviour of the entire KBUS kernel module.
 *
 * Returns 0 or 1, according to the state of the "verbose" flag *before* this
 * function was called, or a negative number (``-errno``) for failure.
 */
extern int kbus_ksock_kernel_module_verbose(kbus_ksock_t       ksock,
                                            uint32_t           request);

/*
 * Request the KBUS kernel module to create a new device (``/dev/kbus<n>``).
 *
 * `device_number` is the ``<n>`` for the new device.
 *
 * Note that it takes the kernel's hotplugging mechanisms a little while to
 * notice/activate the device, so do not expect it to be available immediately
 * on return.
 *
 * Note that although this call is made via an individual Ksock, it affects the
 * behaviour of the entire KBUS kernel module.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
```

```
 */
extern int kbus_ksock_new_device(kbus_ksock_t  ksock,
                                 uint32_t     *device_number);

/*
 * Wait until either the Ksock may be read from or written to.
 *
 * Returns when there is data to be read from the Ksock, or the Ksock
 * may be written to.
 *
 * `wait_for` indicates what to wait for. It should be set to
 * ``KBUS_SOCK_READABLE``, ``KBUS_SOCK_WRITABLE``, or the two "or"ed together,
 * as appropriate.
 *
 * This is a convenience routine for when polling indefinitely on a Ksock is
 * appropriate. It is not intended as a generic routine for any more
 * complicated situation, when specific "poll" (or "select") code should be
 * written.
 *
 * Returns ``KBUS_SOCK_READABLE``, ``KBUS_SOCK_WRITABLE``, or the two "or"ed
 * together to indicate which operation is ready, or a negative number
 * (``-errno``) for failure.
 */
extern int kbus_wait_for_message(kbus_ksock_t  ksock,
                                 int           wait_for);

/*
 * Read a message of length `msg_len` bytes from this Ksock.
 *
 * It is assumed that `msg_len` was returned by a previous call of
 * ``kbus_ksock_next_msg()``. It must be large enough to cause the entire
 * message to be read.
 *
 * `msg` is the message read. This will be an "entire" message, and should be
 * freed by the caller when no longer needed.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 * Specifically, -EBADMSG will be returned if the underlying ``read``
 * returned 0.
 */
extern int kbus_ksock_read_msg(kbus_ksock_t      ksock,
                               kbus_message_t  **msg,
                               size_t            msg_len);

/*
 * Read the next message from this Ksock.
 *
 * This is equivalent to a call of ``kbus_ksock_next_msg()`` followed by a call
 * of ``kbus_ksock_read_msg()``.
 *
 * If there is no next message, ``msg`` will be NULL.
 *
 * If there is a next message, then ``msg`` will be the message read. This will
 * be an "entire" message, and should be freed by the caller when no longer
 * needed.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
```

```
extern int kbus_ksock_read_next_msg(kbus_ksock_t          ksock,
                                    kbus_message_t      **msg);

/*
 * Write the given message to this Ksock. Does not send it.
 *
 * The `msg` may be an "entire" or "pointy" message.
 *
 * If the `msg` is a "pointy" message, then the name and any data must not be
 * freed until the message has been sent (as the pointers are only "followed"
 * when the message is sent).
 *
 * It is normally easier to use ``kbus_ksock_send_msg()``.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_ksock_write_msg(kbus_ksock_t             ksock,
                                const kbus_message_t    *msg);

/*
 * Write data to the Ksock. Does not send.
 *
 * This may be used to write message data in parts. It is normally better to use
 * the "whole message" routines.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_ksock_write_data(kbus_ksock_t    ksock,
                                 uint8_t        *data,
                                 size_t          data_len);

/*
 * Write and send a message on the given Ksock.
 *
 * This combines the "write" and "send" functions into one call, and is the
 * normal way to send a message.
 *
 * The `msg` may be an "entire" or "pointy" message.
 *
 * Once the message has been sent, the message and any name/data pointed to may
 * be freed.
 *
 * `msg_id` returns the message id assigned to the message by KBUS.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_ksock_send_msg(kbus_ksock_t             ksock,
                               const kbus_message_t    *msg,
                               kbus_msg_id_t           *msg_id);

/*
 * Create a message (specifically, a "pointy" message).
 *
 * Note that the message name and data are not copied, and thus should not be
 * freed until the message has been sent (with ``kbus_ksock_send_msg()``).
 *
 * `msg` is the new message, as created by this function.
 *
```

```
 * `name` is the name for the message, and `name_len` the length of the name
 * (the number of characters in the name). A message name is required.
 *
 * 'data' is the data for this message, or NULL if there is no data. `data_len`
 * is then the length of the data, in bytes.
 *
 * `flags` may be any KBUS message flags required. Most messages with flags set
 * can more easily be created by one of the other message creation routines.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_msg_create(kbus_message_t **msg,
                           const char *name,
                           uint32_t name_len, /* bytes  */
                           const void *data,
                           uint32_t data_len, /* bytes */
                           uint32_t flags);

/*
 * Create an "entire" message.
 *
 * Copies are taken of both `name` and `data` (and placed at the end of the
 * message datastructure).
 *
 * Unless you need to be able to free the name and/or data before sending
 * the message, it is more usual to use ``kbus_msg_create()`` instead.
 *
 * `msg` is the new message, as created by this function.
 *
 * `name` is the name for the message, and `name_len` the length of the name
 * (the number of characters in the name). A message name is required. The
 * name will be copied when the message is created.
 *
 * 'data' is the data for this message, or NULL if there is no data. `data_len`
 * is then the length of the data, in bytes. The data will be copied when the
 * message is created.
 *
 * `flags` may be any KBUS message flags required. Most messages with flags set
 * can more easily be created by one of the other message creation routines.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_msg_create_entire(kbus_message_t        **msg,
                                  const char             *name,
                                  uint32_t               name_len, /* bytes  */
                                  const void             *data,
                                  uint32_t               data_len, /* bytes */
                                  uint32_t               flags);

/*
 * Create a Request (specifically, a "pointy" Request message).
 *
 * Note that the message name and data are not copied, and thus should not be
 * freed until the message has been sent (with ``kbus_ksock_send_msg()``).
 *
 * `msg` is the new message, as created by this function.
 *
 * `name` is the name for the message, and `name_len` the length of the name
```

```
 * (the number of characters in the name). A message name is required.
 *
 * 'data' is the data for this message, or NULL if there is no data. `data_len`
 * is then the length of the data, in bytes.
 *
 * `flags` may be any KBUS message flags required. These will be set on the
 * message, and then (after that) the KBUS_BIT_WANT_A_REPLY flag will be set
 * to make the new message a Request.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_msg_create_request(kbus_message_t **msg,
                                   const char *name,
                                   uint32_t name_len, /* bytes  */
                                   const void *data,
                                   uint32_t data_len, /* bytes */
                                   uint32_t flags);

/*
 * Create an "entire" Request message.
 *
 * This is identical in behaviour to ``kbus_msg_create_request()``, except
 * that an "entire" message is created, and thus both the message name and data
 * are copied. This means that the original `name` and `data` may be freed as
 * soon as the `msg` has been created.
 *
 * Unless you need to be able to free the name and/or data before sending
 * the message, it is more usual to use ``kbus_msg_create_request()`` instead.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_msg_create_entire_request(kbus_message_t        **msg,
                                          const char             *name,
                                          uint32_t                name_len, /* bytes  */
                                          const void             *data,
                                          uint32_t                data_len, /* bytes */
                                          uint32_t                flags);

/*
 * Create a Reply message, based on a previous Request.
 *
 * This is a convenience mechanism for creating the Reply to a previous
 * Request.
 *
 * The Request must have been marked as wanting this particular recipient to
 * reply to it (i.e., ``kbus_msg_wants_us_to_reply()`` returns true). If this
 * is not so, -EBADMSG will be returned.
 *
 * `msg` is the new Reply message. `in_reply_to` is the Request message for
 * which a Reply is wanted.
 *
 * The message name for the new message will be taken from the old message.
 *
 * The 'to' field for the new message will be set to the 'from' field in the old.
 *
 * The 'in_reply_to' field for the new message will be set to the message id of the old.
 *
 * 'data' is the data for the new message, or NULL if there is none. 'data_len'
```

```
 * is the length of the data, in bytes.
 *
 * As normal, the message name and data should not be freed until `msg` has
 * been sent. In the normal case, where `in_reply_to` is an "entire" message
 * read from KBUS, this means that `in_reply_to` and `data` should not be
 * freed, since the message name "inside" `in_reply_to` is being referenced.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_msg_create_reply_to(kbus_message_t **msg,
                                    const kbus_message_t *in_reply_to,
                                    const void *data,
                                    uint32_t data_len, /* bytes */
                                    uint32_t flags);

