Goby Underwater Autonomy Project

bzr revision: 38 | bzr revision id: toby-20100615045147-xhr0p92tqfipg6sb

Generated by Doxygen 1.6.3

Sat Jun 19 16:48:44 2010

Contents

Chapter 1

Goby Underwater Autonomy Project

The Goby Underwater Autonomy Project aims to create a unified framework for multiple scientific autonomous marine vehicle collaboration, seamlessly incorporating acoustic, ethernet, wifi, and serial communications. Presently the main thrust of the project is developing a set of robust acoustic networking libraries. Goby is currently written entirely in C++, and licensed under the GNU General Public License v3 <http://www.gnu.org/licenses/gpl.html>.

1.1 Resources

- Home page: <http://gobysoft.com>
- Code, bug tracking, and answers: <https://launchpad.net/goby>.
- Manual: (html) (pdf).
- Wiki: <forthcoming>.

1.2 Developer manual

- goby-acomms: Overview of Acoustic Communications Libraries - tackle the extremely rate limited acoustic networking problem. These libraries were designed together but can operate independently for a developer looking integrate a specific component (e.g. just encoding/decoding) without committing to the entire goby-acomms stack.

1.3 Publications

- The Dynamic Compact Control Language: A Compact Marshalling Scheme for Acoustic Communications. IEEE OCEANS'10 / Sydney.

1.4 Installing Goby

1. Install bazaar (bzr) version control system:
    - Ubuntu/Debian:

```
sudo apt-get install bzr
```

- Everyone else: <http://wiki.bazaar.canonical.com/Download>

2. Check out a branch of the goby project:

   - Read-only (users of goby):

     ```
     bzr branch lp:goby
     ```

   - Read / write (developers of goby):

     (a) Register or login to launchpad: <https://launchpad.net/goby/+login>

     (b) Contact <https://launchpad.net/~tes> for access to the goby project

     (c) Let bzr know your launchpad id:

     ```
     bzr launchpad-login tes
     ```

     (d) Checkout a copy of goby:

     ```
     bzr checkout lp:goby
     ```

3. Satisfy third-party library dependences (boost, etc.):

   - Ubuntu/Debian:

     ```
     cd goby; ./DEPENDENCIES
     ```

   - Everyone else:

     ```
     cd goby; less DEPENDENCIES
     ```

4. Compile to goby/bin, goby/lib, and goby/include:

   ```
   cd goby; ./INSTALL
   ```

   To install to /usr/local (optional):

   ```
   sudo ./INSTALL
   ```

5. Fine tuning of build system (optional):

   (a) `sudo apt-get install cmake-gui`

   (b) `cmake-gui goby/build`

1.5 Goby components in other projects

- pAcommsHandler is a MOOS acoustic communications process that is based on the goby-acomms libraries. See <http://oceanai.mit.edu/moosivp/support/moos-ivp-local.html> for access to this code. For documentation and more information on pAcommsHandler, please contact Toby <https://launchpad.net/~tes>. To learn about the MOOS and MOOS-IvP projects, visit <http://www.robots.ox.ac.uk/~mobile/MOOS/wiki/pmwiki.php> and <http://www.moos-ivp.com/>

1.6 Authors

Goby is developed by T. Schneider (https://launchpad.net/~tes) with input and suggestions from many people, including but not limited to:

- H. Schmidt
- C. Murphy (https://launchpad.net/~chrismurf)
- K. Cockrell
- S. Petillo
- L. Freitag
- M. Grund

1.7 Third party projects used in Goby

Goby appreciates the availability of the following open source projects (listed with best available documentation):

- version control: bzr <http://wiki.bazaar.canonical.com/Documentation>
- build system: cmake <http://www.cmake.org/cmake/help/cmake2.6docs.html>
- general purpose libraries: boost <http://www.boost.org/doc/libs/1_34_0>
- terminal GUI library: ncurses <http://www.c-for-dummies.com/ncurses/>
- XML parsing library: Xerces-C <http://xerces.apache.org/xerces-c/program-3.html>
- asynchronous networking and serial communications library: asio <http://think-async.com/Asio/asio-1.4.1/doc/>
- cryptography: crypto++ <http://www.cryptopp.com>

Chapter 2

goby-acomms: Overview of Acoustic Communications Libraries

Table of Contents for goby-acomms: Overview of Acoustic Communications Libraries.

2.1 Quick Start

To get started using the goby-acomms libraries as quickly as possible:

1. If you haven't yet, follow instructions on Installing Goby
2. Identify which components you need:
   - Encoding and decoding from C++ types to bit-packed messages: libdccl: Encoding and decoding
   - Queuing of DCCL and CCL messages with priority based message selection: libqueue: Priority based message queuing
   - A driver for interacting with the acoustic modem firmware. Presently the WHOI Micro-Modem <http://acomms.whoi.edu/> is supported: libmodemdriver: Modem driver

- Time division multiple access (TDMA) medium access control (MAC): libamac: Medium Access Control (MAC)

3. Look at the "simple" code examples that accompany each library (dccl_simple.cpp, queue_simple.cpp, driver_simple.cpp, amac_simple.cpp). Then look at the example that uses all the libraries together: chat.cpp. The full list of examples is given in this table.

4. Refer to the rest of the documentation as needed.

Please visit <https://answers.launchpad.net/goby> with any questions.

2.2 Overview

2.2.1 Analogy to established networking systems

To start on some (hopefully) common ground, let's begin with an analogy to Open Systems Initiative (OSI) networking layers in this table. For a complete description of the OSI layers see <http://www.itu.int/rec/T-REC-X.200-199407-I/en>.

OSI Layer	Goby library	API class	Example(s)
Application	Not yet part of Goby		MOOS Application: pAcommsHandler
Presentation	libdccl: Encoding and decoding	dccl::DCCLCodec	dccl_simple.cpp two_message.cpp plusnet.cpp test.cpp chat.cpp
Session	Not used, sessions are established passively.		
Transport	libqueue: Priority based message queuing	queue::QueueManager	queue_simple.cpp chat.cpp
Network	Does not yet exist. All transmissions are considered single hop, currently. Addressing routing over multiple hops is an open and pressing research problem.		
Data Link	libmodemdriver: Modem driver	classes derived from modem::DriverBase; e.g. micromodem::MMDriver	driver_simple.cpp chat.cpp
	libamac: Medium Access Control (MAC)	amac::MACManager	amac_simple.cpp chat.cpp
Physical	Not part of Goby		Modem Firmware, e.g. WHOI Micro-Modem Firmware (NMEA 0183 on RS-232) (see Interface Guide)

2.2.2 Acoustic Communications are slow

Do not take the previous analogy too literally; some things we are doing here for acoustic communications (hereafter, acomms) are unconventional from the approach of networking on electromagnetic carriers (hereafter, EM networking). The difference is a vast spread in the expected throughput of a standard internet hardware carrier and acoustic communications. For example, an optical fiber can put through greater than 10 Tbps over greater than 100 km, whereas the WHOI acoustic Micro-Modem can (at best) do 5000 bps over several km. This is a difference of thirteen orders of magnitude for the bit-rate distance product!

2.2.3 Efficiency to make messages small is good

Extremely low throughput means that essentially every efficiency in bit packing messages to the smallest size possible is desirable. The traditional approach of layering (e.g. TCP/IP) creates inefficiencies as each layer wraps the message of the higher layer with its own header. See RFC3439 section 3 ("Layering Considered Harmful") for an interesting discussion of this issue <http://tools.ietf.org/html/rfc3439#page-7>. Thus, the "layers" of goby-acomms are more tightly interrelated than TCP/IP, for example. Higher layers depend on lower layers to carry out functions such as error checking and do not replicate this functionality.

2.2.4 Total throughput unrealistic: prioritize data

The second major difference stemming from this bandwidth constraint is that *total throughput is often an unrealistic goal*. The quality of the acoustic channel varies widely from place to place, and even from hour to hour as changes in the sea affect propagation of sound. This means that it is also difficult to predict what one's throughput will be at any given time. These two considerations manifest themselves in the goby-acomms design as a priority based queueing system for the transport layer. Messages are placed in different queues based on their priority (which is determined by the designer of the message). This means that

- the channel is always utilized (low priority data are sent when the channel quality is high)
- important messages are not swamped by low priority data

In contrast, TCP/IP considers all packets equally. Packets made from a spam email are given the same consideration as a high priority email from the President. This is a tradeoff in efficiency versus simplicity that makes sense for EM networking, but does not for acoustic communications.

2.2.5 Despite all this, simplicity is good

The "law of diminishing returns" means that at some point, if we try to optimize excessively, we will end up making the system more complex without substantial gain. Thus, goby-acomms makes some concessions for the sake of simplicity:

- Numerical message fields are bounded by powers of 10, rather than 2. Humans deal much better with decimal than binary.
- User data splitting (and subsequent stitching) is not done. This is a key component of TCP/IP, but with the number of dropped packets one can expect with acomms, at the moment this does not seem like a good idea. The user is expected to provide data that is smaller or equal to the packet size of the physical layer (e.g. 32 - 256 bytes for the WHOI Micro-Modem).

2.2.6 Component model

A relatively simple component model for the goby-acomms libraries showing the interface classes:

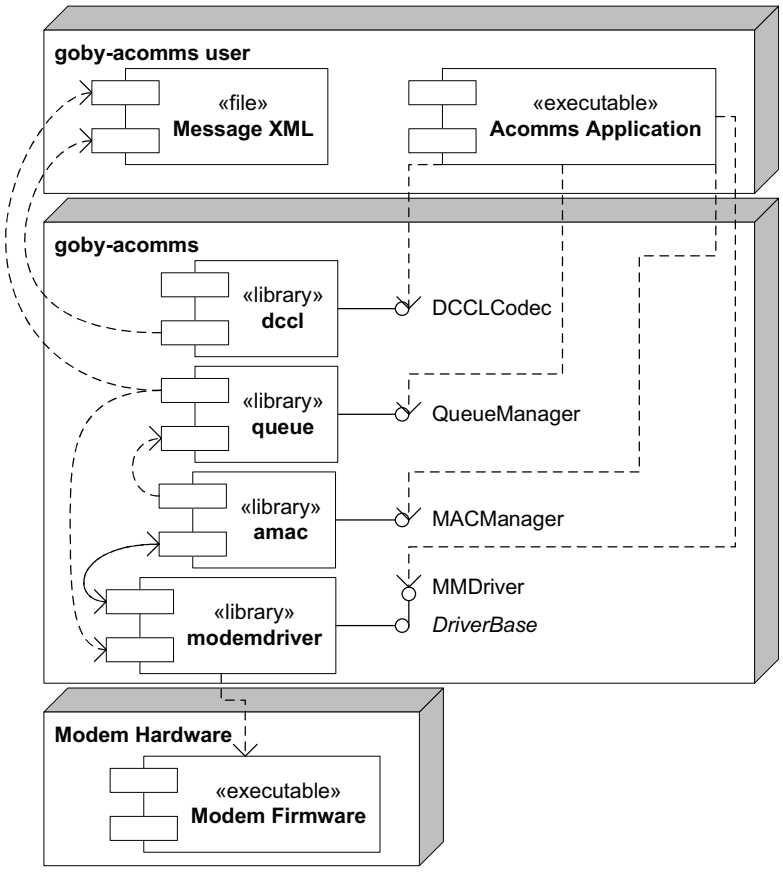

Figure 2.1: Basic overview of goby-acomms libraries.

For a more detailed model, see the UML models section.

2.3 libdccl: Encoding and decoding

Dynamic Compact Control Language (DCCL) provides a structure for defining messages to be sent through an acoustic modem. The messages are configured in XML and are intended to be easily reconfigurable, unlike the original CCL framework used in the REMUS vehicles and others (for information on CCL, see <http://acomms.whoi.edu/ccl/>. DCCL can operate within a CCL network, as the most significant byte (or CCL ID) is 0x20.

DCCL messages are packed based on boundaries determined with knowledge of the XML file. They are not self-describing as this would be prohibitively expensive in terms of data use. Thus, the sender and receiver must have a copy of the same XML file for decoding a given message. Also, each message is defined by an ID that must be unique with a network.

Table of Contents for libdccl:

2.4 libqueue: Priority based message queuing

The goby-acomms queuing library (libqueue) interacts with both the application level process that handles decoding (either through libdccl or other CCL codecs) and the modem driver process that talks directly to the modem.

On the application side, libqueue provides the ability for the application level process to push (CCL or DCCL encoded) messages to various queues and receive messages from a remote sender that correspond to messages in the same queue (e.g. you have a queue for STATUS_MESSAGE that you can push messages to you and also receive other STATUS_MESSAGEs on). The push feature is called by the application level process and received messages are passed as a callback upon receipt.

On the driver side, libqueue provides the modem driver with data upon request. It chooses the data to send based on dynamic priorities (and several other configuration parameters). It will also pack several messages from the user into a single frame from the modem to fully utilize space (e.g. if the modem frame is 32 bytes and the user's data are in 16 byte DCCL messages, libqueue will pack two user frames for each modem frame). This packing and unpacking is transparent to the application side user. Note, however, that libqueue will *not* split a user's data into frames (like TCP/IP). If this functionality is desired, it must be provided at the application layer. Acoustic communications is too unpredictable to reliably stitch together frames.

Table of Contents for libqueue:

2.5 libmodemdriver: Modem driver

The goby-acomms Modem driver library (libmodemdriver) provides an interface from the rest of goby-acomms to the acoustic modem firmware. While currently the only driver available is for the WHOI Micro-Modem, this library is written in such a way that drivers for any acoustic modem that interfaces over a serial connection and can provide (or provide abstractions for) sending data directly to another modem on the link should be able to be written. Any one who is interested in writing a modem driver for another acoustic modem should get in touch with the goby project <https://launchpad.net/goby> and see Writing a new driver.

Table of Contents for libmodemdriver:

- WHOI Micro-Modem Driver: MMDriver

- Writing a new driver

2.6 libamac: Medium Access Control (MAC)

The goby-acomms MAC library (libamac) handles access to the shared medium, in our case the acoustic channel. We assume that we have a single band for transmission so that if vehicles transmit simultaneously, collisions will occur between messaging. Therefore, we use time division multiple access (TDMA) schemes, or "slotting". Networks with multiple frequency bands will have to employ a different MAC scheme or augment libamac for the frequency division multiple access (FDMA) scenario. The default TDMA scheme used also includes basic peer discovery and subsequent expiry of peers after a long time of silence.