/*
 * Create an "entire" Reply message, based on a previous Request.
 *
 * This is identical in behaviour to ``kbus_msg_create_reply_to()``, except
 * that an "entire" message is created, and thus both the message name and data
 * are copied. This means that the original (`in_reply_to`) message and the
 * `data` may be freed as soon as the `msg` has been created.
 *
 * Unless you need to be able to free the original message and/or data before
 * sending * the message, it is more usual to use
 * ``kbus_msg_create_reply_to()`` instead.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_msg_create_entire_reply_to(kbus_message_t          **msg,
                                           const kbus_message_t     *in_reply_to,
                                           const void               *data,
                                           uint32_t                  data_len, /* bytes */
                                           uint32_t                  flags);

/*
 * Create a Stateful Request message, based on a previous Reply or Request.
 *
 * This is a convenience mechanism for creating a Stateful Request message
 * (a Request which must be delivered to a particular Ksock).
 *
 * `msg` is the new Stateful Request message.
 *
 * `earlier_msg` is either a Reply message from the desired Ksock, or a
 * previous Stateful Request to the same Ksock.
 *
 * If the earlier message is a Reply, then the 'to' and 'final_to' fields for
 * the new message will be set to the 'from' and 'orig_from' fields in the old.
 *
 * If the earlier message is a Stateful Request, then the 'to' and 'final_to'
 * fields for the new message will be copied from the old.
 *
 * If the earlier message is neither a Reply nor a Stateful Request, then
 * -EBADMSG will be returned.
 *
 * 'name' is the name for the new message, and 'name_len' is the length of that
 * name.
 *
```

```
 * 'data' is the data for the new message, or NULL if there is none. 'data_len'
 * is the length of the data, in bytes.
 *
 * 'flags' is any KBUS flags to set on the message (flags will not be copied
 * from the earlier message).
 *
 * As normal, the message name and data should not be freed until `msg` has
 * been sent. `earlier_msg` may be freed after this call has completed, as
 * any necessary data will have been copied from it.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_msg_create_stateful_request(kbus_message_t         **msg,
                                            const kbus_message_t    *earlier_msg,
                                            const char          *name,
                                            uint32_t             name_len,
                                            const void          *data,
                                            uint32_t             data_len, /* bytes */
                                            uint32_t             flags);

/*
 * Create an "entire" Stateful Request message, based on a previous Reply or
 * Request.
 *
 * This is identical in behaviour to ``kbus_msg_create_stateful_request()``,
 * except that an "entire" message is created, and thus both the message name
 * and data are copied. This means that both the `name` and the `data` may be
 * freed as soon as the `msg` has been created.
 *
 * Unless you need to be able to free the name and/or data before sending
 * the message, it is more usual to use ``kbus_msg_create_statefule_request()``
 * instead.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_msg_create_entire_stateful_request(kbus_message_t       **msg,
                                                   const kbus_message_t  *earlier_msg,
                                                   const char            *name,
                                                   uint32_t               name_len,
                                                   const void            *data,
                                                   uint32_t               data_len, /* bytes */
                                                   uint32_t               flags);

/*
 * Delete a message datastructure.
 *
 * Does nothing if `msg_p` is NULL, or `*msg_p` is NULL.
 *
 * Frees the message datastructure, but does not free any name or data that is
 * pointed to.
 */
extern void kbus_msg_delete(kbus_message_t **msg_p);

/*
 * Delete a message datastructure, and any name/data it points to.
 *
 * Does nothing if `msg_p` is NULL, or `*msg_p` is NULL.
 *
```

```
 * Frees the message datastructure. If the message is "pointy", also
 * frees the message name and any message data.
 *
 * Caveat: do not pass this an "entire" message which has had its "name"
 * and/or "data" pointers set to point "inside" itself, to the name and
 * data at the end of the "entire" message. Really, it will end very badly
 * (and you probably shouldn't have done that, anyway).
 */
extern void kbus_msg_delete_all(kbus_message_t **msg_p);

/*
 * Determine the size of a KBUS message.
 *
 * For a "pointy" message, returns the size of the message header.
 *
 * For an "entire" message, returns the size of the entire message.
 *
 * In either case, this is the length of data that would (for instance)
 * be written to a Ksock to actually write the message. In other words::
 *
 *   int len, rv;
 *   len = kbus_msg_sizeof(&msg);
 *   rv = kbus_ksock_write_data(ksock, &msg, len);
 *   if (rv < 0) return rv;
 *
 * is the "low level" equivalent of::
 *
 *   int rv = kbus_ksock_write_msg(ksock, &msg);
 *   if (rv < 0) return rv;
 *
 * Returns the length of 'msg', as described above.
 */
extern int kbus_msg_sizeof(const kbus_message_t *msg);

/*
 * A convenience routine to split the data of a Replier bind event.
 *
 * Replier bind events contain the following information:
 *
 * * `is_replier` is true if the event was a "bind", false it if was an
 *   "unbind".
 * * `binder` is the Ksock id of the binder.
 * * `name` is the name of the message that was being (un)bound.
 *
 * Note that `name` is a copy of the name (from the original `msg`), so that
 * the user may free the original message immediately. Clearly this copy will
 * also need freeing when finished with.
 *
 * Returns 0 for success, or a negative number (``-errno``) for failure.
 */
extern int kbus_msg_split_bind_event(const kbus_message_t  *msg,
                                     uint32_t              *is_bind,
                                     uint32_t              *binder,
                                     char                 **name);

/*
 * Print out a representation of a message.
 *
```

```
 * `stream` is the output stream to print to -- typically stdout.
 *
 * Does not print a newline.
 */
extern void kbus_msg_print(FILE                 *stream,
                           const kbus_message_t *msg);


#define KBUS_MSG_PRINT_FLAGS_ABBREVIATE (1<<0)

/*
 * Print out a representation of a message.
 *
 * `stream` is the output stream to print to -- typically stdout.
 *
 * Does not print a newline.
 *
 * flags is an OR of KBUS_MSG_PRINT_FLAGS_XXX.
 */
extern void kbus_msg_print2(FILE                 *stream,
                            const kbus_message_t *msg,
                            unsigned int flags);

/*
 * Print out (on stdout) information about a message.
 *
 * If `dump_data` is true, also print out the message data (in several forms).
 */
extern void kbus_msg_dump(const kbus_message_t *msg,
                          int                   dump_data);
// -------- TEXT BEFORE THIS AUTOGENERATED - DO NOT EDIT --------

/*
 * Check if a message is "entire".
 *
 * Returns true if the message is "entire", false if it is "pointy".
 * Strongly assumes the message is well-structured.
 */
static inline int kbus_msg_is_entire(const kbus_message_t     *msg)
{
  return msg->name == NULL;
}

/*
 * Check if a message is a Reply.
 */
static inline int kbus_msg_is_reply(const kbus_message_t    *msg)
{
  return msg->in_reply_to.network_id != 0 ||
         msg->in_reply_to.serial_num != 0;
}

/*
 * Check if a message is a Request.
 */
static inline int kbus_msg_is_request(const kbus_message_t      *msg)
{
  return (msg->flags & KBUS_BIT_WANT_A_REPLY) != 0;
```

```
}

/*
 * Check if a message is a Stateful Request.
 */
static inline int kbus_msg_is_stateful_request(const kbus_message_t      *msg)
{
  return (msg->flags & KBUS_BIT_WANT_A_REPLY) && (msg->to != 0);
}

/*
 * Check if a message is a Request to which we should reply.
 */
static inline int kbus_msg_wants_us_to_reply(const kbus_message_t       *msg)
{
  return (msg->flags & KBUS_BIT_WANT_A_REPLY) &&
         (msg->flags & KBUS_BIT_WANT_YOU_TO_REPLY);
}

/*
 * Compare two message ids.
 *
 * Returns -1 if id1 < id2, 0 if id1 == id2, +1 if id1 > id2.
 */
static inline int kbus_msg_compare_ids(const kbus_msg_id_t  *id1,
                                       const kbus_msg_id_t  *id2)
{
  if (id1->network_id == id2->network_id) {
    if (id1->serial_num == id2->serial_num)
      return 0;
    else if (id1->serial_num < id2->serial_num)
      return -1;
    else
      return 1;
  } else if (id1->network_id < id2->network_id)
    return -1;
  else
    return 1;
}

#ifdef __cplusplus
}
#endif

// Local Variables:
// tab-width: 8
// indent-tabs-mode: nil
// c-basic-offset: 2
// End:
// vim: set tabstop=8 shiftwidth=2 softtabstop=2 expandtab:
#endif /* _LKBUS_H_INCLUDED_ */
```

## 5.3 libkbus/limpet.h

```
/*
 * ***** BEGIN LICENSE BLOCK *****
 * Version: MPL 1.1
 *
 * The contents of this file are subject to the Mozilla Public License Version
 * 1.1 (the "License"); you may not use this file except in compliance with
 * the License. You may obtain a copy of the License at
 * http://www.mozilla.org/MPL/
 *
 * Software distributed under the License is distributed on an "AS IS" basis,
 * WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License
 * for the specific language governing rights and limitations under the
 * License.
 *
 * The Original Code is the KBUS Lightweight Linux-kernel mediated
 * message system
 *
 * The Initial Developer of the Original Code is Kynesim, Cambridge UK.
 * Portions created by the Initial Developer are Copyright (C) 2010
 * the Initial Developer. All Rights Reserved.
 *
 * Contributor(s):
 *   Kynesim, Cambridge UK
 *   Tony Ibbs <tibs@tonyibbs.co.uk>
 *
 * ***** END LICENSE BLOCK *****
 */

#ifndef _LIMPET_H_INCLUDED_
#define _LIMPET_H_INCLUDED_

#ifdef __cplusplus
extern "C" {
#endif

// NOTE that the middle portion of this file is autogenerated from limpet.c
// so that the function header comments and function prototypes may be
// automatically kept in-step. This allows me to treat the C file as the main
// specification of the functions it defines, and also to keep C header
// comments in the C file, which I find easier when keeping the comments
// correct as the code is edited.
//
// The Python script extract_hdrs.py is used to perform this autogeneration.
// It should transfer any C function marked as 'extern' and with a header
// comment (of the '/*...*...*/' form).

/*
 * The information needed to transform KBUS messages for Limpets:
 *
 * 1. after reading from KBUS and before writing to the other Limpet
 * 2. after receiving from the other Limpet and before writing to KBUS
 *
 * This is created by a call to kbus_limpet_new_context(), and freed
 * by a call to kbus_limpet_free_context().
 */
struct kbus_limpet_context;
```

```
typedef struct kbus_limpet_context kbus_limpet_context_t;


/*
 * Length of an array sufficient to hold the parts of a message header that
 * we need to send to/receive from the other Limpet.
 */
#define KBUS_SERIALISED_HDR_LEN 16

/*
 * Limpet specific messages.
 *
 * The "RemoteError" message has an error number appended to its name.
 */
#define KBUS_MSG_NOT_SAME_KSOCK         "$.KBUS.Replier.NotSameKsock"
#define KBUS_MSG_REMOTE_ERROR_PREFIX    "$.KBUS.RemoteError."

// -------- TEXT AFTER THIS AUTOGENERATED - DO NOT EDIT --------
// Autogenerated by extract_hdrs.py on 2013-01-16 (Wed 16 Jan 2013) at 11:47

/*
 * Given a KBUS message, set the `result` array to its content, suitable for
 * sending across the network
 *
 * Ignores the message's name and data pointers.
 *
 * Thus we end up with::
 *
 *  result[0]  = msg->start_guard
 *  result[1]  = msg->id.network_id
 *  result[2]  = msg->id.serial_num
 *  result[3]  = msg->in_reply_to.network_id
 *  result[4]  = msg->in_reply_to.serial_num
 *  result[5]  = msg->to
 *  result[6]  = msg->from
 *  result[7]  = msg->orig_from.network_id
 *  result[8]  = msg->orig_from.local_id
 *  result[9]  = msg->final_to.network_id
 *  result[10] = msg->final_to.local_id
 *  result[11] = msg->extra
 *  result[12] = msg->flags
 *  result[13] = msg->name_len
 *  result[14] = msg->data_len
 *  result[15] = msg->end_guard
 */
extern void kbus_serialise_message_header(kbus_message_t *msg,
                                          uint32_t        result[KBUS_SERIALISED_HDR_LEN]);

/*
 * Given a serialised message header from the network, set the message's header
 *
 * Leaves the message's name and data pointers unset (NULL).
 */
extern void kbus_unserialise_message_header(uint32_t        serial[KBUS_SERIALISED_HDR_LEN],
                                            kbus_message_t *msg);

/*
 * Prepare for Limper handling on the given Ksock, and return a Limpet context.
```

```
 *
 * This function binds to the requested message name, sets up Replier Bind
 * Event messages, and requests only one copy of each message.
 *
 * - 'ksock' is the Ksock which is to this end of our Limpet. It must be open
 *   for read and write.
 * - 'network_id' is the network id which identifies this Limpet. It is set in
 *   message ids when we are forwarding a message to the other Limpet. It must
 *   be greater than zero.
 * - 'other_network_id' is the network if of the other Limpet. It must not be
 *   the same as our_network_id. It must be greater than zero.
 * - 'message_name' is the message name that this Limpet will bind to, and
 *   forward. This will normally be a wildcard, and defaults to "$.*". Other
 *   messages will treated as ignorable. A copy is taken of this string.
 * - if 'verbosity' is:
 *
 *   * 0, we are as silent as possible
 *   * 1, we announce ourselves, and output any error/warning messages
 *   * 2 (or higher), we output information about each message as it is
 *     processed.
 *
 * 'context' is the Limpet context, allocated by this function. Free it with
 * ksock_limpet_free_context() when it is no longer required.
 *
 * Returns 0 if all goes well, a negative number (``-errno``) for
 * failure (in particular, returns -EBADMSG if 'message_name' is NULL or too
 * short).
 */
extern int kbus_limpet_new_context(kbus_ksock_t              ksock,
                                   uint32_t                  network_id,
                                   uint32_t                  other_network_id,
                                   char                     *message_name,
                                   uint32_t                  verbosity,
                                   kbus_limpet_context_t   **context);

/*
 * Change the verbosity level for a Limpet context
 */
extern void kbus_limpet_set_verbosity(kbus_limpet_context_t *context,
                                      uint32_t               verbosity);

/*
 * Free a Kbus Limpet context that is no longer required.
 *
 * After freeing 'context', it will be set to a pointer to NULL.
 *
 * If 'context' is is already a pointer to NULL, this function does nothing.
 */
extern void kbus_limpet_free_context(kbus_limpet_context_t  **context);

/*
 * Given a message read from KBUS, amend it for sending to the other Limpet.
 *
 * Returns:
 *
 * * 0 if the message has successfully been amended, and should be sent to
 *   KBUS.
 * * 1 if the message is not of interest and should be ignored.
```

```
 * * A negative number (``-errno``) for failure.
 */
extern int kbus_limpet_amend_msg_from_kbus(kbus_limpet_context_t *context,
                                           kbus_message_t        *msg);

/*
 * Given a message read from the other Limpet, amend it for sending to KBUS.
 *
 * * 'context' describes the Limpet environment
 * * 'msg' is the message to be amended. It will be changed appropriately.
 *   Note that the message data will never be touched.
 * * 'error' will be NULL or an error message to be sent to the other Limpet.
 *   In the latter case, it is up to the caller to free it.
 *
 * Returns:
 *
 * * 0 if the message has successfully been amended, and should be sent to
 *   KBUS.
 * * 1 if the message is not of interest and should be ignored.
 * * 2 if an error occurred, and the 'error' message should be sent (back)
 *   to the other Limpet (in this case the original error should not be
 *   send to KBUS).
 * * A negative number (``-errno``) for failure.
 */
extern int kbus_limpet_amend_msg_to_kbus(kbus_limpet_context_t  *context,
                                         kbus_message_t      *msg,
                                         kbus_message_t     **error);

/*
 * Convert the data of a Replier Bind Event message to network order.
 *
 * Does not check the message name, so please only call it for
 * messages called "$.ReplierBindEvent" (KBUS_MSG_NAME_REPLIER_BIND_EVENT).
 */
extern void kbus_limpet_ReplierBindEvent_hton(kbus_message_t  *msg);

/*
 * Convert the data of a Replier Bind Event message to host order.
 *
 * Does not check the message name, so please only call it for
 * messages called "$.ReplierBindEvent" (KBUS_MSG_NAME_REPLIER_BIND_EVENT).
 */
extern void kbus_limpet_ReplierBindEvent_ntoh(kbus_message_t  *msg);