For legacy support and scenarios where fine control is needed, "polling" is also provided. This is a TDMA enforced by a central computer (the "poller"). The "poller" sends a request for data from a list of nodes in sequential order. The advantage of polling is that synchronous clocks are not needed and the MAC scheme can be changed on short notice by the topside operator.

Table of Contents for libamac:

- Supported MAC schemes

- Interacting with the amac::MACManager

2.7 UML models

Model that describes the static structure of goby-acomms as a whole:

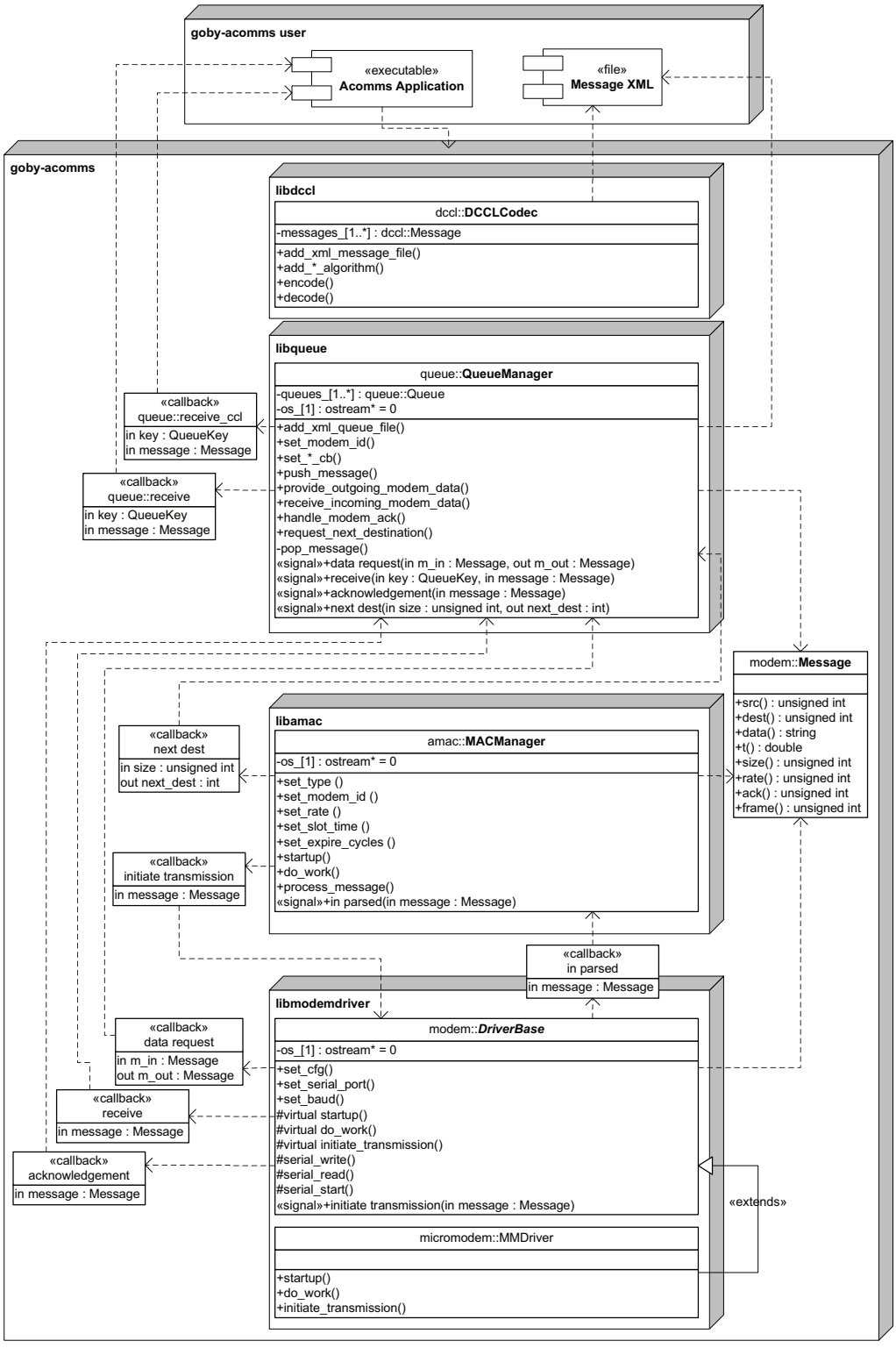

Figure 2.2: UML Model that shows detailed overview of class interfaces and their connections when using all the goby-acomms libraries together. Note that is possible to replace any of the libraries as long as the user handles the nessary callbacks (e.g. to replace libqueue you need functions to handle the data request, acknowledgement, and receive callbacks from libmodemdriver).

Model that gives the sequence for sending a message with goby-acomms:

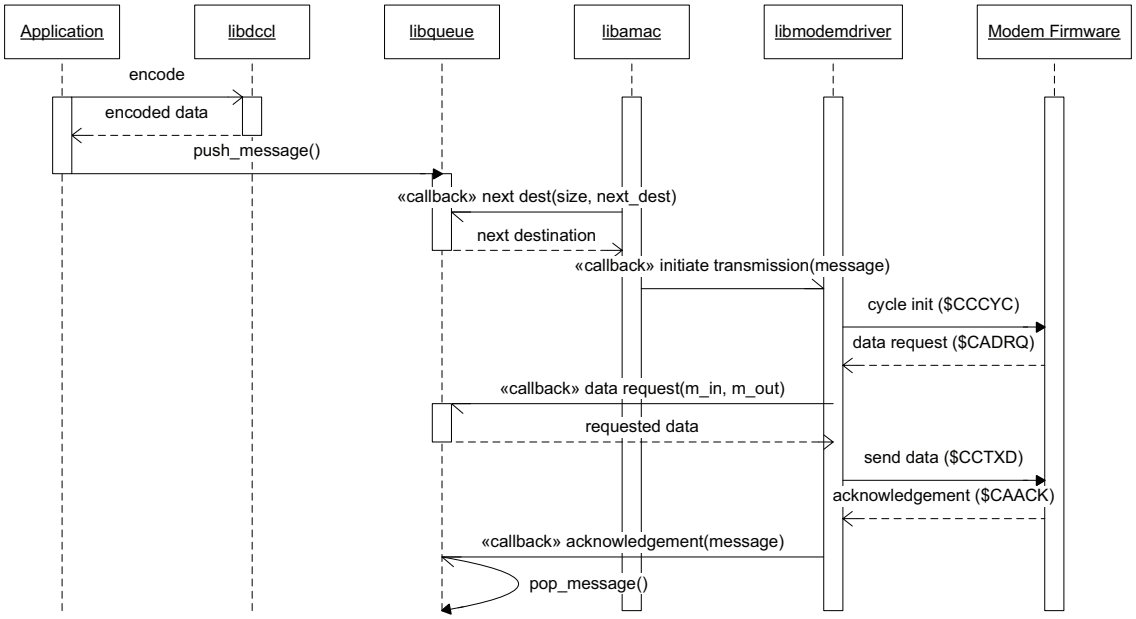

Figure 2.3: UML model that gives the sequence of calls required in sending a message using goby-acomms. The WHOI Micro-Modem is used as example firmware but the specific libmodemdriver-firmware interaction will depend on the acoustic modem used.

Model that shows the commands needed to start and keep goby-acomms running:

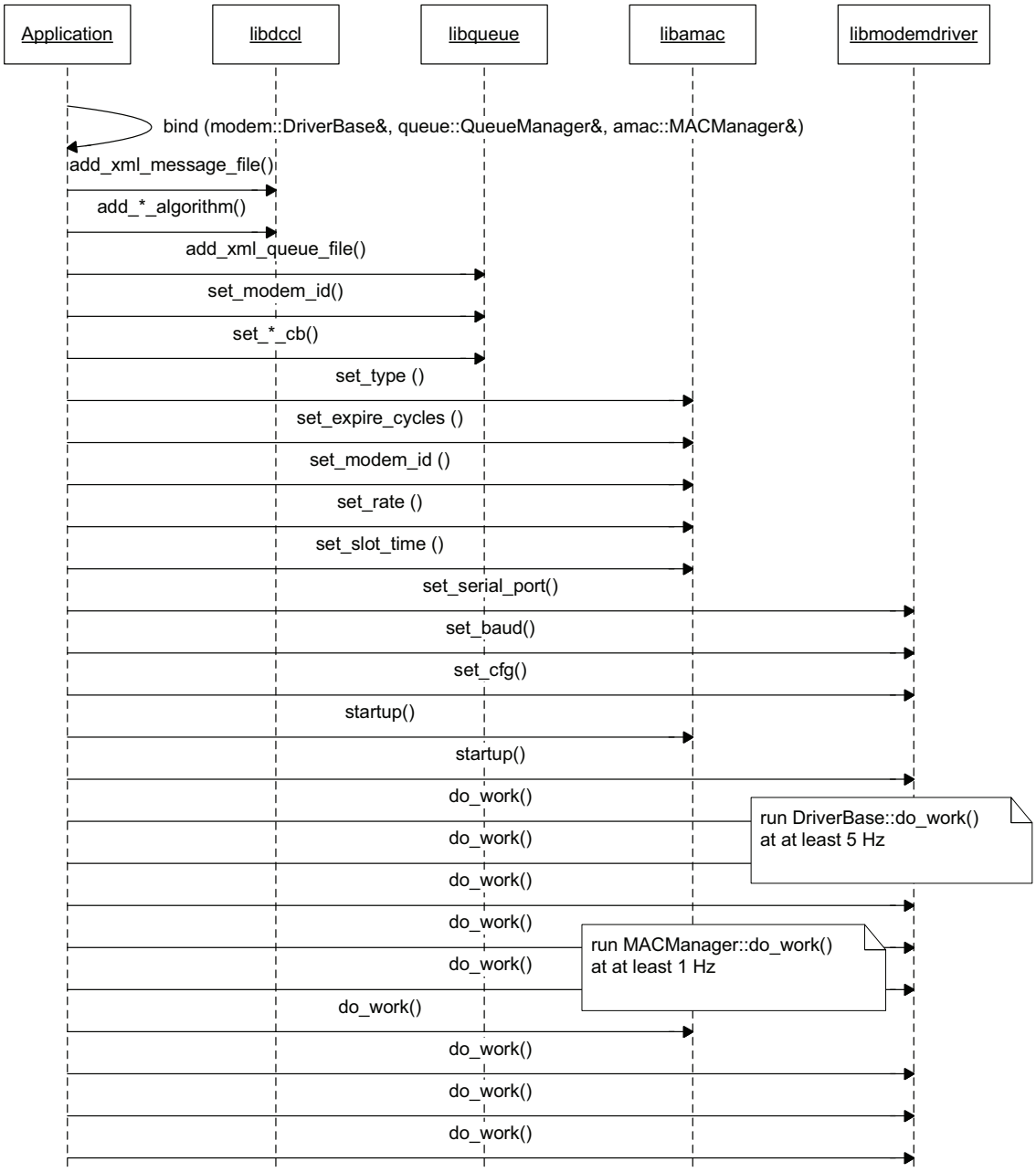

Figure 2.4: UML model that illustrates the set of commands needed to start up goby-acomms and keep it running.

Chapter 3

goby-acomms: libdccl (Dynamic Compact Control Language)

Table of contents for libdccl:

Return to goby-acomms: Overview of Acoustic Communications Libraries.

3.1 Designing a message

Scenario 1: Send a string command to a vehicle:

We need to send an ASCII string command to an underwater vehicle. We thus make the <layout> section of the message contain a single message variable, a <string>. We know that a string uses a byte for each character and DCCL uses six header bytes (CCL id, DCCL id, time, src, dest, flags), making the <max_-length> of our string 26 (32-6). We need some sort of name for our string to use internally when encoding and decoding this message, so we'll use <name> of "s_key" to stand for "string key".

We want to have the ability to use the lowest rate WHOI Micro-Modem message size, so we pick <size> to be 32. We have no other DCCL messages currently in the system so we start with an <id> of 1. Since this is a simple case we choose a "Simple" for our <name>.

See dccl_simple.cpp for the final result (simple.xml) and for an example of how to use this message.

Scenario 2: Send a more realistic command and receive a status message from the vehicle:

We want to be able to command our vehicle (to which we have assigned an ID number of "2") to go to a specific point on a local XY grid (meters from some known latitude / longitude), but no more than 10 kilometers from the datum. We also want to be able to turn the lights on or off, and send a short string for other new instructions. Finally, we need to be able to command a speed. Our vehicle can move no faster

than 3 m/s, but its control is precise enough to handle hundredths of a m/s (wow!). It's probably easiest to make a table with our conditions:

message variable name	description	type	bounds
dest_id	id number of the vehicle we are commanding	integer	built into the header [0, 31]
goto_x	meters east to transit from datum	integer	[0, 10000]
goto_y	meters north to transit from datum	integer	[0, 10000]
lights_on	turn on the lights?	boolean	
new_instructions	string instructions	string	no longer than 10 characters
goto_speed	transit speed (m/s)	float	[0.00, 3.00]

Taking all this into account, we form the <layout> section of the first message (named GoToCommand) in the file two_message.xml (see two_message.cpp).

We choose <id> of 2 to avoid conflicting with the message from Scenario 1 (simple.xml) and a <size> of 32 bytes to again allow sending in the WHOI Micro-Modem rate 0 packet.

Now, for the second message in two_message.xml. We want to receive the vehicle's present position and its current health, which can either be "good", "low_battery" or "abort". We make a similar table to before:

message variable name	description	type	bounds
nav_x	current vehicle position (meters east of the datum)	integer	[0, 10000]
nav_y	current vehicle position (meters north of the datum)	integer	[0, 10000]
health	vehicle state	enumeration	good, low_battery, or abort

The resulting message, along with an example of how to use it, can be seen here: two_message.cpp.