/*
 * If sending to our Ksock failed, maybe generate a message suitable for
 * sending back to the other Limpet.
 *
 * 'msg' is the message we tried to send, 'errnum' is the error returned
 * by KBUS when it tried to send the message.
 *
 * 'error' is the new error message. The caller is responsible for freeing
 * 'error'.
 *
 * An 'error' message will be generated if the original message was a Request,
 * and (at time of writing) not otherwise.
 *
 * Returns
```

```
 *
 * * 0 if all goes well (in which case 'error' is non-NULL).
 * * 1 if there is no need to send an error to the other Limpet, 'error' is
 *   NULL, and the event should be ignored.
 * * A negative number (``-errno``) for failure.
 */
extern int kbus_limpet_could_not_send_to_kbus_msg(kbus_limpet_context_t  *context,
                                                  kbus_message_t      *msg,
                                                  int                  errnum,
                                                  kbus_message_t     **error);
// -------- TEXT BEFORE THIS AUTOGENERATED - DO NOT EDIT --------

#ifdef __cplusplus
}
#endif

// Local Variables:
// tab-width: 8
// indent-tabs-mode: nil
// c-basic-offset: 2
// End:
// vim: set tabstop=8 shiftwidth=2 softtabstop=2 expandtab:
#endif /* _LIMPET_H_INCLUDED_ */
```

CHAPTER 6

# Utilities

In the KBUS source tree, there is a `utils` directory, which contains a variety of useful tools.

## 6.1 errno.py

`errno.py` takes an `errno` integer or name and prints out both the "normal" meaning of that error number, and also (if there is one) the KBUS use of it. For instance:

```
$ errno.py 1
Error 1 (0x1) is EPERM: Operation not permitted
$
$ errno.py EPIPE
EPIPE is error 32 (0x20): Broken pipe

KBUS:
On attempting to send 'to' a specific replier, the replier with that id
is no longer bound to the given message's name.
```

## 6.2 kmsg

This is a simple standalone tool for sending messages to KBUS, for testing purposes. Run it with no arguments (or with `-help`) to get help.

It can send an announcement, send a message and wait for a reply, or bind as listener/replier and report messages as they are received.

Example usage:

```
$ ./kmsg send $.Fred s Hellow
Msg data:
48 65 6c 6c 6f 77
> Sending $.Fred [want_reply? 0]
<Announcement '$.Fred' data=Hellow>
> Sent message 0:1 ..
```

## 6.3 runlimpet and runlimpet.py

These are C and Python versions of the same utility, to run a Limpet. Their command lines are the same - run with no arguments, or with `-help`, to get help.

Example usage - on one machine (the "server" - the host name is not actually used on the server):

```
$ ./runlimpet -server -id 1 -kbus 2  ignored_host:1234
C Limpet: Server via TCP/IP, address 'ignored_host' port 1234 for KBUS 2, using network id 1, listen
```

and on the "client", 10.29.27.95:

```
$ ./runlimpet.py -client -id 2 -kbus 1  10.29.27.95:1234
Python Limpet: Client via ('10.29.27.95', 1234) for KBUS 1, using network id 2
Connected to "localhost:1234" as client
```

(by design, it should not matter whether you use the C or Python Limpet, as they should behave identically).

CHAPTER 7

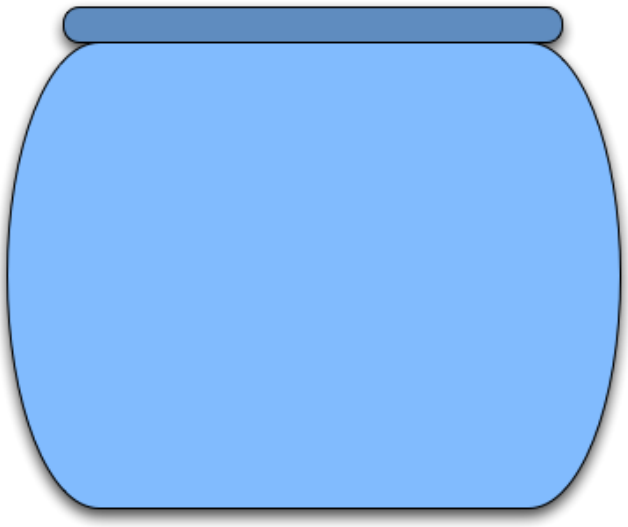
# KBUS Limpets - an introduction with goldfish

**Note:** This is a placeholder for the final version of this section.

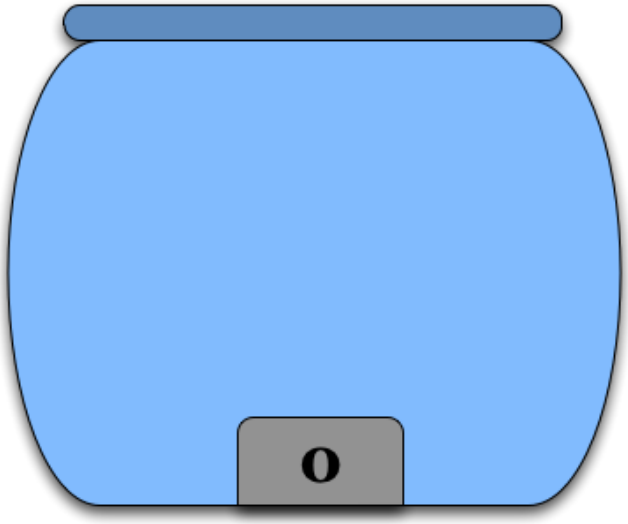
This is a metaphorical goldfish  called A.

This is another metaphorical goldfish  called B.

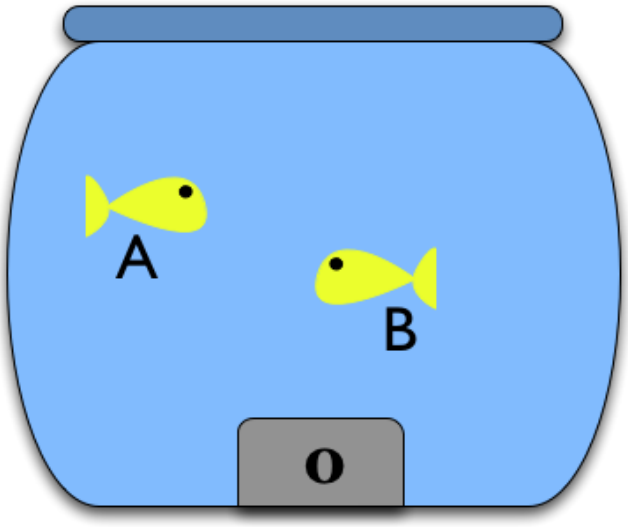
A and B live in a metaphorical goldfish bowl.



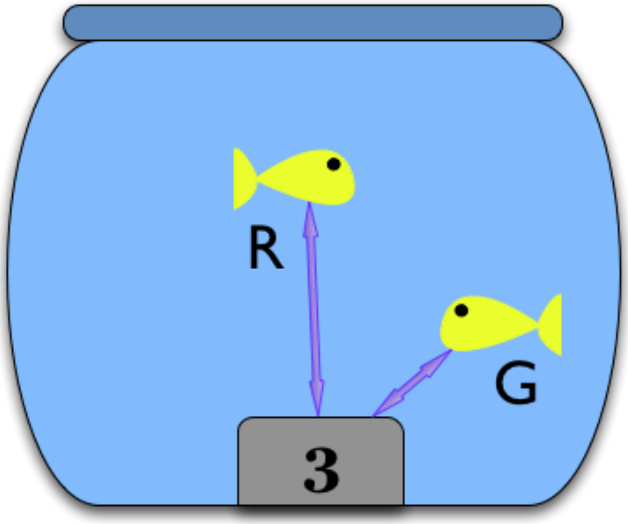
Each metaphorical goldfish bowl has a KBUS device in it - this bowl has KBUS device 0:



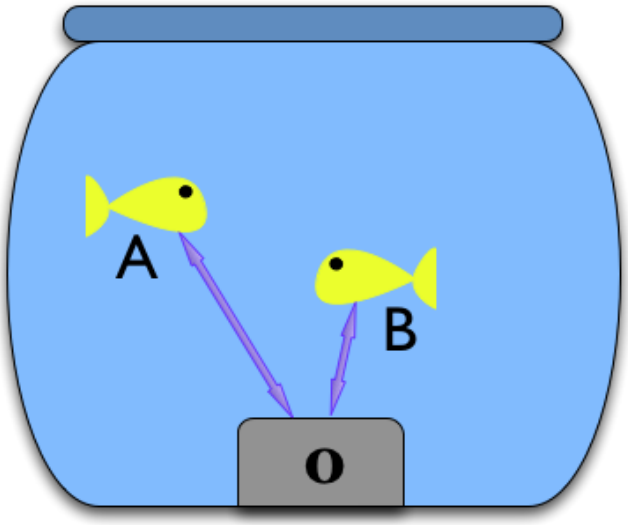
Here are the two fish in their bowl.




Metaphorical goldfish are simple creatures. They can only communicate with each other using KBUS messages.




Here is another metaphorical goldfish bowl. This one contains metaphorical goldfish called R and G. Their bowl has KBUS device 3.

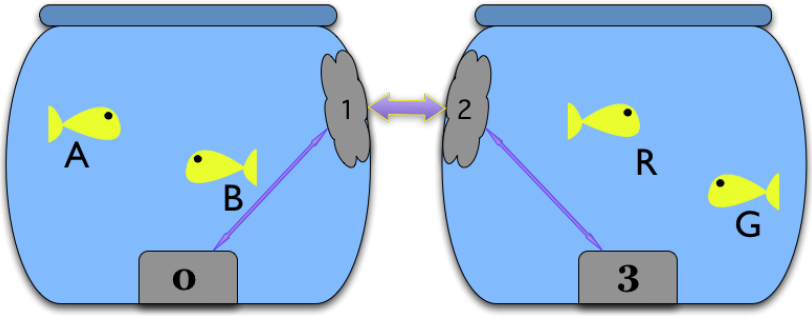



Unfortunately, A and B cannot communicate with R and G. Even if the two metaphorical goldfish bowls are running on the same computer, KBUS does not permit sending KBUS messages between different KBUS devices.

Luckily, KBUS provides Limpets.

A Limpet lives on the side of a metaphorical goldfish bowl, and communicates with another Limpet on another metaphorical bowl.

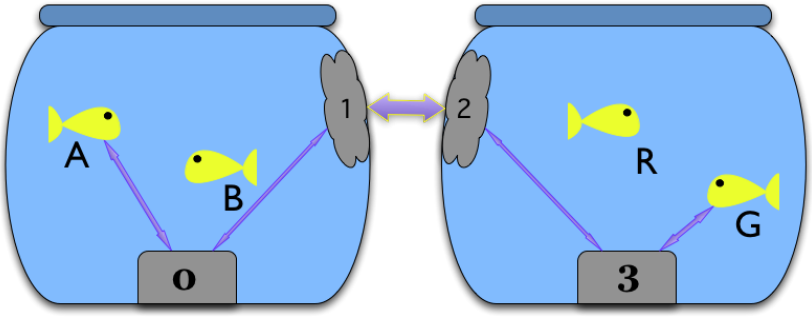
Here we have Limpet 1 connected to a Ksock on KBUS device 0, and Limpet 2 connected to a Ksock on KBUS device 3:




Limpets always come in pairs. Each Limpet can proxy KBUS messages from the KBUS device in its metaphorical goldfish bowl to and from the other Limpet in its pair.

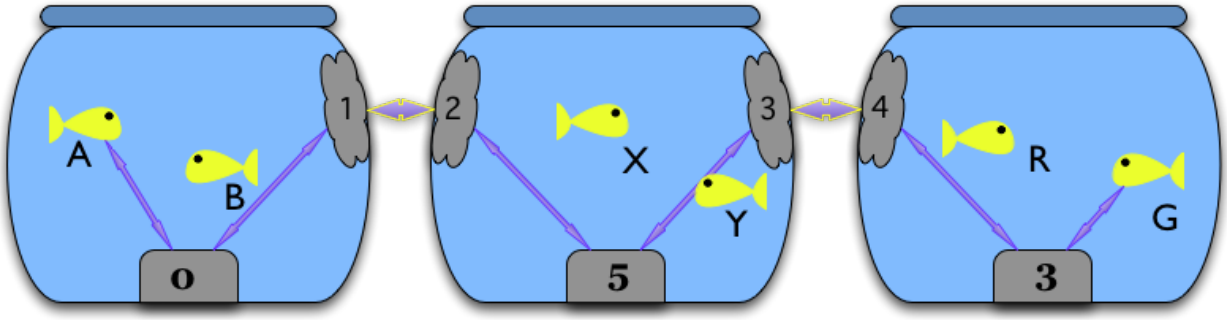
Think of them as using very low power line-of-sight lasers to send messages between each other.

So a pair of Limpets hide the fact that the two KBUS devices are not the same. This means that fish A and G can talk just as if they were in the same bowl:




To the metaphorical goldfish, it is as if the messages magically pass between the two KBUS devices, and thus A and B can communicate with R and G.

This mechanism even allows such communication if there are intermediate bowls:




## 7.1 A few more technical details

KBUS Limpets are currently experimental, and have not been extensively tested yet. Limpet daemons are available in Python and C (which intercommunicate happily).

In order to keep Limpets and their implementation as simple as possible, we're willing to put up with some limitations:

1. We use TCP/IP or named sockets to communicate between Limpets, which means one Limpet in each pair has to be a server, and thus started up first.
2. All Limpets on a connected network must have unique ids.

3. Limpets may not be used to form a network with loops in it. This greatly simplifies message management.
4. We assume a "safe" or trusted network - if it acts like a Limpet, it is a Limpet.
5. Finally, one cannot connect both ends of a Limpet pair to the same KBUS device (the same KBUS device number on different computers is OK, of course).

CHAPTER 8

# KBUS Limpets - an introduction

## 8.1 The problem, in brief

By design, KBUS does not allow message sending between KBUS devices.

Also, KBUS the-kernel-module only provides message sending on a single machine.

In the KBUS tradition of trying to provide simple, but just sufficient, solutions, KBUS Limpets allow intercommunication between pairs of KBUS devices, possibly on different machines.

## 8.2 Summary

A Limpet proxies KBUS message between a KBUS device and another Limpet.

Thus one has a connection something like:

```
KBUS<x> <----> L<x> : L<y> <----> KBUS<y>
```

The paired Limpets communicate via a socket.

To someone talking to KBUS<x>, it should appear as if messages to/from someone using KBUS<y> are sent/received directly. In particular, Requests and Replies (including Stateful Requests) are proxied transparently.

## 8.3 Restrictions and caveats

There are various restrictions on what Limpets can do and how they must be used. These are mostly intrinsic in the approach taken, and avoiding them would require doing something more sophisticated.

- The name is unintuitive. Sorry.
- All Limpets that can be reached by a message must have distinct network ids.

  Limpet network ids are used to identify the Limpet's pair, and also for determining if a message has originated with this particular Limpet. If the network ids are not unique, this will go horribly wrong.
- Limpets do not support closed loops. Thus the "network" formed by Limpets and KBUS devices must be "open" - for instance, a tree structure or star.

  If a loop is formed in the network, messages may go round and round forever. And other bad stuff.

- When starting up a Limpet pair, one must be designated the server and the other the client (in the traditional socket handling manner). This is a nuisance, but doing otherwise would be more complex, and possibly unreliable.
- Don't connect both ends of a Limpet pair to the same KBUS device. Just don't.
- Limpets intrinsically assume a "safe" network, i.e., there is no way of "proving" that the end of a message passing chain is a proper Limpet, rather than someone spoofing one.

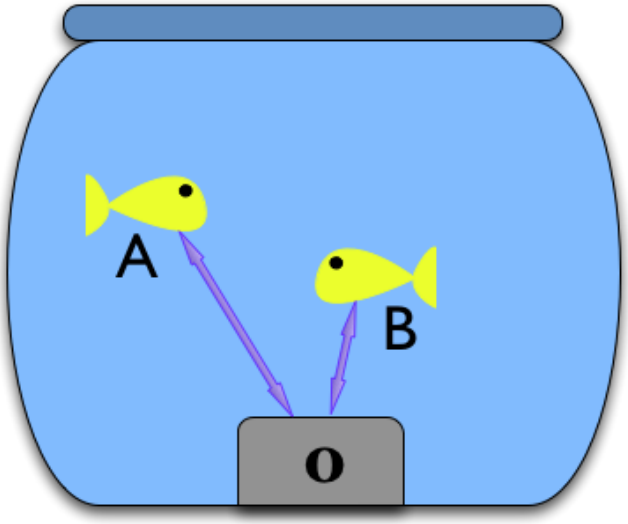
  In particular, if we have a chain (e.g., x->K1->L1:L2->K2->L3:L4->K3->y) and are passing through a Stateful Request, L2 has to trust that the Replier for the message on K2 is actually a Limpet who is going to pass the message on (ultimately to y). The message is only marked with who it was originally from (x) and who it is aimed at (y), and if L3 is not actually a Limpet (but someone who has taken its place), there is little that we can do about it.
- KBUS itself is intended to give a very good guarantee that a Request will always generate a Reply (of some sort). Whilst the Limpet system tries to provide a reasonably reliable mechanism, there is no way that it can be as robust in this matter as just talking to a KBUS device.
- Limpets default to proxying "$.*" (i.e., all messages). It is possible to proxy something different, but this is untested, and actually of unproven utility (I just had a feeling it might be useful). So if you try, cross your fingers and let me know the results.
- It *is* possible to tell if a message is going through Limpets, by inspection of the message id (the network id will be set when it is intermediate networks), and the use of the "originally from" and "finally to" fields. Thus the use of Limpets is not strictly transparent.

  Also, there are some specialised messages returned only by Limpets.

  I do not believe this to be a problem. It may possibly be an advantage.
- The most important problem with Limpets is that they are not (at the moment) particularly well tested. Although I've done a fair amount of paper-and-pen figuring of message/KBUS/Limpet interaction, and some testing, it is still possible that I've missed a whole chunk of necessary functionality. So please treat the whole thing as heavily ALPHA for the moment (as of February 2010).

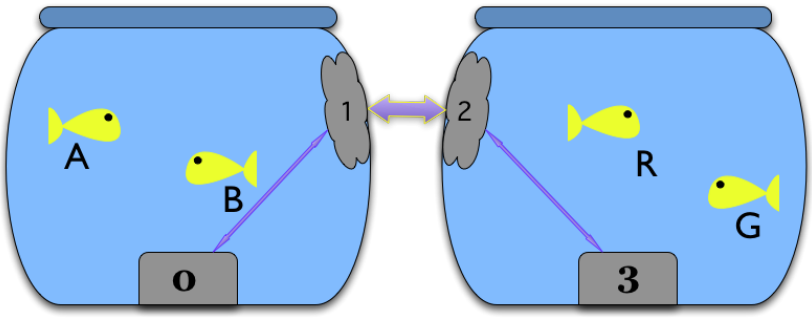
## 8.4 With goldfish bowls

Consider a particular machine as a goldfish bowl. Inside is a KBUS kernel module, and the contents of the bowl communicate with this (and thus each other) in the normal KBUS manner.



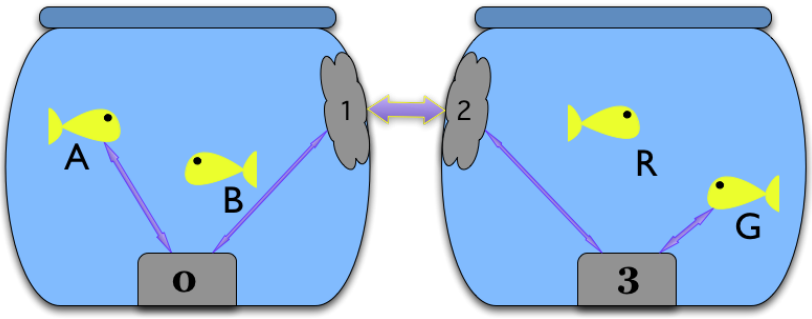

Now consider another goldfish bowl. We'd like to be able to make the two KBUS kernel modles (one in each) communicate.

So, let's place a limpet on the inside of each bowl's glass. Each limpet can communicate with the other using a simple laser communications link (so they're clever cyborg limpets), and each limpet can also communicate with its KBUS kernel module.




So the Limpet needs to proxy things for KBUS users in its bowl to the other bowl, and back again.

So if goldfish G in Bowl 3 wants to bind as a listener to message `$.Gulp`, then we want the Limpet in Bowl 0 to forward all `$.Gulp` messages from its KBUS kernel module, and the Limpet in Bowl 3 then needs to echo them to the KBUS kernel module in its bowl. So when goldfish A sends a `$.Gulp` message, goldfish G will hear it:




What if goldfish G wants to bind as a Replier for message `$.Gulp`? Limpets handle that as well, by binding as a proxy-replier in the other goldfish bowls.

So:

- Goldfish G binds as a replier for `$.Gulp`.
- KBUS device 3 sends out a Replier Bind Event message, saying that goldfish G has bound as a replier for `$.Gulp`.
- Limpet 2 receives this message, and tells Limpet 1.
- Limpet 1 binds as a replier for `$.Gulp`, on KBUS device 0.

This allows goldfish A and G to interact with a Request/Reply sequence as normal:

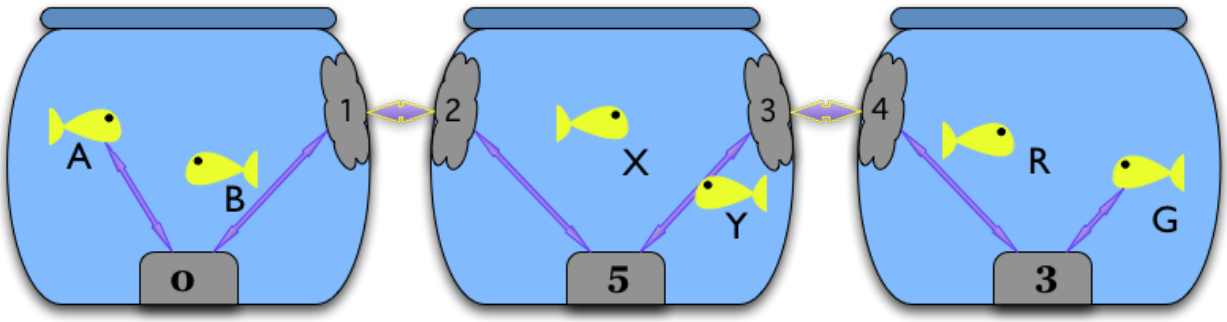
- Goldfish A send a request `$.Gulp`.
- Limpet 1 receives it, as the Replier for that message on KBUS device 0.
- Limpet 1 sends the message to Limpet 2.
- Limpet 2 sends the message, as a Request, to KBUS device 3.
- Goldfish G receives the message, marked as a Request to which it should reply.
- Goldfish G replies to the request.
- Limpet 2 receives the reply (since it issued the request on this KBUS device).
- Limpet 2 sends the message to Limpet 1.
- Limpet 1 uses the message to form *its* reply, which it then sends to KBUS device 0, since in this bowl *it* is the replier.
- Goldfish A receives the reply.

Handling Stateful Requests (and Replies) needs a bit more infrastructure, but is essentially handled by the same mechanism, although we need to show it in a bit more detail this time.

> (Stateful transactions use the message header `orig_from` and `final_to` fields. When a message is sent through a Limpet, the `orig_from` indicates both the original Ksock (goldfish G) and also the first Limpet (Limpet 2). This can then be copied to the `final_to` field of a Stateful Request to indicate that it really is goldfish G that is wanted, even though goldfish A can't "see" them.)

*TODO: insert a detailed explanation of how Stateful Transactions work*

These mechanisms will also work when there are intermediate bowls:




## 8.5 Python and C implementations

**Note:** This section is out-of-date, as the example socket-based Limpet implementations are no longer in the C library or the Python module. This section will be updated later on...

(note to self: remember the Python kbus.limpet.LimpetExample class)

Limpets were originally developed in Python.