You can run `analyze_dccl_xml` to view more information on your messages:

```
> analyze_dccl_xml /path/to/two_message.xml /path/to/message_schema.xsd
```

The schema (second parameter) is optional but handy for debugging syntax errors. When I ran the above command I got (omitting parts of the header we're not using to save space):

```
creating DCCLCodec using xml file: [examples/two_message/two_message.xml] and schema: [../../message_schem
schema must be specified with an absolute path or a relative path to the xml file location (not pwd!)
parsed file ok!
##############################
detailed message summary:
##############################
********************
message 2: {GoToCommand}
requested size {bytes} [bits]: {32} [256]
actual size {bytes} [bits]: {21} [167]
>>>> HEADER <<<<
destination (int):
size [bits]: [5]
[min, max] = [0,31]
>>>> LAYOUT (message_vars) <<<<
```

```
type (static):
size [bits]: [0]
value: "goto"
goto_x (int):
size [bits]: [14]
[min, max] = [0,10000]
goto_y (int):
size [bits]: [14]
[min, max] = [0,10000]
lights_on (bool):
size [bits]: [2]
new_instructions (string):
size [bits]: [80]
goto_speed (float):
size [bits]: [9]
[min, max] = [0,3]
precision: {2}
********************
********************
message 3: {VehicleStatus}
requested size {bytes} [bits]: {32} [256]
actual size {bytes} [bits]: {11} [84]
>>>> LAYOUT (message_vars) <<<<
nav_x (float):
size [bits]: [17]
[min, max] = [0,10000]
precision: {1}
nav_y (float):
size [bits]: [17]
[min, max] = [0,10000]
precision: {1}
health (enum):
size [bits]: [2]
values:{good,low_battery,abort}
********************
```

Besides validity checking, the most useful feature of `analyze_dccl_xml` is the calculation of the size (in bits) of each message variable. This lets you see which fields in the message are too big. To make fields smaller, tighten up bounds (depending on the type, increase <min>, decrease <max>, decrease <precision>, decrease <max_length>, decrease <num_bytes>, or decrease the number of <enum> <value> options).

3.2 DCCL tag structure

This section gives a brief outline of the tag structure of an XML file for defining a DCCL message. See tags_dccl for a full description of each tag.

DCCL root tags:

- `<?xml version="1.0" encoding="UTF-8"?>`: specifies that the file is XML; must be the first line of every message XML file.
- <message_set>
  - <message>

Children of <message> needed for normal dccl::DCCLCodec::encode and dccl::DCCLCodec::decode:

- <message>

  - <name>
  - <id>
  - <size>
  - <header>
    * <time>
      · <name>
    * <src_id>
      · <name>
    * <dest_id>
      · <name>
  - <layout>
    * <static>
      · <name>
      · <value>
    * <bool>
      · <name>
    * <int>
      · <name>
      · <max>
      · <min>
    * <float>
      · <name>
      · <max>
      · <min>
      · <precision>
    * <string>
      · <name>
      · <max_length>
    * <enum>
      · <name>
      · <value>
    * <hex>
      · <name>
      · <num_bytes>

Children of <message> needed (in addition to those above) for publish/subscribe architecture methods dccl::DCCLCodec::encode_from_src_vars and dccl::DCCLCodec::decode_to_publish (these tags are ignored for calls to dccl::DCCLCodec::encode and dccl::DCCLCodec::decode):

- <message>
  - <header>
    * <src_id>, <dest_id>, or <time>
      · <src_var>
  - <layout>
    * <int>, <hex>, <string>, <float>, <enum>, or <bool>

        - <src_var>
    - <trigger>
    - <trigger_var>
    - <trigger_time>
    - <on_receipt>
        - <publish>
            - <publish_var>
            - <format>
            - <message_var>
            - <all>

3.3 Interacting with the DCCLCodec

Using the dccl::DCCLCodec is a fairly straightforward endeavor. First you need to instantiate a copy of this object with the XML files you want to be able to use and the validating schema (which is optional, but highly recommended especially if editing XML files by hand):

```
dccl::DCCLCodec dccl("/path/to/file.xml", "/path/to/message_schema.xsd");
```

Then, to encode a message, fill up std::maps of dccl::MessageVal where the key of the map (i.e. the `it->first` if `it` is the map iterator) is the <name> of each message variable, and the value (`it->second`) is the quantity you wish to encode. All reasonable type conversions will be made by DCCL using dccl::MessageVal (doubles to <int>, for example), but the most predictable (and fastest) results will be gained by using the following mapping between DCCL message variable types and C++ types:

DCCL Message Variable Type	C++ Type	Example
<int>	long	421
<float>	double	42.1
<hex>	std::string	"abc23" or "ABC23" (case does not matter)
<enum>	std::string	"ON" (case matters, this will not match `<value>on</value>`, but will match `<value>ON</value>`)
<string>	std::string	"i am hungry" (case matters, string cannot contain any null characters in the middle, i.e. '\0')
<bool>	bool	true

After filling up the std::maps, pass pointers to the maps to dccl::DCCLCodec::encode along with a reference to a string in which to store the result:

```
std::map<std::string, dccl::MessageVal> vals;

// code to insert values into map
// ...
//

// store the result here
```

```
std::string hex;

dccl.encode(id, hex, vals);
```

`hex` will now contain the encoded message in the form of a hexadecimal string (capital letters for the alphabetic characters).

You may now send this message through whatever channel you would like, or pass it to the queue::QueueManager to queue for sending later.

To decode a message (stored in `hex` as a hexadecimal string), simply pass hex as a reference along with pointers to the maps to store the results. Based on the maps you provide, values will be cast and stored as the best fit.

```
std::map<std::string, std::string> vals;

dccl.decode(1, hex, vals);
```

For line by line interaction with the dccl::DCCLCodec and for advanced use, investigate the code examples given in the Examples column of this table.

3.4 Encryption

Encryption of all messages can be enabled by providing a secret passphrase to dccl::DCCLCodec::set_crypto_passphrase. All parties to the communication must have the same secret key.

DCCL provides AES (Rijndael) encryption for the body (<layout>) of the message. The header, which is sent in plain text, is hashed to form an initialization vector (IV), and the passphrase is hashed using SHA-256 to form the cipher key.

AES is considered secure and is used for United States top secret information.

3.5 Details of encoding/decoding scheme

We may want to know the actual layout of the binary/hex message. For the first of the two messages in two_message.xml, we can run `analyze_dccl_xml` to find the sizes of each message variable. The calculated sizes are used to determine the boundaries (which are by bit, not by byte) when the message is packed. Each field is placed in the order it is declared in the XML file such that the message is as follows (where left to right is the same as reading the hex string from left to right):

```
[[header][0 {1}][goto_x {14}][goto_y {14}][lights_on {2}][new_instructions {80}][goto_speed {9}]]
```

where `[0 {1}]` means zero fill the message to the closest whole byte (15 bytes = 120 bits minus 119 for other fields = 1). Byte boundaries are dissolved and encoded as a string "ABCDEF..." where the most significant byte (MSB, or leftmost 8 bits) is 0xAB, second MSB is 0xCD, etc. Encoding and decoding are done by functions available in tes_utils.h. You will notice that the resulting size is 15 bytes which is short of the 32 bytes specified for the <size>. This is because the <size> is a maximum size before a warning is generated, not the actual size always returned by the DCCLCodec. This allows libqueue to pack multiple messages together to form a modem message frame and thus fit a nearly optimal amount of data into each modem packet. For example, you could now fit one of the GoToCommand messages *and* another message up to 17 bytes in a single 32 byte WHOI Micro-Modem rate 0 frame.

The encoding of each *message_var* is done as an unsigned integer, with the exception of strings, which are store as ASCII. The value 0 (all bits zero) always indicates "not specified" or "Not a Number" (nan). This

means that the user did not specify any value for this field, specified a value causing overflow (<int> or <float> greater than <max> or less than <min>), or provided a value for an <enum> that did not match any of the enumerate's <value> options. Along with this rule, the method for encoding and decoding is summarized below:

message_var	size (bits)	encode	decode
static	0	not sent	not sent
bool	2	$val_{enc} = (val ==$ to_lower("true") OR val != 0) ? 2 : 1	$val = (val_{enc} == 2)$? true : false
enum	$ceil(log_2(total\_enums + 1))$	val_{enc} = 1 + index to array of enum values based on order they were declared	val = value at index $val_{enc} - 1$
string	$max\_length \cdot 8$	string is filled at end with zeros ('\0') to max_length then encoded using ASCII byte values	ASCII, ignoring null termination chars (if any)
int	$ceil(log_2(max - min + 2))$	$val_{enc} = round(val - min, 0) + 1$	$val = val_{enc} + min - 1$
float	$ceil(log_2((max - min) \cdot 10^{precision} + 2))$	$val_{enc} = round((val - min) \cdot 10^{precision}, 0) + 1$	$val = (val_{enc} - 1)/10^{precision} + min$
hex	$num\_bytes \cdot 8$	$val_{enc} = val$	$val = val_{enc}$

where val is the original (and decoded) value, val_{enc} is the encoded value, min, max, max_length, $precision$ are the contents of the <min>, <max>, <max_length>, and <precision> tags, respectively, and round(x,0) means round x to the nearest integer.

An example. Say you have the following XML file:

```
...
  <id>1</id>
   <header>
   <src_id>
      <name>Src</name>
    </src_id>
    <dest_id>
      <name>Dest</name>
    </dest_id>
  </header>
  <layout>
    <bool>
      <name>B</name>
    </bool>
    <enum>
    <name>E</name>
      <value>cat</value>
      <value>dog</value>
      <value>mouse</value>
    </enum>
    <string>
      <name>S</name>
      <max_length>4</max_length>
    </string>
    <int>
      <name>I</name>
      <max>100</max>
      <min>-50</min>
    </int>
    <float>
```

```
      <name>F</name>
      <max>100</max>
      <min>-50</min>
      <precision>2</precision>
    </float>
  </layout>
...
```

The header is always the same size and is given by (sizes shown in bits):

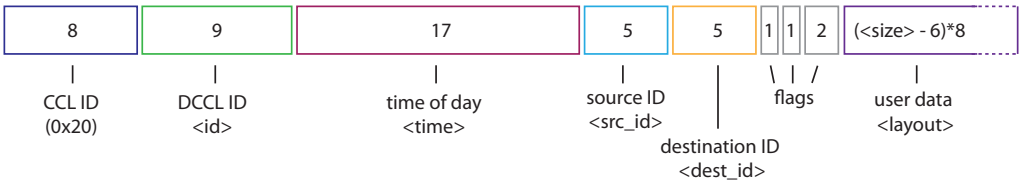

Figure 3.1: Header for all DCCL messages. Sizes shown in bits

Next, the size of each *message_var* (in the <layout> section) and its encoded value in decimal and binary are calculated for an example set of inputs:

message_var	**example** val	**size (bits)**	val_{enc} **(decimal)**	val_{enc} **(binary)**
B	true	2	2	10
E	cat	2	1	01
S	FAT	32	1178686464	01000110 01000001 01010100 00000000
I	34	8	85	01010101
F	-22.49	14	2752	00101011000000

and thus the whole message (zero padded from the most significant bits to the closest byte) sent would be

```
00000010 01010001 10010000 01010101 00000000 00010101 01001010 11000000
```

or

```
0x0251905500154AC0
```

plus the header, which in this case is 0x2000AA300230, so the full message sent is

```
0x2000AA3002300251905500154AC0
```

3.5.1 UML diagrams

The class structure of DCCL:

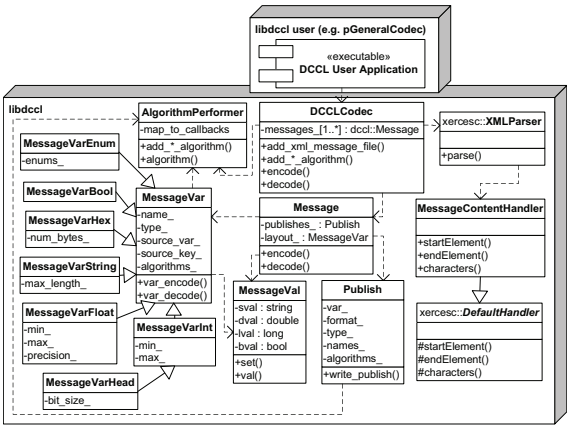

Figure 3.2: Structure diagram of libdccl.

3.6 DCCL XML Tags Reference

3.6.1 <all>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <on_receipt>
      <publish>
        ...
        <all />
      </publish>
    </on_receipt>
  </message>
</message_set>
```

Description: Equivalent to <message_var> for all the *message_var*s in the message. This is a shortcut when you want to publish all the data in a human readable string. [optional, one allowed].

3.6.2 <bool>

```
\b Syntax:
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <layout>
      ...
      <bool algorithm="">
        <src_var></src_var>
        <name></name>
      </bool>
    </layout>
  </message>
</message_set>
```

Description: a boolean (true or false) *message_var* The optional parameter `algorithm` allows you to perform certain algorithms on the data before encoding. See below. [optional, one or more allowed].

3.6.3 <dest_id>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <header>
      <dest_id>
        <name></name>
        <src_var></src_var>
      <dest_id>
    </header>
  </message>
</message_set>
```

Description: Allows setting a name other than the default ("_dest_id") and a <src_var> for the destination id field of the header.

3.6.4 <destination_var>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    ...
    <!-- OUT_MESSAGE: destination=abcd,... -->
    <destination_var key="destination">OUT_MESSAGE</destination_var>
  </message>
</message_set>
```

Description: deprecated. Use <dest_id> instead.

architecture variable to find where this message should be sent. Specify attribute "key=" to specify a substring to look for within the value of this architecture variable. For example, if `COMMAND` contained the string `Destination=3` and you want this message sent to modem_id 3, then you should set `key=Destination` to properly parse that string. [optional: default is 0 (broadcast), one allowed].

3.6.5 <enum>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <layout>
      ...
      <enum algorithm="">
        <src_var></src_var>
        <name></name>
        <value></value>
        <value></value>
        <value></value>
      </enum>
    </layout>
  </message>
</message_set>
```

Description: an enumeration *message_var* [optional, one or more allowed].

3.6.6 <float>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <layout>
      ...
      <float algorithm="">
        <src_var></src_var>
        <name></name>
        <max></max>
        <min></min>
        <precision></precision>
      </float>
    </layout>
  </message>
</message_set>
```

Description: a decimal valued real number *message_var* [optional, one or more allowed].

3.6.7 <format>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <on_receipt>
      <publish>
        ...
        <format>A=%1%,B=%2%</format>
      </publish>
    </on_receipt>
  </message>
</message_set>
```

Description: a string conforming to the format string syntax of the `boost::format library`. This field will specify the format of the string published to the architecture variable defined in <publish_var>. At its simplest, it is a string of incrementing numbers surrounded by %%. Or, instead, you may also use a printf style string, using %d for int *message_var*, %lf for floats, and %s for strings, bools, enums and hex. [optional: default is `name1=%1%,name2=%2%,name3=%3%`, where `name1` is the name of the first <message_var> field to follow, `name2` is the second, etc. exception: default is `%1%` if only a single <message_var> defined. one allowed].

3.6.8 <header>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <header>
      <time></time>
      <src_id></src_id>
      <dest_id></dest_id>
    </header>
    <layout>
      ...
```

```
    </layout>
  </message>
</message_set>
```

Description: holds tags allowing some parts of the DCCL header to be referenced by a new name (other than the defaults: "_time", "_src_id", "_dest_id") at encode and decode time. See <src_id>, <dest_id>, <time>.

3.6.9 <hex>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <layout>
      ...
      <hex algorithm="">
        <src_var></src_var>
        <name></name>
        <num_bytes></num_bytes>
      </hex>
    </layout>
  </message>
</message_set>
```

Description: a message variable represented pre-encoded hexadecimal to add to the message. This field is useful if another source is encoding part or all of a DCCL message. [optional, one or more allowed].

3.6.10 <id>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    ...
    <id>23</id>
  </message>
</message_set>
```

Description: an unsigned six bit integer (0-63) that identifies this message within a network. very similar to the CCL identifier, but for DCCL messages. The CCL identifier occupies the most significant byte (MSB) of the message followed by this id which takes the part of the second MSB (two flags, multimessage and broadcast, use the remainder of the second MSB). *This must be unique within a network*. [mandatory, one allowed]

3.6.11 <int>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <layout>
      ...
      <int algorithm="">
```

```
        <src_var></src_var>
        <name></name>
        <max></max>
        <min></min>
      </int>
    </layout>
  </message>
</message_set>
```

Description: an integer *message_var* [optional, one or more allowed].

3.6.12 <layout>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    ...
    <layout>
      <int></int>
      <string></string>
      <float></float>
      <bool></bool>
      <hex></hex>
      <static></static>
      <enum></enum>
    </layout>
  </message>
</message_set>
```

Description: defines the message structure itself (what fields [the message variables or *message_var*s] the message contains and how they are to be encoded). [mandatory, one allowed].

3.6.13 <max_length>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <layout>
      <string>
        ...
        <max_length>10</max_length>
      </string>
    </layout>
  </message>
</message_set>
```

Description: the length of the string value in this field. Longer strings are truncated. `<max_length>4</max_length>` means "ABCDEFG" is sent as "ABCD". [mandatory, one allowed].

3.6.14 <max>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<message_set>
  <message>
    <layout>
      <int>
        <max>100</max>
      </int>
      <float>
        <max>100</max>
      </float>
    </layout>
  </message>
</message_set>
```

Description: the maximum value this field can take. [mandatory, one allowed].

3.6.15 <message_set>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message></message>
  <message></message>
  <message></message>
</message_set>
```

Description: the root element. All XML files must have a single root element. Since we are define a set of messages (one or more per file), this is a logical choice of name for the root element. [mandatory, one allowed].

3.6.16 <message_var>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <on_receipt>
      <publish>
        ...
        <message_var></message_var>
        <message_var></message_var>
      </publish>
    </on_receipt>
  </message>
</message_set>
```

Description: the name (<name> above) of a *message_var* contained in this message (i.e. an <int>, <bool>, etc.) the values of these fields upon receipt of a message will be used to populate the format string and the result will be published to <publish_var>. The optional parameter `algorithm` allows you to perform certain algorithms on the data after receipt before publishing. See below. [mandatory unless <all> used, one or more allowed].

3.6.17 <message>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <name></name>
    <id></id>
    <size></size>
    <layout></layout>
    <destination_var key=""></destination_var>
    <trigger>publish</trigger>
    <trigger_var mandatory_content=""></trigger_var>
    <!-- OR -->
    <trigger>time</trigger>
    <trigger_time></trigger_time>
    <on_receipt></on_receipt>
    <queuing></queuing>
  </message>
  <message>
    ...
  </message>
</message_set>
```

Description: defines the start of a message. [mandatory, one or more allowed].

3.6.18 <min>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <layout>
      <int>
        <min>-100</min>
      </int>
      <float>
        <min>-100</min>
      </float>
    </layout>
  </message>
</message_set>
```

Description: the minimum value this field can take. [mandatory, one allowed].

3.6.19 <name>

Syntax:

```
<message_set>
  <message>
    ...
    <name>STATUS_REPORT</name>
  </message>
</message_set>
```

or

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <layout>
      <int>
```

```
        <name>parameter1</name>
      </int>
      <string>
        <name>parameter2</name>
      </string>
      <float>
        <name>parameter3</name>
      </float>
      <bool>
        <name>parameter4</name>
      </bool>
      <hex>
        <name>parameter5</name>
      </hex>
      <static>
        <name>parameter6</name>
      </static>
      <enum>
        <name>parameter7</name>
      </enum>
    </layout>
  </message>
</message_set>
```

Description: (as child of <message>): a human readable name for the message. [mandatory, one allowed]

(as child of <int>, <hex>, <string>, <float>, <enum>, or <bool>): the name of this *message_var*. [mandatory, one allowed].

3.6.20 <num_bytes>

the number of bytes for this field. The string provided should be twice as many characters as <num_bytes> since each character of a hexadecimal string is one nibble (4 bits or 1/2 byte). [mandatory, one allowed].

3.6.21 <on_receipt>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Description: contains the various <publish> options for publishing parts of a message upon receipt when using the publish-subscribe architecture method (dccl::DCCLCodec::encode_to_publish). [optional, one allowed].

3.6.22 <precision>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <layout>
      <float>
        <precision>3</precision>
      </float>
    </layout>
  </message>
</message_set>
```

Description: an integer that specifies the number of decimal digits to preserve. Negatives are allowed. For example, `<precision>2</precision>` rounds 1042.1234 to 1042.12; `<precision>-1</precision>` rounds 1042.1234 to 1.04e3. [mandatory, one allowed].

3.6.23 <publish>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <on_receipt>
      <publish>
        <publish_var></publish_var>
        <format></format>
        <message_var></message_var>
      </publish>
    </on_receipt>
  </message>
</message_set>
```

Description: defines a single output value upon receipt of a message. Any number of publishes containing any subset of the *message_var*s can be specified. [mandatory, one or more allowed].

3.6.24 <publish_var>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <on_receipt>
      <publish>
        <publish_var type="string">OUT_STATUS_REPORT</publish_var>
      </publish>
    </on_receipt>
  </message>
</message_set>
```

Description: the name of the architecture variable to publish to. If desired, a format string is allowed here as well (e.g. `%1%_NAV_X` will fill `%1%` with the first *message_var*). See the <format> tag description for more info. [mandatory, one allowed].

3.6.25 <size>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <size>32</size>
  </message>
</message_set>
```

Description: the size of the message in bytes. There are eight bits (binary digits) to a byte. Use N here for messages passed to the Micro-Modem where N is the desired Micro-Modem frame size (N=32, 64, or 256 depending on the rate). If the <layout> of the message exceeds this size, pGeneralCodec will exit on startup with information about sizes, from which you can remove or reduce the size of certain *message_var*s.

3.6.26 <src_id>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <header>
      <src_id>
        <name></name>
        <src_var></src_var>
      <src_id>
    </header>
  </message>
</message_set>
```

Description: Allows setting a name other than the default ("_src_id") and a <src_var> for the source id field of the header.

3.6.27 <src_var>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <layout>
      <!-- OUT_MESSAGE: ...,parameter1=123,parameter2=abc,parameter3=3.42,
                        parameter4=true,parameter5=ON
           SOME_OTHER_HEX: 24bbc231  -->
      <int>
        <src_var key="parameter1">OUT_MESSAGE</src_var>
      </int>
      <string>
        <src_var key="parameter2">OUT_MESSAGE</src_var>
      </string>
      <float>
        <src_var key="parameter3">OUT_MESSAGE</src_var>
      </float>
      <bool>
        <src_var key="parameter4">OUT_MESSAGE</src_var>
      </bool>
      <hex>
        <src_var>SOME_OTHER_HEX</src_var>
      </hex>
      <enum>
        <src_var key="parameter5>OUT_MESSAGE</src_var>
      </enum>
    </layout>
  </message>
</message_set>
```

Description: the architecture variable from which to pull the value of this field. [optional if <trigger>publish</trigger>: default is trigger_var; mandatory if <trigger>time</trigger>, one allowed].

3.6.28 <static>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<message_set>
  <message>
    <layout>
      <static algorithm="">
        <name></name>
        <value></value>
      </static>
    </layout>
  </message>
</message_set>
```

Description: a *message_var* that is not actually sent with the message but can be used to include in received messages (*publishes*). [optional, one or more allowed].

3.6.29 <string>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <layout>
      <string algorithm="">
        <src_var></src_var>
        <name></name>
        <max_length></max_length>
      </string>
    </layout>
  </message>
</message_set>
```

Description: an ASCII string *message_var* [optional, one or more allowed].

3.6.30 <time>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <header>
      <time>
        <name></name>
        <src_var></src_var>
      </time>
    </header>
  </message>
</message_set>
```

Description: Allows setting a name other than the default ("_time") and a <src_var> for the time field of the header. Note that the value of the <src_var> should be a UNIX timestamp (seconds since 1/1/1970 00:00:00).

3.6.31 <trigger_var>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<message_set>
  <message>
    <trigger>publish</trigger>
    <trigger_var mandatory_content="">OUT_MESSAGE</trigger_var>
  </message>
</message_set>
```

Description: used if <trigger>publish</trigger>, this field gives the architecture variable that publishes to will trigger the creation of this message [mandatory if and only if <trigger>publish</trigger>]. optional attribute mandatory_content specifies a string that must be a substring of the contents of the trigger variable in order to trigger the creation of a message. For example, if you wanted to create a certain message every time COMMAND contained the string CommandType=GoTo... but no other time, you would specify mandatory_content="CommandType=GoTo" within this tag.

3.6.32 <trigger_time>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <trigger>time</trigger>
    <trigger_time>60</trigger_time>
  </message>
</message_set>
```

Description: used if <trigger>time</trigger>, this field gives the time interval used to create this message. For example, a value of <trigger_time>10</trigger_time> would mean a message should be created every ten seconds. [mandatory if and only if <trigger>time</trigger>].

3.6.33 <trigger>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <trigger>publish</trigger>
    <trigger_var mandatory_content=""></trigger_var>
    <!-- OR -->
    <trigger>time</trigger>
    <trigger_time></trigger_time>
  </message>
</message_set>
```

Description: how the message is created. Currently this field must take the value "publish" (meaning a message is created on a publish event to a certain architecture variable) or "time" (a message is created on a certain time interval). [mandatory, one allowed]

3.6.34 <value>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
```

```
  <message>
    <layout>
      <static>
        <value>my static value</value>
      </static>
    </layout>
  </message>
</message_set>
```

or

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <layout>
      <enum>
        <value>ON</value>
        <value>OFF</value>
        <value>IN_BETWEEN</value>
      </enum>
    </layout>
  </message>
</message_set>
```

Description: (as child of <static>): the value of this static variable. [mandatory, one allowed].

(as child of <enum>): a possible value (string) the enum can take. Any number of values can be specified. [mandatory, one or more allowed].

Chapter 4

goby-acomms: libqueue (Message Priority Queuing)

Table of Contents for libqueue:

Return to goby-acomms: Overview of Acoustic Communications Libraries.

4.1 Understanding priority queuing

Each queue has a base value (V_{base}) and a time-to-live (ttl) that create the priority ($P(t)$) at any given time (t):

$$P(t) = V_{base} \frac{(t - t_{last})}{ttl}$$

where t_{last} is the time of the last send from this queue.

This means for every queue, the user has control over two variables (V_{base} and ttl). V_{base} is intended to capture how important the message type is in general. Higher base values mean the message is of higher importance. The ttl governs the number of seconds the message lives from creation until it is destroyed by libqueue. The ttl also factors into the priority calculation since all things being equal (same V_{base}), it is preferable to send more time sensitive messages first. So in these two parameters, the user can capture both overall value (i.e. V_{base}) and latency tolerance (ttl) of the message queue.

The following graph illustrates the priority growth over time of three queues with different ttl and V_{base}. A message is sent every 100 seconds and the queue that is chosen is marked on the graph.

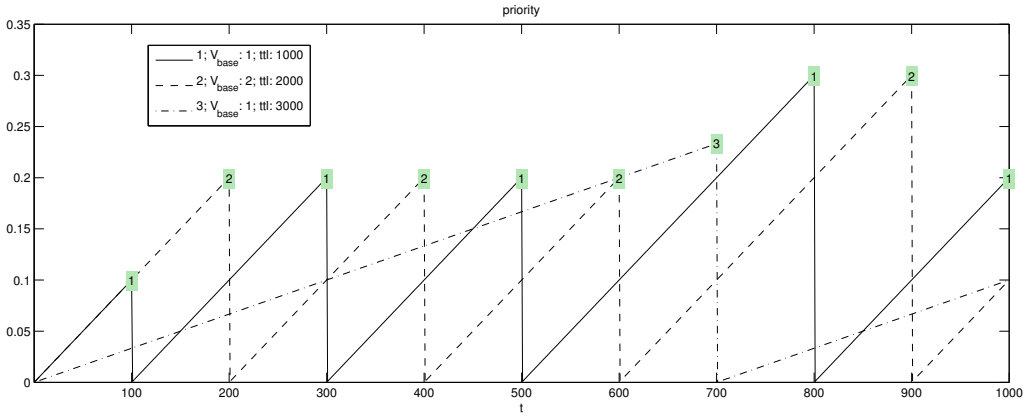

Figure 4.1: Graph of the growth of queueing priorities for libqueue for three different queues. A message is sent every 100 seconds from the %queue with the highest priority (numbered on the graph).

4.2 Queuing tag structure

This section gives a brief outline of the tag structure of an XML file for defining the queuing for a DCCL message. The tags fit in the same file used for DCCL encoding and decoding; see DCCL tag structure for more information.