```
>>> from kbus import run_a_limpet
>>> import socket
>>> run_a_limpet(True, '/tmp/limpet-socket', socket.AF_UNIX, 0, 1)
```

There is a Limpet class, and a `runlimpet.py` utility in `kbus/utils`.

Subsequently, a C implementation has been added to libkbus:

```
err = run_limpet(kbus_device, message_name, is_server, address, port,
                 network_id, termination_message, verbosity);
if (err) return 1;
```

and there is a `runlimpet` utility in `kbus/utils`, with an identical command line to the Python equivalent.

At the moment, the logging messages output by these two are not identical, but otherwise their behaviour should be, and in particular it should be possible to run a C Limpet communicating with a Python Limpet.

The `test_limpet.py` utility, in `kbus/python/sandbox`, provides some limited testing of limpets. It defaults to testing the Python version, but if run as `./test_limpet.py C` will test the C version. It is *not* a robust test, as it doesn't always give the same results (for reasons I've still to figure out, but probably timing issues). It's also an incredibly unrealistic use of the limpet mechanism.

## 8.6 Network protocol

When a Limpet starts up, it contacts its pair to swap network ids.

Thus each Limpet sends:

```
HELO
<network_id>                -- an unsigned 32 bit integer
```

when <network_id> is in network order.

Otherwise, Limpets swap "entire" messages, but omitting the <name> and <data> pointers (which would by definition be NULL for an "entire" message). Thus:

```
start_guard
  id.network_id
  id.serial_num
  in_reply_to.network_id
  in_reply_to.serial_num
  to
  from_
  orig_from.network_id
  orig_from.local_id
  final_to.network_id
  final_to.local_id
  extra
  flags
  name_len
  data_len
end_guard

name, including 0 byte terminator, padded to 4-byte boundary
```

```
if data_len > 0:
  data, padded to 4-byte boundary

end_guard
```

The various integer fields in the header are in network order.

Name/data padding is done with zero bytes.

A Replier Bind Event message is treated specially, in that the <is_binder>, <binder_id> and <name_len> fields in the data are automatically converted to/from network order as the message is written/read.

All other message data is just treated as a byte stream.

CHAPTER 9

# KBUS Python bindings for Limpets

Limpet - a mechanism for proxying KBUS messages to/from another Limpet.

This allows messages to be communicated from one KBUS device to another, either on the same machine or over a network.

## 9.1 Limpets

See also the `LimpetExample` class, which shows how a limpet might be deployed in practise.

### 9.1.1 Limpet and run_a_limpet

**class** `kbus.limpet.`**`LimpetKsock`**(*which*, *our_network_id*, *their_network_id*, *message_name='$.*'*, *verbosity=1*, *termination_message=None*)

Bases: `kbus.ksock.Ksock`

A Limpet proxies KBUS messages to/from another Limpet.

This class wraps itself around a Ksock, transforming messages as they are read from the Ksock, or written to it.

**`bind`**(**args*)

Not meaningful for this class.

**`could_not_send_to_kbus_msg`**(*msg*, *exc*)

If a send to our Ksock failed, call this to generate a message.

'msg' is the message we tried to send.

'exc' is the IOError exception that was raised by the failed call of 'send_msg()'

We return an appropriate message to send to the other Limpet, or None if we could not determine one.

**`discard`**()

Not meaningful for this class.

We only support reading an entire message in one go with *`read_next_msg()`*.

**`len_left`**()

Not meaningful for this class.

We only support reading an entire message in one go with *`read_next_msg()`*.

**read_data**(*count*)
Not meaningful for this class.

We only support reading an entire message in one go with *read_next_msg()*.

**read_msg**(*length*)
Read a Message of length *length* bytes.

It is assumed that *length* was returned by a previous call of *next_msg()*. It must be large enough to cause the entire message to be read.

After the data has been read, it is passed to *Message()* to construct a message instance, which is returned.

Returns None if there was nothing to be read, or if the message read is one that this Limpet should ignore.

**read_next_msg**()
Read the next Message completely.

Equivalent to a call of *kbus.Ksock.next_msg()*, followed by reading the appropriate number of bytes and passing that to *Message()* to construct a message instance, which is returned.

Returns None if there was nothing to be read, or if the message read is one that this Limpet should ignore.

**report_replier_binds**(*report_events=True*, *just_ask=False*)
Not meaningful for this class.

Limpets require this option to be set in order to work properly, and do not allow the user to change it.

**send_msg**(*message*)
Write a *Message* (from the other Limpet) to our *Ksock*, and send it.

Entirely equivalent to calling *write_msg()* and then *send()*, and returns the MessageId of the sent message, like *send()* does.

If the message was one we need to ignore (i.e., we're not interested in sending it), raises *NoMessage*.

If we need to send a message back to the other Limpet, then that exception will have the messsage as its argument.

**unbind**(**args*)
Not meaningful for this class.

**want_messages_once**(*only_once=False*, *just_ask=False*)
Not meaningful for this class.

Limpets require this option to be set in order to work properly, and do not allow the user to change it.

**write_data**(*data*)
Not meaningful for this class.

We only support writing an entire message in one go with *send_msg()*.

**write_msg**(*message*)
Not meaningful for this class.

We only support writing and sending an entire message in one go with *send_msg()*.

kbus.limpet.**run_a_limpet**(*is_server*, *address*, *family*, *kbus_device*, *network_id*, *message_name='$.*'*, *termination_message=None*, *verbosity=1*)
Run a Limpet.

A Limpet has two "ends":

1. *kbus_device* specifies which KBUS device it should communicate with, via `ksock = Ksock(kbus_device, 'rw')`.

2.*socket_address* is the address for the socket used to communicate with its paired Limpet. This should generally be a path name (if communication is with another Limpet on the same machine, via Unix domain sockets), or a `(host, port)` tuple (for communication with a Limpet on another machine, via the internet).

Messages received from KBUS get sent to the other Limpet.

Messages sent from the other Limpet get sent to KBUS.

- *is_server* is true if we are the "server" of the Limpet pair, false if we are the "client"
- *address* is the socket address we use to talk to the other Limpet
- *family* is AF_UNIX or AF_INET, determining what sort of address we want – a pathname for the former, a <host>:<port> string for the latter
- *kbus_device* is which KBUS device to open
- *network_id* is the network id to set in message ids when we are forwarding a message to the other Limpet. It must be greater than zero.
- *message_name* is the name of the message (presumably a wildcard) we are forwarding
- if *termination_message* is not None, then we will stop when a message with that name is read from KBUS
- if *verbosity* is 0, we don't output any "useful" messages, if it is 1 we just announce ourselves, if it is 2 (or higher) we output information about each message as it is processed.

### 9.1.2 Limpet exceptions

**class** kbus.limpet.**GiveUp**
Bases: exceptions.Exception

Signifies a fatal condition such as a failure to connect to the other Limpet, or when the `termination_message` of a *LimpetKsock* is read.

**class** kbus.limpet.**OtherLimpetGoneAway**
Bases: exceptions.Exception

The other end has closed its end of the socket.

**class** kbus.limpet.**NoMessage**
Bases: exceptions.Exception

There was no message.

**class** kbus.limpet.**BadMessage**
Bases: exceptions.Exception

We have read a badly formatted KBUS message.

**class** kbus.limpet.**ErrorMessage**(*error*)
Bases: exceptions.Exception

Something went wrong trying to send a message to KBUS.

There is an error message, for sending back to the other Limpet, in our `.error` value.

### 9.1.3 Helper functions

kbus.limpet.**parse_address**(*word*)
Work out what sort of address we have.

Returns (address, family).

kbus.limpet.**connect_as_server**(*address*, *family*, *verbosity=1*)
Connect to a socket as a server.

We start listening, until we get someone connecting to us.

Returns a tuple (listener_socket, connection_socket).

kbus.limpet.**connect_as_client**(*address*, *family*, *verbosity=1*)
Connect to a socket as a client.

Returns the socket.

kbus.limpet.**remove_socket_file**(*name*)
Attempts to clean up a socket whose address is a file in the filesystem. This currently only applies to sockets of the AF_UNIX family.

CHAPTER 10

# The KBUS documentation and sphinx

## 10.1 Pre-built documentation

For your comfort and convenience, a pre-built version of the KBus documentation is available at:

http://html.kbus.googlecode.com/hg/docs/html/index.html

or within the Mercurial `html` repository at:

`html/docs/html/index.html`

## 10.2 Building the documentation

The KBus documentation is built using Sphinx.

**Note:** The documentation needs (at least) version 0.6 of Sphinx. Recent versions of Ubuntu provide this in the `python-sphinx` package; on older versions you should use `easy_install` as described on the Sphinx website.

You also need graphviz (which provides `dot`).

With luck, the HTML in the `web` repository will be up-to-date, and you won't need to (re)build the documentation. However, if you should need to (for instance, because you've updated it), just use the Makefile:

```
make html
```

Note that the `html` repository includes `default` as a *subrepository*; this means that if you've made changes to the doc source anywhere else, you have to update the subrepository before you build in `html`, probably with a line like this:

```
hg -R kbus pull <repo>
hg -R kbus update
```

KBUS developers can push rebuilt docs back to Mercurial in the usual way; beware that the `inheritance` graphic is rebuilt every time and require `hg add` (and, ideally, the old ones removed).

## 10.3 The Python bindings

Read the `kbus-python-*.txt` files to see how individual classes and functions within kbus.py are documented. Obviously, if you add, remove or rename such, you may need to alter these files – please do so appropriately.

## 10.4 Mime type magic

In order for the documentation in the Google Code Mercurial repository to be useable as documentation, the HTML, CSS and JavaScript files in the docs/html directory tree need to have the correct mime types. Mercurial is clever enough to be able to cope with this, though Subversion needed extra help.

## 10.5 Mercurial gotchas

Sphinx believes that the contents of `docs` are transitory - i.e., that it is free to delete them if it wishes. In particular, `make clean` will delete all of the contents of `docs`.

Meanwhile, we've committed `docs/html` and its contents to Mercurial.

This used to be a problem under Subversion, but is no longer since moving to Mercurial. If you accidentally `make clean` the docs away, you can use `hg checkout` to retrieve them. With luck the dependency tracking in the `make` process will cope.

CHAPTER 11

# Indices and tables

- genindex
- modindex
- search

**Note:** This documentation is kept as reStructuredText documents, managed with Sphinx. The HTML files in the Google code repository are provided for convenience, and may not always be absolutely up-to-date with head-of-tree.

## k

kbus, 31
kbus.limpet, 103

## A

ALL_OR_FAIL (kbus.Message attribute), 34
ALL_OR_WAIT (kbus.Message attribute), 34
Announcement (class in kbus), 37

## B

BadMessage (class in kbus.limpet), 105
bind() (kbus.Ksock method), 48
bind() (kbus.limpet.LimpetKsock method), 103
BindStruct (class in kbus), 51

## C

cast() (kbus.Message method), 34
close() (kbus.Ksock method), 48
connect_as_client() (in module kbus.limpet), 106
connect_as_server() (in module kbus.limpet), 106
could_not_send_to_kbus_msg() (kbus.limpet.LimpetKsock method), 103

## D

data (kbus.Message attribute), 34
discard() (kbus.Ksock method), 48
discard() (kbus.limpet.LimpetKsock method), 103

## E

END_GUARD (kbus.Message attribute), 34
equivalent() (kbus.Message method), 34
ErrorMessage (class in kbus.limpet), 105
extract() (kbus.Message method), 34

## F

fileno() (kbus.Ksock method), 48
final_to (kbus.Message attribute), 35
find_replier() (kbus.Ksock method), 48
flags (kbus.Message attribute), 35
from_ (kbus.Message attribute), 35
from_bytes() (kbus.Announcement static method), 38
from_bytes() (kbus.Message static method), 35
from_bytes() (kbus.Reply static method), 42
from_bytes() (kbus.Request static method), 40
from_bytes() (kbus.Status static method), 44
from_message() (kbus.Announcement static method), 38
from_message() (kbus.Message static method), 35
from_message() (kbus.Reply static method), 42
from_message() (kbus.Request static method), 40
from_message() (kbus.Status static method), 45
from_sequence() (kbus.Announcement static method), 38
from_sequence() (kbus.Message static method), 35
from_sequence() (kbus.Reply static method), 42
from_sequence() (kbus.Request static method), 40
from_sequence() (kbus.Status static method), 45

## G

GiveUp (class in kbus.limpet), 105

## I

id (kbus.Message attribute), 35
in_reply_to (kbus.Message attribute), 35
IOC_BIND (kbus.Ksock attribute), 47
IOC_DISCARD (kbus.Ksock attribute), 47
IOC_KSOCKID (kbus.Ksock attribute), 47
IOC_LASTSENT (kbus.Ksock attribute), 47
IOC_LENLEFT (kbus.Ksock attribute), 47
IOC_MAGIC (kbus.Ksock attribute), 47
IOC_MAXMSGS (kbus.Ksock attribute), 47
IOC_MAXMSGSIZE (kbus.Ksock attribute), 47
IOC_MSGONLYONCE (kbus.Ksock attribute), 47
IOC_NEWDEVICE (kbus.Ksock attribute), 47
IOC_NEXTMSG (kbus.Ksock attribute), 47
IOC_NUMMSGS (kbus.Ksock attribute), 47
IOC_REPLIER (kbus.Ksock attribute), 47
IOC_REPORTREPLIERBINDS (kbus.Ksock attribute), 47
IOC_RESET (kbus.Ksock attribute), 47
IOC_SEND (kbus.Ksock attribute), 48
IOC_UNBIND (kbus.Ksock attribute), 48
IOC_UNREPLIEDTO (kbus.Ksock attribute), 48
IOC_VERBOSE (kbus.Ksock attribute), 48
is_replier (kbus.BindStruct attribute), 51
is_reply() (kbus.Message method), 35

is_request() (kbus.Message method), 36
is_stateful_request() (kbus.Message method), 36
is_synthetic() (kbus.Message method), 36
is_urgent() (kbus.Message method), 36

## K

kbus (module), 31, 47
kbus.limpet (module), 103
kernel_module_verbose() (kbus.Ksock method), 48
Ksock (class in kbus), 47
ksock_id() (kbus.Ksock method), 48

## L

last_msg_id() (kbus.Ksock method), 48
len (kbus.BindStruct attribute), 52
len (kbus.ReplierStruct attribute), 52
len_left() (kbus.Ksock method), 49
len_left() (kbus.limpet.LimpetKsock method), 103
LimpetKsock (class in kbus.limpet), 103
local_id (kbus.OrigFrom attribute), 45

## M

max_message_size() (kbus.Ksock method), 49
max_messages() (kbus.Ksock method), 49
Message (class in kbus), 31
MessageId (class in kbus), 43
msg_id (kbus.SendResultStruct attribute), 52

## N

name (kbus.BindStruct attribute), 52
name (kbus.Message attribute), 36
name (kbus.ReplierStruct attribute), 52
network_id (kbus.MessageId attribute), 44
network_id (kbus.OrigFrom attribute), 45
new_device() (kbus.Ksock method), 49
next() (kbus.Ksock method), 49
next_msg() (kbus.Ksock method), 49
NoMessage (class in kbus.limpet), 105
num_messages() (kbus.Ksock method), 49
num_unreplied_to() (kbus.Ksock method), 49

## O

orig_from (kbus.Message attribute), 36
OrigFrom (class in kbus), 45
OtherLimpetGoneAway (class in kbus.limpet), 105

## P

parse_address() (in module kbus.limpet), 105

## R

read_bindings() (in module kbus), 52
read_data() (kbus.Ksock method), 49
read_data() (kbus.limpet.LimpetKsock method), 103
read_msg() (kbus.Ksock method), 49
read_msg() (kbus.limpet.LimpetKsock method), 104
read_next_msg() (kbus.Ksock method), 49
read_next_msg() (kbus.limpet.LimpetKsock method), 104
remove_socket_file() (in module kbus.limpet), 106
ReplierStruct (class in kbus), 52
Reply (class in kbus), 41
reply_to() (in module kbus), 43
report_replier_binds() (kbus.Ksock method), 49
report_replier_binds() (kbus.limpet.LimpetKsock method), 104
Request (class in kbus), 39
return_id (kbus.ReplierStruct attribute), 52
retval (kbus.SendResultStruct attribute), 52
run_a_limpet() (in module kbus.limpet), 104

## S

send() (kbus.Ksock method), 50
send_msg() (kbus.Ksock method), 50
send_msg() (kbus.limpet.LimpetKsock method), 104
SendResultStruct (class in kbus), 52
serial_num (kbus.MessageId attribute), 44
set_max_message_size() (kbus.Ksock method), 50
set_max_messages() (kbus.Ksock method), 50
set_urgent() (kbus.Message method), 36
set_want_reply() (kbus.Announcement method), 39
set_want_reply() (kbus.Message method), 36
set_want_reply() (kbus.Request method), 40
split_replier_bind_event_data() (in module kbus), 46
START_GUARD (kbus.Message attribute), 34
stateful_request() (in module kbus), 40
Status (class in kbus), 44
SYNTHETIC (kbus.Message attribute), 34

## T

to (kbus.Message attribute), 36
to_bytes() (kbus.Message method), 36
total_length() (kbus.Message method), 36

## U

unbind() (kbus.Ksock method), 51
unbind() (kbus.limpet.LimpetKsock method), 104
URGENT (kbus.Message attribute), 34

## W

wait_for_msg() (kbus.Ksock method), 51
WANT_A_REPLY (kbus.Message attribute), 34
want_messages_once() (kbus.Ksock method), 51
want_messages_once() (kbus.limpet.LimpetKsock method), 104
WANT_YOU_TO_REPLY (kbus.Message attribute), 34
wants_us_to_reply() (kbus.Message method), 36

write_data() (kbus.Ksock method), 51
write_data() (kbus.limpet.LimpetKsock method), 104
write_msg() (kbus.Ksock method), 51
write_msg() (kbus.limpet.LimpetKsock method), 104