- `<?xml version="1.0" encoding="UTF-8"?>`: specifies that the file is XML; must be the first line of every message XML file.
- <message_set>
  - <message>
    - <queuing>
      - <ack>
      - <blackout_time>
      - <max_queue>
      - <newest_first>
      - <ttl>
      - <value_base>

4.3 Interacting with the QueueManager

The queue::QueueManager is configured similarly to the dccl::DCCLCodec. You need to add queues to the QueueManager which is done by feeding it either XML files (for DCCL queues with queue::QueueManager::add_xml_queue_file) or queue::QueueConfig objects (for CCL queues with queue::QueueManager::add_queue). You can instantiate the queue::QueueManager (optionally with XML files) like

```
queue::QueueManager q_manager("/path/to/file.xml","/path/to/message_schema.xsd");
```

Then, you need to do a few more initialization chores:

- Set the modem id for the current vehicle: queue::QueueManager::set_modem_id
- Pass function pointers or function objects for these callbacks (any of these are optional if you do not need their functionality):
  - Received DCCL data: queue::QueueManager::set_receive_cb
  - Received CCL data: queue::QueueManager::set_receive_ccl_cb
  - Received acknowledgements: queue::QueueManager::set_ack_cb
- Optional advanced features
  - Set a callback for every time a queue size changes due to a new message being pushed or a message being sent: queue::QueueManager::set_queue_size_change_cb
  - Request that a queue be *on_demand*, that is, request data from the application layer every time the modem layer requests data. This effectively bypasses the queue and forwards the modem's data request to the application layer. Use this for sending highly time sensitive data which needs to be encoded immediately prior to sending: queue::QueueManager::set_on_demand. You must also provide a function callback that will be executed each time data is request: queue::QueueManager::set_data_on_demand_cb.

At this point the queue::QueueManager is ready to use. At the application layer, new messages are pushed to the queues for sending using queue::QueueManager::push_message. Each queue is identified by a unique queue::QueueKey, which is simply the identification number of the queue (<id> for DCCL queues or the decimal representation of the first byte of a CCL message) and the queue type (queue::queue_dccl or queue::queue_ccl).

At the driver layer, messages are requested using queue::QueueManager::provide_outgoing_modem_data, incoming messages are published using queue::QueueManager::receive_incoming_modem_data, and queue::QueueManager::acknowledgements are given using queue::QueueManager::handle_modem_ack. If using the goby-acomms drivers (i.e. some class derived from modem::DriverBase), simply call acomms_util::bind(modem::DriverBase&, queue::QueueManager&) and these methods will be invoked automatically from the proper driver callbacks.

4.4 Queuing XML Tags Reference

4.4.1 <ack>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <queuing>
      ...
      <ack>true</ack>
    </queuing>
  </message>
</message_set>
```

Description: boolean flag (1=true, 0=false) whether to request an acoustic acknowledgment on all sent messages from this field. If omitted, default of 0 (false, no ack) is used.

4.4.2 <blackout_time>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <queuing>
      ...
      <blackout_time>0</blackout_time>
    </queuing>
  </message>
</message_set>
```

Description: time in seconds after sending a message from this queue for which no more messages will be sent. Use this field to stop an always full queue from hogging the channel. If omitted, default of 0 (no blackout) is used.

4.4.3 <max_queue>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <queuing>
      ...
      <max_queue>1</max_queue>
    </queuing>
  </message>
</message_set>
```

Description: number of messages allowed in the queue before discarding messages. If <newest_first> is set to true, the oldest message in the queue is discarded to make room for the new message. Otherwise, any new messages are disregarded until the space in the queue opens up.

4.4.4 <newest_first>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <queuing>
      ...
      <newest_first>true</newest_first>
    </queuing>
  </message>
</message_set>
```

Description: boolean flag (1=true=FILO, 0=false=FIFO) whether to send newest messages in the queue first (FILO) or not (FIFO).

4.4.5 <value_base>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <queuing>
      ...
      <value_base>10</value_base>
    </queuing>
  </message>
</message_set>
```

Description: base priority value for this message queue. priorities are calculated on a request for data by the modem (to send a message). The queue with the highest priority (and isn't in blackout) is chosen. The actual priority (P) is calculated by P(t) = $V_{base}\frac{(t-t_{last})}{ttl}$ where V_{base} is the value set here, t is the current time (in seconds), t_{last} is the time of the last send from this queue, and ttl is the <ttl>. Essentially, a message with low <ttl> will become effective quickly again after a sent message (the priority line grows faster). See Understanding priority queuing for further discussion.

4.4.6 <ttl>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <queuing>
      ...
      <ttl>120</ttl>
    </queuing>
  </message>
</message_set>
```

Description: the time in seconds a message lives after its creation before being discarded. This time-to-live also factors into the growth in priority of a queue. see <value_base> for the main discussion on this. 0 is a special value indicating no infinite live (i.e. <ttl>0</ttl> is effectively the same as <ttl>∞</ttl>).

4.4.7 <queuing>

Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <queuing>
      <ack></ack>
      <blackout_time></blackout_time>
      <max_queue></max_queue>
      <newest_first></newest_first>
      <priority_base></priority_base>
      <priority_time_const></priority_time_const>
    </queuing>
  </message>
</message_set>
```

Description: Represents the XML embodiment of the queue::QueueConfig.

Chapter 5

goby-acomms: libmodemdriver (Driver to interact with modem firmware)

Table of contents for libmodemdriver:

Return to goby-acomms: Overview of Acoustic Communications Libraries.

5.1 Abstract class: DriverBase

modem::DriverBase defines the core functionality for an acoustic modem. It provides

- a serial class (and methods to configure it) that reads the serial port data into a buffer for use by the modem::DriverBase derived class (e.g. micromodem::MMDriver). These methods are modem::DriverBase::set_serial_port and modem::DriverBase::set_baud.
- methods to set all six callbacks provided by the derived class (receive, acknowledgement, data request, parsed incoming message, raw incoming message, raw outgoing message). At the application layer, either bind the modem driver to a queue::QueueManager (acomms_util::bind(modem::DriverBase&, queue::QueueManager&) or pass custom function pointers or objects to the driver layer callbacks: modem::DriverBase::set_receive_cb, modem::DriverBase::set_ack_cb, modem::DriverBase::set_datarequest_cb. The parsed incoming message callback is used by the amac::MACManager to "discover" vehicles so this callback should be bound to the amac::MACManager (using acomms_util::bind(amac::MACManager&, modem::DriverBase&)). The remaining two callbacks (raw incoming and raw outgoing) are for use by the application layer if desired to monitor the functionality of the modem.
- three virtual functions: for starting the driver (modem::DriverBase::startup), running the driver (modem::DriverBase::do_work), and initiating the transmission of a message (modem::DriverBase::initiate_transmission).
- a method to set configuration values for the acoustic modem (modem::DriverBase::set_cfg). this configuration takes the form of a vector of strings, the details of which depend on the specific modem.

5.2 WHOI Micro-Modem Driver: MMDriver

The micromodem::MMDriver extends the DriverBase for the WHOI Micro-Modem acoustic modem. It is tested to work with revision 0.93.0.30 of the Micro-Modem firmware, but is known to work with older firmware (at least 0.92.0.85). The following commands of the WHOI Micro-Modem are implemented:

Modem to Control Computer ($CA):

- $CAACK - acknowledgement of sent message. Will be transformed into a modem::Message and passed to the callback provided to modem::DriverBase::set_ack_cb.
- $CADRQ - data request. Will be transformed into a modem::Message and passed to the callback provided to modem::DriverBase::set_datarequest_cb. The second parameter of that callback holds a reference to a modem::Message into which the callback should place the requested data.
- $CARXD - received hexadecimal data. Will be transformed into a modem::Message and passed to the callback provided to modem::DriverBase::set_receive_cb.
- $CAREV - revision number and heartbeat. Used to check for correct clock time and modem reboots.
- $CAERR - error message. The error message is logged to the std::ostream provided to micromodem::MMDriver at instantiation.

Control Computer to Modem ($CC). Also implemented is the NMEA acknowledge (e.g. $CACYC for $CCCYC):

- $CCTXD - transmit data. Sent using the second parameter of the callback provided to modem::DriverBase::set_datarequest_cb (see $CADRQ).
- $CCCYC - initiate a cycle. Sent on response to a call of micromodem::MMDriver::initiate_transmission.
- $CCCLK - set the clock. The clock is set on startup until a suitably value within 1 second of the computer time is reported back. If the modem reboots ($CAREV,...,INIT), the clock is set again.
- $CCCFG - configure NVRAM value. All values passed to modem::DriverBase::set_cfg will be passed to $CACFG at startup. For example, to send $CACFG,SRC,3, place the string "SRC,3" in the vector passed to modem::DriverBase::set_cfg.
- $CCCFQ - query configuration values. $CCCFQ,ALL is sent after all the $CCCFG lines to log the NVRAM parameters.

Mapping between modem::Message and NMEA fields (see http://acomms.whoi.edu/documents/uModem%20Soft
for NMEA fields of the WHOI Micro-Modem):

NMEA talker	Mapping
$CAACK	modem::Message::src = SRC modem::Message::dest = DEST modem::Message::frame = Frame# modem::Message::ack = A
$CADRQ	modem::Message::src = SRC modem::Message::dest = DEST modem::Message::ack = ACK modem::Message::size = N modem::Message::frame = F#
$CARXD	modem::Message::src = SRC modem::Message::dest = DEST modem::Message::ack = ACK modem::Message::frame = F# modem::Message::data = HH...HH modem::Message::size = number of bytes in HH...HH
$CAREV	not translated into a modem::Message.
$CAERR	not translated into a modem::Message.
$CCTXD	SRC = modem::Message::src DEST = modem::Message::dest A = modem::Message::ack HH...HH = modem::Message::data
$CCCYC	CMD = 0 ADR1 = modem::Message::src ADR2 = modem::Message::dest Packet Type = modem::Message::rate ACK = modem::Message::ack Nframes = modem::Message::frame
$CCCLK	not translated from a modem::Message (taken from the system time using the boost::date_time library)
$CCCFG	not translated from a modem::Message. (taken from the std::vector<std::string> passed to modem::DriverBase::set_cfg)
$CCCFQ	not translated from a modem::Message. $CCCFG,ALL sent at startup.

5.3 Writing a new driver

Most of goby-acomms is intended to be agnostic of which physical modem is used. However, since the whole system was designed for the WHOI Micro-Modem initially, it will probably take some work to initially incorporate a different device.

These are the assumptions of the acoustic modem:

- it communicates using a line based duplex serial connection.

- it is configurable using string values.

The new driver must provide this functionality:

- Overload the three pure virtual functions:
  1. modem::DriverBase::startup: do any initialization here you need to do before the modem is ready to do work. Probably the place to read and set the values in std::vector<std::string> cfg_.
  2. modem::DriverBase::do_work: this is called periodically (say at 10 Hz) by the application layer. You should read all the messages in the serial buffer by calling modem::DriverBase::serial_read until it returns false indicating no more lines to read.
  3. modem::DriverBase::initiate_transmission: this is called when the application wants to send a message. The modem::Message provided here *does not* contain the data to be sent. To get data, you must call modem::DriverBase::callback_datarequest where the first parameter is the modem::Message of the request and the second parameter is a reference to a modem::Message that will contain the data to be sent. The modem::Message provided to initiate_transmission may request multiple frames (modem::Message::frame). You can, *but are not required to*, call as many instances of modem::DriverBase::callback_datarequest as frames requested. It will be important to map the hardware packet sizes to the sizes requested in modem::Message::size of the first parameter of modem::DriverBase::callback_datarequest. This will obviously depend on the specifics of the modem and the data rate requested. For the WHOI Micro-Modem, for example, there are six possible data rates, four of which are implemented, each with a different frame size and number of frames (rate 0 = 1x 32 byte frame, rate 2 = 3x 64 byte frames, rate 3 = 2x 256 byte frames, rate 5 = 8x 256 byte frames). For example, if your modem uses a single 512 byte frame, call modem::DriverBase::callback_datarequest(modem::Message("size=512")). On the other hand, if it uses 3x 128 byte frames, call modem::DriverBase::callback_datarequest(modem::Message("size=128")) three different times, one for each frame. If your modem has variable length packets, pick a reasonable size for which the duration of the packet in the water is on the order of several seconds.
- Call callback_decoded with a modem::Message mapped for each incoming acoustic sourced message (acknowledgement, data packet, control message, anything). Ideally should contain at least a modem::Message::src and modem::Message::dest, and intelligent mappings to the other fields of modem::Message where possible or for which the probability of reception is acceptably high.
- Call callback_ack with the a modem::Message containing modem::Message::src, modem::Message::dest, and modem::Message::frame for each frame acknowledged. You should acknowledge any data that is to be acknowledged before requesting the next set of frames. That is, do not leave a frame 1 unacknowledged (assuming <ack> is set true for that queue) and request another frame 1.
- Call callback_receive with any received data. This should include at least a modem::Message::src, modem::Message::dest, and modem::Message::data.

Modem bit rates need to be mapped onto a set of unsigned integers from 0 to 5 where 0 is the lowest (and thus presumably most robust) bit rate and 5 is the highest. All vehicles in the network must be identified by unsigned integers (in the range 0-stdnumeric_limits<unsigned>::max).

See micromodem::MMDriver for an example implementation.

Chapter 6

goby-acomms: libamac (Medium Access Control)

Table of Contents for libamac:

Return to goby-acomms: Overview of Acoustic Communications Libraries.

6.1 Supported MAC schemes

The Medium Access Control schemes provided by libamac are based on Time Division Multiple Access (TDMA) where different communicators share the same bandwidth but transmit at different times to avoid conflicts. Time is divided into slots and each vehicle is given a slot to transmit on. The set of slots comprising all the vehicles is referred to here as a cycle, which repeats itself when it reaches the end. The two variations on this scheme provided by libamac are:

1. Slotted TDMA (amac::mac_slotted_tdma): Each vehicle has a single slot in the cycle on which it transmits. Each vehicle initiates its own transmission at the start of its slot. Collisions are avoided by each vehicle following the same rules about slot placement within the time window (based on real time of day). This scheme requires that each vehicle have reasonably accurate clocks (perhaps better than +/- 0.5 seconds).
2. Centralized Polling (amac::mac_polled): The TDMA cycle is set up and operated by a centralized modem ("poller"), which is usually the modem connected to the vehicle operator's topside. The poller initiates each transmission and thus the vehicles are not required to maintain synchronous clocks.

6.2 Interacting with the amac::MACManager

To use the amac::MACManager, you need to instantiate it (optionally with a std::ostream pointer to a location to log to):

```
amac::MACManager mac(&std::cout);
```

Then you need to provide a callback for providing the next destination of a message (amac::MACManager::set_destination_cb) or simply bind the amac::MACManager to the queue::QueueManager, if you are using one (acomms_util::bind(amac::MACManager&, queue::QueueManager&)). This callback takes as an argument the size (in bytes) of the next message to be sent and should return the modem id number of the next destination (or -1 if no data is to be sent). The other callback to set is for initiating transmission (amac::MACManager::set_initiate_transmission_cb). This callback will be called when the amac::MACManager determines it is time to send a message. If using libmodemdriver, simply call acomms_util::bind(amac::MACManager&, modem::DriverBase&) to bind this callback to the modem driver.

Next you need to decide which type of MAC to use: slotted TDMA or centralized polling and set the type of the amac::MACManager with the corresponding amac::MACType:

```
mac.set_type(amac::mac_slotted_tdma);
```

The usage of the amac::MACManager depends now on the type:

- amac::mac_slotted_tdma: Set the rest of the parameters (modem rate (integer from 0-5), slot time (seconds), cycles before removing a "dead" vehicle from the cycle, and the modem id of this vehicle):

  ```
  mac.set_rate(0);
  mac.set_slot_time(10);
  mac.set_expire_cycles(2);
  mac.set_modem_id(1);
  ```

- amac::mac_polled: On the vehicles, you do not need to run the amac::MACManager at all, or simply leave it unconfigured. All the MAC is done on the topside (the centralized poller). On the poller, you need to manually set up a list of vehicles to be polled by adding an amac::Slot (amac::MACManager::add_slot) for each vehicle to be polled. You can poll the same vehicle multiple times, just add more amac::Slot objects corresponding to that vehicle. Each slot has a source, destination, rate, type (data or ping [not yet implemented]), and length (in seconds). If the source is the poller, you can set the destination to -1 to make the amac::MACManager call the callback given to amac::MACManager::set_destination_cb to determine the destination. All amac::Slot objects for vehicles must have a specified destination (the broadcast id of 0 is a good choice). For example:

  ```
  // let's call our poller id 1
  mac.set_modem_id(1);

  // poll ourselves (for commands, perhaps)
  // src: 1, dest: ask (-1), rate 0, send data, 10 second slot
  mac.add_slot(amac::Slot(1, -1, 0, amac::slot_data, 10));
  // poll vehicle 3 at rate 0
  mac.add_slot(amac::Slot(3, acomms_util::BROADCAST_ID, 0, amac::slot_data, 10));
  // poll vehicle 3 at rate 5
  mac.add_slot(amac::Slot(3, acomms_util::BROADCAST_ID, 5, amac::slot_data, 10));
  // poll vehicle 4 at rate 0
  mac.add_slot(amac::Slot(4, acomms_util::BROADCAST_ID, 0, amac::slot_data, 10));
  ```

  You can remove vehicles by a call to amac::MACManager::remove_slot or clear out the entire cycle and start over with amac::MACManager::clear_all_slots.

Then, for either MAC scheme, start the amac::MACManager running (amac::MACManager::startup), and call amac::MACManager::do_work() periodically (1 Hz is fine). If using a derivative of modem::DriverBase, vehicles will automatically be discovered as they are heard from and added to the cycle (via a callback to amac::MACManager::process_message). If using a different driver, you need to inform the amac::MACManager each time you hear from a vehicle by passing a modem::Message representing any transmission (doesn't need to contain data) to amac::MACManager::process_message. If a vehicle isn't heard from for a certain number of cycles (set by amac::MACManager::set_expire_cycles), it will be removed from the cycle to increase throughput for the remaining vehicles.

Chapter 7

goby-util: Overview of Utility Libraries

7.1 Overview

forthcoming...

7.2 Logging

forthcoming...

7.2.1 C++ STL Streams

forthcoming...

7.2.2 Configurable extension of std::ostream - libflexostream

forthcoming...

7.3 Serial port communications - libserial

Appendix A

Namespace Index

A.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Appendix B

Class Index

B.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Appendix C

Class Index

C.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Appendix D

Namespace Documentation

D.1 acomms_util Namespace Reference

utilites for dealing with goby-acomms

Functions

- void bind (modem::DriverBase &driver, queue::QueueManager &queue_manager)

  binds the driver link-layer callbacks to the QueueManager

- void bind (amac::MACManager &mac, modem::DriverBase &driver)

  binds the MAC initiate transmission callback to the driver and the driver parsed message callback to the MAC

- void bind (amac::MACManager &mac, queue::QueueManager &queue_manager)

  binds the MAC destination request to the queue_manager

- void bind (modem::DriverBase &driver, queue::QueueManager &queue_manager, amac::MACManager &mac)

  bind all three (shortcut to calling the other three bind functions)

D.1.1 Detailed Description

utilites for dealing with goby-acomms

D.2 amac Namespace Reference

contains the medium access control objects

Classes

- class Slot

Represents a slot of the TDMA cycle.

- class MACManager

  provides an API to the goby-acomms MAC library.

Typedefs

Acoustic MAC Library callback function type definitions

- typedef boost::function< int(unsigned)> IdFunc

  boost::function for a function taking a unsigned and returning an integer.

- typedef boost::function< void(const modem::Message &message)> MsgFunc1

  boost::function for a function taking a single modem::Message reference.

Enumerations

- enum MACType { mac_notype, mac_slotted_tdma, mac_polled }

  Enumeration of MAC types.

Functions

- bool operator== (const Slot &a, const Slot &b)

  Are two amac::Slot equal?

D.2.1 Detailed Description

contains the medium access control objects Use

```
#include <goby/acomms/amac.h>
```

to gain access to all these objects.

D.2.2 Typedef Documentation

D.2.2.1 typedef boost::function<int (unsigned)> amac::IdFunc

boost::function for a function taking a unsigned and returning an integer.

Think of this as a generalized version of a function pointer (int (*) (unsigned)). See http://www.boost.org/doc/libs/1_34_0/doc/html/function.html for more on boost:function.

Definition at line 43 of file mac_manager.h.

D.2.2.2 typedef boost::function<void (const modem::Message & message)> amac::MsgFunc1

boost::function for a function taking a single modem::Message reference.

Think of this as a generalized version of a function pointer (void (∗)(modem::Message&)). See http://www.boost.org/doc/libs/1_34_0/doc/html/function.html for more on boost:function.

Definition at line 48 of file mac_manager.h.

D.2.3 Enumeration Type Documentation

D.2.3.1 enum amac::MACType

Enumeration of MAC types.

Enumerator:

mac_notype no MAC

mac_slotted_tdma decentralized time division multiple access

mac_polled centralized polling

Definition at line 53 of file mac_manager.h.

D.3 dccl Namespace Reference

contains Dynamic Compact Control Language objects.

Classes

- class DCCLCodec

  provides an API to the Dynamic CCL Codec.

- class MessageVal

  defines a DCCL value

Typedefs

- typedef boost::function< void(MessageVal &)> AlgFunction1

  boost::function for a function taking a single dccl::MessageVal reference. Used for algorithm callbacks.

- typedef boost::function< void(MessageVal &, const std::vector< MessageVal > &)> AlgFunction2

  boost::function for a function taking a dccl::MessageVal reference, and the MessageVal of a second part of the message. Used for algorithm callbacks.

Enumerations

- enum DCCLType {
  dccl_static, dccl_bool, dccl_int, dccl_float,
  dccl_enum, dccl_string, dccl_hex }
  Enumeration of DCCL types used for sending messages. dccl_enum and dccl_string primarily map to cpp_string, dccl_bool to cpp_bool, dccl_int to cpp_long, dccl_float to cpp_double.

- enum DCCLCppType {
  cpp_notype, cpp_bool, cpp_string, cpp_long,
  cpp_double }
  Enumeration of C++ types used in DCCL.

Functions

- template<typename Value >
  std::ostream & operator<< (std::ostream &out, const std::map< std::string, Value > &m)
  use this for displaying a human readable version

- std::ostream & operator<< (std::ostream &out, const std::set< unsigned > &s)
  use this for displaying a human readable version of this STL object

- std::ostream & operator<< (std::ostream &out, const std::set< std::string > &s)
  use this for displaying a human readable version of this STL object

- std::ostream & operator<< (std::ostream &out, const DCCLCodec &d)
  outputs information about all available messages (same as std::string summary())

D.3.1 Detailed Description

contains Dynamic Compact Control Language objects. Use

```
#include <goby/acomms/dccl.h>
```

to gain access to all these objects.

D.3.2 Typedef Documentation

D.3.2.1 typedef boost::function<void (MessageVal&)> dccl::AlgFunction1

boost::function for a function taking a single dccl::MessageVal reference. Used for algorithm callbacks.

Think of this as a generalized version of a function pointer (void (*)(dccl::MessageVal&)). See http://www.boost.org/doc/libs/1_34_0/doc/html/function.html for more on boost:function.

Definition at line 34 of file message_algorithms.h.

D.3.2.2 typedef boost::function<void (MessageVal&, const std::vector<MessageVal>&)> dccl::AlgFunction2

boost::function for a function taking a dccl::MessageVal reference, and the MessageVal of a second part of the message. Used for algorithm callbacks.

Think of this as a generalized version of a function pointer (void (*)(MessageVal&, const MessageVal&). See http://www.boost.org/doc/libs/1_34_0/doc/html/function.html for more on boost:function.

Definition at line 43 of file message_algorithms.h.

D.3.3 Enumeration Type Documentation

D.3.3.1 enum dccl::DCCLCppType

Enumeration of C++ types used in DCCL.

Enumerator:

cpp_notype not one of the C++ types used in DCCL

cpp_bool C++ bool

cpp_string C++ std::string

cpp_long C++ long

cpp_double C++ double

Definition at line 46 of file dccl_constants.h.

D.3.3.2 enum dccl::DCCLType

Enumeration of DCCL types used for sending messages. dccl_enum and dccl_string primarily map to cpp_string, dccl_bool to cpp_bool, dccl_int to cpp_long, dccl_float to cpp_double.

Enumerator:

dccl_static <static>

dccl_bool <bool>

dccl_int <int>

dccl_float <float>

dccl_enum <enum>

dccl_string <string>

dccl_hex <hex>

Definition at line 37 of file dccl_constants.h.

D.4 micromodem Namespace Reference

contains WHOI Micro-Modem specific objects.

Classes

- class MMDriver

  provides an API to the WHOI Micro-Modem driver

D.4.1 Detailed Description

contains WHOI Micro-Modem specific objects.

D.5 modem Namespace Reference

Acoustic Modem specific objects.

Classes

- class DriverBase

  provides a base class for acoustic modem drivers (i.e. for different manufacturer modems) to derive

- class Message

  represents a message to or from the acoustic modem.

Typedefs

Driver Library callback function type definitions

- typedef boost::function< void(const modem::Message &message)> MsgFunc1

  boost::function for a function taking a single modem::Message reference.

- typedef boost::function< bool(const modem::Message &message1, modem::Message &message2)> MsgFunc2

  boost::function for a function taking a modem::Message reference as input and filling a modem::Message reference as output.

- typedef boost::function< void(const std::string &s)> StrFunc1

  boost::function for a function passed a string.

D.5.1 Detailed Description

Acoustic Modem specific objects.

D.5.2 Typedef Documentation

D.5.2.1 typedef boost::function<void (const modem::Message& message)> modem::MsgFunc1

boost::function for a function taking a single modem::Message reference.

Think of this as a generalized version of a function pointer (bool (∗)(const modem::Message&)). See http://www.boost.org/doc/libs/1_34_0/doc/html/function.html for more on boost:function.

Definition at line 37 of file driver_base.h.

D.5.2.2 typedef boost::function<bool (const modem::Message& message1, modem::Message& message2)> modem::MsgFunc2

boost::function for a function taking a modem::Message reference as input and filling a modem::Message reference as output.

Think of this as a generalized version of a function pointer (bool (∗)(const modem::Message&, modem::Message&)). See http://www.boost.org/doc/libs/1_34_0/doc/html/function.html for more on boost:function.

Definition at line 42 of file driver_base.h.

D.5.2.3 typedef boost::function<void (const std::string& s)> modem::StrFunc1

boost::function for a function passed a string.

Think of this as a generalized version of a function pointer (void (∗)(const std::string&)). See http://www.boost.org/doc/libs/1_34_0/doc/html/function.html for more on boost:function.

Definition at line 47 of file driver_base.h.

D.6 queue Namespace Reference

contains the message priority queuing objects.

Classes

- class QueueConfig

  defines parameters for configuring a message queue

- class QueueKey

  forms a unique key for a given message queue

- class QueueManager

  provides an API to the goby-acomms Queuing Library.

Typedefs

Queue Library callback function type definitions

- typedef boost::function< void(QueueKey key, const modem::Message &message)> MsgFunc1

  boost::function for a function taking a single modem::Message reference.

- typedef boost::function< bool(QueueKey key, const modem::Message &message1, modem::Message &message2)> MsgFunc2

  boost::function for a function taking a modem::Message reference as input and filling a modem::Message reference as output.

- typedef boost::function< void(QueueKey key, unsigned size)> QSizeFunc

  boost::function for a function returning a single unsigned

Enumerations

- enum QueueType { queue_notype, queue_dccl, queue_ccl, queue_data }

  specifies the type of the Queue

Functions

- std::ostream & operator<< (std::ostream &out, const QueueType &qt)

  human readable output of the QueueType

- std::ostream & operator<< (std::ostream &out, const QueueManager &d)

  outputs information about all available messages (same as std::string summary())

D.6.1 Detailed Description

contains the message priority queuing objects. Use

```
#include <goby/acomms/queue.h>
```

to gain access to all these objects.

D.6.2 Typedef Documentation

D.6.2.1 typedef boost::function<void (QueueKey key, const modem::Message& message)> queue::MsgFunc1

boost::function for a function taking a single modem::Message reference.

Think of this as a generalized version of a function pointer (void (∗)(QueueKey, modem::Message&)). See http://www.boost.org/doc/libs/1_34_0/doc/html/function.html for more on boost:function.

Definition at line 45 of file queue_manager.h.

D.6.2.2 typedef boost::function<bool (QueueKey key, const modem::Message& message1, modem::Message & message2)> queue::MsgFunc2

boost::function for a function taking a modem::Message reference as input and filling a modem::Message reference as output.

Think of this as a generalized version of a function pointer (void (∗)(QueueKey, modem::Message&, modem::Message&)). See http://www.boost.org/doc/libs/1_34_-0/doc/html/function.html for more on boost:function.

Definition at line 49 of file queue_manager.h.

D.6.2.3 typedef boost::function<void (QueueKey key, unsigned size)> queue::QSizeFunc

boost::function for a function returning a single unsigned

Think of this as a generalized version of a function pointer (void (∗)(QueueKey, unsigned)). See http://www.boost.org/doc/libs/1_34_0/doc/html/function.html for more on boost:function.

Definition at line 53 of file queue_manager.h.

D.6.3 Enumeration Type Documentation

D.6.3.1 enum queue::QueueType

specifies the type of the Queue

Enumerator:

queue_notype no type (queue is useless)

queue_dccl queue holds DCCL messages

queue_ccl queue holds CCL messages

queue_data queue holds miscellaneous data

Definition at line 32 of file queue_config.h.

Appendix E

Class Documentation

E.1 ChatCurses Class Reference

provides a terminal GUI for a chat window (lower box to type and upper box to receive messages)

`#include <chat_curses.h>`

Public Member Functions

- void set_modem_id (unsigned id)
  give the modem_id so we know how to label our messages
- void startup ()
  start the display
- void run_input (std::string &line)
  grab a character and if there's a line to return it will be returned in `line`
- void cleanup ()
  end the display
- void post_message (unsigned id, const std::string &line)
  add a message to the upper window (the chat log)

Constructors/Destructor

- **ChatCurses** ()
- ~**ChatCurses** ()

E.1.1 Detailed Description

provides a terminal GUI for a chat window (lower box to type and upper box to receive messages)

Examples:

acomms/examples/chat/chat.cpp.

Definition at line 23 of file chat_curses.h.

The documentation for this class was generated from the following files:

- /var/www/root/goby-bzr/src/acomms/examples/chat/chat_curses.h
- /var/www/root/goby-bzr/src/acomms/examples/chat/chat_curses.cpp

E.2 dccl::DCCLCodec Class Reference

provides an API to the Dynamic CCL Codec.

`#include <dccl.h>`

Public Member Functions

Constructors/Destructor

- DCCLCodec ()
  Instantiate with no XML files.
- DCCLCodec (const std::string &file, const std::string schema="")
  Instantiate with a single XML file.
- DCCLCodec (const std::set< std::string > &files, const std::string schema="")
  Instantiate with a set of XML files.
- ~DCCLCodec ()
  destructor

Initialization Methods.

These methods are intended to be called before doing any work with the class. However, they may be called at any time as desired.

- std::set< unsigned > add_xml_message_file (const std::string &xml_file, const std::string xml_schema="")
  Add more messages to this instance of the codec.
- void set_schema (const std::string &schema)
  Set the schema used for xml syntax checking.
- void set_crypto_passphrase (const std::string &passphrase)
  Set a passphrase for encrypting all messages with.
- void set_modem_id (unsigned modem_id)
  Set the modem id for this vehicle.
- void add_algorithm (const std::string &name, AlgFunction1 func)
  Add an algorithm callback for a dccl::MessageVal. The return value is stored back into the input parameter (dccl::MessageVal). See test.cpp for an example.
- void add_adv_algorithm (const std::string &name, AlgFunction2 func)

Add an advanced algorithm callback for any DCCL C++ type that may also require knowledge of all the other message variables and can optionally have additional parameters.

Codec functions.

This is where the real work happens.

- template<typename Key >
  void encode (const Key &k, std::string &hex, const std::map< std::string, MessageVal > &m)
  Encode a message.

- template<typename Key >
  void encode (const Key &k, std::string &hex, const std::map< std::string, std::vector< MessageVal > > &m)
  Encode a message.

- template<typename Key >
  void decode (const Key &k, const std::string &hex, std::map< std::string, MessageVal > &m)
  Decode a message.

- template<typename Key >
  void decode (const Key &k, const std::string &hex, std::map< std::string, std::vector< MessageVal > > &m)
  Decode a message.

Informational Methods

- template<typename Key >
  std::string summary (const Key &k) const
- std::string summary () const
  long summary of a message for all loaded messages

- template<typename Key >
  std::string brief_summary (const Key &k) const
  brief summary of a message for a given Key (std::string name or unsigned id)

- std::string brief_summary () const
  brief summary of a message for all loaded messages

- unsigned message_count ()
- template<typename Key >
  unsigned get_repeat (const Key &k)
- std::set< unsigned > all_message_ids ()
- std::set< std::string > all_message_names ()
- template<typename Key >
  std::map< std::string, std::string > message_var_names (const Key &k) const
- std::string id2name (unsigned id)
- unsigned name2id (const std::string &name)

Publish/subscribe architecture related methods

Methods written largely to support DCCL in the context of a publish/subscribe architecture (e.g., see MOOS (http://www.robots.ox.ac.uk/~mobile/MOOS/wiki/pmwiki.php)). The other methods provide a complete interface for encoding and decoding DCCL messages. However, the methods listed here extend the functionality to allow for

- *message creation triggering (a message is encoded on a certain event, either time based or publish based)*
- *encoding that will parse strings of the form: "key1=value,key2=value,key3=value"*
- *decoding to an arbitrarily formatted string (similar concept to printf)*

These methods will be useful if you are interested in any of the features mentioned above.

- template<typename Key >
  void pubsub_encode (const Key &k, modem::Message &msg, const std::map< std::string, std::vector< dccl::MessageVal > > &pubsub_vals)
  Encode a message using <src_var> tags instead of <name> tags.

- template<typename Key >
  void pubsub_encode (const Key &k, modem::Message &msg, const std::map< std::string, MessageVal > &pubsub_vals)
  Encode a message using <src_var> tags instead of <name> tags.

- template<typename Key >
  void pubsub_decode (const Key &k, const modem::Message &msg, std::multimap< std::string, dccl::MessageVal > &pubsub_vals)
  Decode a message using formatting specified in <publish> tags.

- template<typename Key >
  std::set< std::string > get_pubsub_src_vars (const Key &k)
  what moos variables do i need to provide to create a message with a call to encode_using_src_vars

- template<typename Key >
  std::set< std::string > get_pubsub_all_vars (const Key &k)
  for a given message name, all architecture variables (sources, input, destination, trigger)

- template<typename Key >
  std::set< std::string > get_pubsub_encode_vars (const Key &k)
  all architecture variables needed for encoding (includes trigger)

- template<typename Key >
  std::set< std::string > get_pubsub_decode_vars (const Key &k)
  for a given message, all architecture variables for decoding (input)

- template<typename Key >
  std::string get_outgoing_hex_var (const Key &k)
  returns outgoing architecture hexadecimal variable

- template<typename Key >
  std::string get_incoming_hex_var (const Key &k)
  returns incoming architecture hexadecimal variable

- bool is_publish_trigger (std::set< unsigned > &id, const std::string &key, const std::string &value)
  look if key / value are trigger for any loaded messages if so, store to id and return true

- bool is_time_trigger (std::set< unsigned > &id)
  look if the time is right for trigger for any loaded messages if so, store to id and return true

- bool is_incoming (unsigned &id, const std::string &key)
  see if this key is for an incoming message if so, return id for decoding

E.2.1 Detailed Description

provides an API to the Dynamic CCL Codec.

Examples:

acomms/examples/chat/chat.cpp, libdccl/examples/dccl_simple/dccl_simple.cpp, libdccl/examples/plusnet/plusnet.cpp, libdccl/examples/test/test.cpp, and libdccl/examples/two_message/two_message.cpp.

Definition at line 76 of file dccl.h.

E.2.2 Constructor & Destructor Documentation

E.2.2.1 dccl::DCCLCodec::DCCLCodec (const std::string & *file*, const std::string *schema* = "")

Instantiate with a single XML file.

Parameters

file path to an XML message file (i.e. contains <layout> and (optionally) <publish> sections) to parse for use by the codec.

schema path (absolute or relative to the XML file path) for the validating schema (message_schema.xsd) (optional).

Definition at line 35 of file dccl.cpp.

E.2.2.2 dccl::DCCLCodec::DCCLCodec (const std::set< std::string > & *files*, const std::string *schema* = "")

Instantiate with a set of XML files.

Parameters

files set of paths to XML message files to parse for use by the codec.

schema path (absolute or relative to the XML file path) for the validating schema (message_schema.xsd) (optional).

Definition at line 38 of file dccl.cpp.

E.2.3 Member Function Documentation

E.2.3.1 void dccl::DCCLCodec::add_adv_algorithm (const std::string & *name*, AlgFunction2 *func*)

Add an advanced algorithm callback for any DCCL C++ type that may also require knowledge of all the other message variables and can optionally have additional parameters.

Parameters

name name of the algorithm (<... algorithm="name:param1:param2">)

func has the form void name(MessageVal& val_to_edit, const std::vector<std::string> params, const std::map<std::string,MessageVal>& vals) (see dccl::AdvAlgFunction3). func can be a function pointer (&name) or any function object supported by boost::function (http://www.boost.org/doc/libs/1_34_0/doc/html/function.html).

params (passed to func) a list of colon separated parameters passed by the user in the XML file. param[0] is the name.

vals (passed to func) a map of <name> to current values for all message variables.

Definition at line 118 of file dccl.cpp.

E.2.3.2 void dccl::DCCLCodec::add_algorithm (const std::string & *name*, AlgFunction1 *func*)

Add an algorithm callback for a dccl::MessageVal. The return value is stored back into the input parameter (dccl::MessageVal). See test.cpp for an example.

Parameters

name name of the algorithm (<... algorithm="name">)

func has the form void name(dccl::MessageVal& val_to_edit) (see dccl::AdvAlgFunction1). can be a function pointer (&name) or any function object supported by boost::function (http://www.boost.org/doc/libs/1_34_0/doc/html/function.html)

Definition at line 112 of file dccl.cpp.

E.2.3.3 std::set< unsigned > dccl::DCCLCodec::add_xml_message_file (const std::string & *xml_file*, const std::string *xml_schema* = "")

Add more messages to this instance of the codec.

Parameters

xml_file path to the xml file to parse and add to this codec.

xml_schema path to the message_schema.xsd file to validate XML with. if using a relative path this must be relative to the directory of the xml_file, not the present working directory. if not provided no validation is done.

Returns

returns id of the last message file parsed. note that there can be more than one message in a file

Examples:

acomms/examples/chat/chat.cpp, and libdccl/examples/plusnet/plusnet.cpp.

Definition at line 45 of file dccl.cpp.

E.2.3.4 std::set< unsigned > dccl::DCCLCodec::all_message_ids ()

Returns

set of all message ids loaded

Definition at line 80 of file dccl.cpp.

E.2.3.5 std::set< std::string > dccl::DCCLCodec::all_message_names ()

Returns

set of all message names loaded

Definition at line 87 of file dccl.cpp.

E.2.3.6 template<typename Key > void dccl::DCCLCodec::decode (const Key & *k*, const std::string & *hex*, std::map< std::string, std::vector< MessageVal > > & *m*) [inline]

Decode a message.

Parameters

k can either be std::string (the name of the message) or unsigned (the id of the message

hex the hexadecimal to be decoded.

m map of std::string (<name>) to dccl::MessageVal to store the values to be decoded

Definition at line 212 of file dccl.h.

E.2.3.7 template<typename Key > void dccl::DCCLCodec::decode (const Key & *k*, const std::string & *hex*, std::map< std::string, MessageVal > & *m*) [inline]

Decode a message.

Parameters

k can either be std::string (the name of the message) or unsigned (the id of the message

hex the hexadecimal to be decoded.

m map of std::string (<name>) to dccl::MessageVal to store the values to be decoded. No fields can be arrays using this call. If fields are arrays, only the first value is returned.

Examples:

acomms/examples/chat/chat.cpp, and libdccl/examples/dccl_simple/dccl_simple.cpp.

Definition at line 194 of file dccl.h.

E.2.3.8 template<typename Key > void dccl::DCCLCodec::encode (const Key & *k*, std::string & *hex*, const std::map< std::string, std::vector< MessageVal > > & *m*) [inline]

Encode a message.

Parameters

k can either be std::string (the name of the message) or unsigned (the id of the message)

hex location for the encoded hexadecimal to be stored. this is suitable for sending to the Micro-Modem

m map of std::string (<name>) to a vector of dccl::MessageVal representing the values to encode. Fields can be arrays.

Definition at line 183 of file dccl.h.

E.2.3.9 template<typename Key > void dccl::DCCLCodec::encode (const Key & *k*, std::string & *hex*, const std::map< std::string, MessageVal > & *m*) [inline]

Encode a message.

Parameters

k can either be std::string (the name of the message) or unsigned (the id of the message)

hex location for the encoded hexadecimal to be stored. this is suitable for sending to the Micro-Modem

m map of std::string (<name>) to a vector of dccl::MessageVal representing the values to encode. No fields can be arrays using this call. If fields are arrays, all values but the first in the array will be set to NaN or blank.

Examples:

acomms/examples/chat/chat.cpp, and libdccl/examples/dccl_simple/dccl_simple.cpp.

Definition at line 165 of file dccl.h.

E.2.3.10 template<typename Key > unsigned dccl::DCCLCodec::get_repeat (const Key & *k*) [inline]

Returns

repeat value (number of copies of the message per encode)

Definition at line 243 of file dccl.h.

E.2.3.11 std::string dccl::DCCLCodec::id2name (unsigned *id*) [inline]

Parameters

id message id

Returns

name of message

Definition at line 257 of file dccl.h.

E.2.3.12 unsigned dccl::DCCLCodec::message_count () [inline]

how many message are loaded?

Returns

number of messages loaded

Examples:

libdccl/examples/two_message/two_message.cpp.

Definition at line 239 of file dccl.h.

E.2.3.13 template<typename Key > std::map<std::string, std::string> dccl::DCCLCodec::message_var_names (const Key & *k*) const [inline]

Returns

map of names to DCCL types needed to encode a given message

Definition at line 252 of file dccl.h.

E.2.3.14 unsigned dccl::DCCLCodec::name2id (const std::string & *name*) [inline]

Parameters

name message name

Returns

id of message

Definition at line 260 of file dccl.h.

E.2.3.15 template<typename Key > void dccl::DCCLCodec::pubsub_decode (const Key & *k*, const modem::Message & *msg*, std::multimap< std::string, dccl::MessageVal > & *pubsub_vals*) [inline]

Decode a message using formatting specified in <publish> tags.

Values will be received in two maps, one of strings and the other of doubles. The <publish> value will be placed either based on the "type" parameter of the <publish_var> tag (e.g. <publish_var type="long">SOMEVAR</publish_var> will be placed as a long). If no type parameter is given and the variable is numeric (e.g. "23242.23") it will be considered a double. If not numeric, it will be considered a string.

Parameters

k can either be std::string (the name of the message) or unsigned (the id of the message)

msg modem::Message or std::string to be decode.

vals pointer to std::multimap of publish variable name to std::string values.

Definition at line 340 of file dccl.h.

E.2.3.16 template<typename Key > void dccl::DCCLCodec::pubsub_encode (const Key & *k*, modem::Message & *msg*, const std::map< std::string, MessageVal > & *pubsub_vals*) [inline]

Encode a message using <src_var> tags instead of <name> tags.

Use this version if you do not have vectors of src_var values

Definition at line 317 of file dccl.h.

E.2.3.17 template<typename Key > void dccl::DCCLCodec::pubsub_encode (const Key & *k*, modem::Message & *msg*, const std::map< std::string, std::vector< dccl::MessageVal > > & *pubsub_vals*) [inline]

Encode a message using <src_var> tags instead of <name> tags.

Values can be passed in on one or more maps of names to values, similar to DCCLCodec::encode. Casts are made and string parsing of key=value comma delimited fields is performed. This differs substantially from the behavior of encode above. For example, take this message variable:

```
<int>
  <name>myint</name>
  <src_var>somevar</src_var>
</int>
```

Using this method you can pass vals["somevar"] = "mystring=foo,blah=dog,myint=32" or vals["somevar"] = 32.0 and both cases will properly parse out 32 as the value for this field. In comparison, using the normal encode you would pass vals["myint"] = 32

Parameters

k can either be std::string (the name of the message) or unsigned (the id of the message)

msg modem::Message or std::string for encoded message to be stored.

vals map of source variable name to dccl::MessageVal values.

Examples:

libdccl/examples/plusnet/plusnet.cpp.

Definition at line 296 of file dccl.h.

E.2.3.18 void dccl::DCCLCodec::set_crypto_passphrase (const std::string & *passphrase*)

Set a passphrase for encrypting all messages with.

Parameters

passphrase text passphrase

Definition at line 322 of file dccl.cpp.

E.2.3.19 void dccl::DCCLCodec::set_modem_id (unsigned *modem_id*) [inline]

Set the modem id for this vehicle.

Parameters

modem_id unique (within a network) number representing the modem on this vehicle.

Definition at line 130 of file dccl.h.

E.2.3.20 void dccl::DCCLCodec::set_schema (const std::string & *schema*) [inline]

Set the schema used for xml syntax checking.

location is relative to the XML file location! if you have XML files in different places you must pass the proper relative path (or just use absolute paths)

Parameters

schema location of the message_schema.xsd file

Definition at line 120 of file dccl.h.

E.2.3.21 template<typename Key > std::string dccl::DCCLCodec::summary (const Key & *k*) const [inline]

long summary of a message for a given Key (std::string name or unsigned id)

Parameters

k can either be std::string (the name of the message) or unsigned (the id of the message)

Definition at line 225 of file dccl.h.

The documentation for this class was generated from the following files:

- /var/www/root/goby-bzr/src/acomms/libdccl/dccl.h
- /var/www/root/goby-bzr/src/acomms/libdccl/dccl.cpp

E.3 modem::DriverBase Class Reference

provides a base class for acoustic modem drivers (i.e. for different manufacturer modems) to derive

`#include <driver_base.h>`

Inherited by micromodem::MMDriver.

Public Member Functions

- virtual void startup ()=0

  Virtual startup method. see derived classes (e.g. micromodem::MMDriver) for examples.

- virtual void do_work ()=0

  Virtual do_work method. see derived classes (e.g. micromodem::MMDriver) for examples.

- virtual void initiate_transmission (const modem::Message &m)=0

  Virtual initiate_transmission method. see derived classes (e.g. micromodem::MMDriver) for examples.

- virtual void initiate_ranging (const modem::Message &m)=0

  Virtual initiate_ranging method. see derived classes (e.g. micromodem::MMDriver) for examples.

- void set_cfg (const std::vector< std::string > &cfg)

Set configuration strings for the modem. The contents of these strings depends on the specific modem.

- void set_serial_port (const std::string &s)

  Set the serial port name (e.g. /dev/ttyS0).

- void set_baud (unsigned u)

  Set the serial port baud rate (e.g. 19200).

- void set_receive_cb (MsgFunc1 func)

  Set the callback to receive incoming modem messages.

- void set_range_reply_cb (MsgFunc1 func)

  Set the callback to receive incoming ranging responses.

- void set_ack_cb (MsgFunc1 func)

  Set the callback to receive acknowledgements from the modem.

- void set_datarequest_cb (MsgFunc2 func)

  Set the callback to handle data requests from the modem (or the modem driver, if the modem does not have this functionality).

- void set_in_parsed_cb (MsgFunc1 func)

  Set the callback to pass all parsed messages to (i.e. modem::Message representation of the serial line).

- void set_in_raw_cb (StrFunc1 func)

  Set the callback to pass raw incoming modem strings to.

- void set_out_raw_cb (StrFunc1 func)

  Set the callback to pass all raw outgoing modem strings to.

- std::string serial_port ()
- unsigned baud ()

Protected Member Functions

- DriverBase (std::ostream *out, const std::string &line_delimiter)

  Constructor.

- ~DriverBase ()

  Destructor.

- void serial_write (const std::string &out)

  write a line to the serial port.

- bool serial_read (std::string &in)

  read a line from the serial port, including end-of-line character(s)

- void serial_start ()

  start the serial port. must be called before DriverBase::serial_read() or DriverBase::serial_write()

Protected Attributes

- std::vector< std::string > cfg_
  vector containing the configuration parameters intended to be set during startup()

E.3.1 Detailed Description

provides a base class for acoustic modem drivers (i.e. for different manufacturer modems) to derive

Definition at line 50 of file driver_base.h.

E.3.2 Constructor & Destructor Documentation

E.3.2.1 modem::DriverBase::DriverBase (std::ostream * *out*, const std::string & *line_delimiter*) [protected]

Constructor.

Parameters

out pointer to std::ostream to log human readable debugging and runtime information

line_delimiter string indicating the end-of-line character(s) from the serial port (usually newline ("\n") or carriage-return and newline ("\r\n"))

Definition at line 26 of file driver_base.cpp.

E.3.3 Member Function Documentation

E.3.3.1 unsigned modem::DriverBase::baud () [inline]

Returns

the serial port baud rate

Definition at line 133 of file driver_base.h.

E.3.3.2 std::string modem::DriverBase::serial_port () [inline]

Returns

the serial port name

Definition at line 130 of file driver_base.h.

E.3.3.3 bool modem::DriverBase::serial_read (std::string & *in*) [protected]

read a line from the serial port, including end-of-line character(s)

Parameters

in reference to string to store line

Returns

true if a line was available, false if no line available

Definition at line 42 of file driver_base.cpp.

E.3.3.4 void modem::DriverBase::serial_start () [protected]

start the serial port. must be called before DriverBase::serial_read() or DriverBase::serial_write()

Exceptions

std::runtime_error Serial port could not be opened.

Definition at line 59 of file driver_base.cpp.

E.3.3.5 void modem::DriverBase::serial_write (const std::string & *out*) [protected]

write a line to the serial port.

Parameters

out reference to string to write. Must already include any end-of-line character(s).

Definition at line 36 of file driver_base.cpp.

E.3.3.6 void modem::DriverBase::set_ack_cb (MsgFunc1 *func*) [inline]

Set the callback to receive acknowledgements from the modem.

If using the queue::QueueManager, pass queue::QueueManager::handle_modem_ack to this method.

Parameters

func Pointer to function (or any other object boost::function accepts) matching the signature of modem::MsgFunc1. The callback (func) will be invoked with the following parameters:

message A modem::Message containing the acknowledgement message

Definition at line 94 of file driver_base.h.

E.3.3.7 void modem::DriverBase::set_datarequest_cb (MsgFunc2 *func*) [inline]

Set the callback to handle data requests from the modem (or the modem driver, if the modem does not have this functionality).

If using the queue::QueueManager, pass queue::QueueManager::provide_outgoing_modem_data to this method.

Parameters

func Pointer to function (or other boost::function object) of the signature modem::MsgFunc2. The callback (func) will be invoked with the following parameters:

message1 (incoming) The modem::Message containing the details of the request (source, destination, size, etc.)

message2 (outgoing) The modem::Message to be sent. This should be populated by the callback.

Definition at line 102 of file driver_base.h.

E.3.3.8 void modem::DriverBase::set_in_parsed_cb (MsgFunc1 *func*) [inline]

Set the callback to pass all parsed messages to (i.e. modem::Message representation of the serial line).

If using the amac::MACManager, pass amac::MACManager::process_message to this method.

Parameters

func Pointer to function (or any other object boost::function accepts) matching the signature of modem::MsgFunc1. The callback (func) will be invoked with the following parameters:

message A modem::Message containing the received modem message

Definition at line 110 of file driver_base.h.

E.3.3.9 void modem::DriverBase::set_in_raw_cb (StrFunc1 *func*) [inline]

Set the callback to pass raw incoming modem strings to.

Parameters

func Pointer to function (or any other object boost::function accepts) matching the signature of modem::StrFunc1. The callback (func) will be invoked with the following parameters:

std::string The raw incoming string

Definition at line 118 of file driver_base.h.

E.3.3.10 void modem::DriverBase::set_out_raw_cb (StrFunc1 *func*) [inline]

Set the callback to pass all raw outgoing modem strings to.

Parameters

func Pointer to function (or any other object boost::function accepts) matching the signature of modem::StrFunc1. The callback (func) will be invoked with the following parameters:

std::string The raw outgoing string

Definition at line 126 of file driver_base.h.

E.3.3.11 void modem::DriverBase::set_range_reply_cb (MsgFunc1 *func*) [inline]

Set the callback to receive incoming ranging responses.

Parameters

func Pointer to function (or any other object boost::function accepts) matching the signature of modem::MsgFunc1. The callback (func) will be invoked with the following parameters:

message A modem::Message reference containing the contents of the received ranging (in travel time in seconds)

Definition at line 86 of file driver_base.h.

E.3.3.12 void modem::DriverBase::set_receive_cb (MsgFunc1 *func*) [inline]

Set the callback to receive incoming modem messages.

Any messages received before this callback is set will be discarded. If using the queue::QueueManager, pass queue::QueueManager::receive_incoming_modem_data to this method.

Parameters

func Pointer to function (or any other object boost::function accepts) matching the signature of modem::MsgFunc1. The callback (func) will be invoked with the following parameters:

message A modem::Message reference containing the contents of the received modem message.

Definition at line 79 of file driver_base.h.

The documentation for this class was generated from the following files:

- /var/www/root/goby-bzr/src/acomms/libmodemdriver/driver_base.h
- /var/www/root/goby-bzr/src/acomms/libmodemdriver/driver_base.cpp

E.4 amac::MACManager Class Reference

provides an API to the goby-acomms MAC library.

`#include <mac_manager.h>`

Public Member Functions

- void startup ()

  Starts the MAC.

- void do_work ()

  Must be called regularly for the MAC to perform its work.

- void send_poll (const asio::error_code &)

  Manually initiate a transmission out of the normal cycle. This is not normally called by the user of MACManager.

- void process_message (const modem::Message &m)

  Call every time a message is received from vehicle to "discover" this vehicle or reset the expire timer. Only needed when the type is amac::mac_slotted_tdma.

Constructors/Destructor

- MACManager (std::ostream *os=0)

  Default constructor.

- ~**MACManager** ()

Manipulate slots

- std::map< unsigned, amac::Slot >::iterator add_slot (const amac::Slot &s)
- bool remove_slot (const amac::Slot &s)

  removes any slots in the cycle where amac::operator==(const Slot&, const Slot&) is true.

- void **clear_all_slots** ()

Set

- void **set_type** (MACType type)
- void **set_modem_id** (unsigned modem_id)
- void **set_modem_id** (const std::string &s)
- void **set_rate** (int rate)
- void **set_rate** (const std::string &s)
- void **set_slot_time** (unsigned slot_time)
- void **set_slot_time** (const std::string &s)
- void **set_expire_cycles** (unsigned expire_cycles)
- void **set_expire_cycles** (const std::string &s)
- void set_destination_cb (IdFunc func)

  Callback to call to request which vehicle id should be the next destination. Typically bound to queue::QueueManager::request_next_destination.

- void set_initiate_transmission_cb (MsgFunc1 func)

  Callback for initiate a tranmission. Typically bound to modem::DriverBase::initiate_transmission.

- void set_initiate_ranging_cb (MsgFunc1 func)

  Callback for initiate ranging ("ping"). Typically bound to modem::DriverBase::initiate_ranging.

Get

- int **rate** ()
- unsigned **slot_time** ()
- MACType **type** ()

E.4.1 Detailed Description

provides an API to the goby-acomms MAC library.

Examples:

acomms/examples/chat/chat.cpp, and libamac/examples/amac_simple/amac_simple.cpp.

Definition at line 144 of file mac_manager.h.

E.4.2 Constructor & Destructor Documentation

E.4.2.1 amac::MACManager::MACManager (std::ostream * *os* = 0)

Default constructor.

Parameters

os std::ostream object or FlexOstream to capture all humanly readable runtime and debug information (optional).

Definition at line 36 of file mac_manager.cpp.

E.4.3 Member Function Documentation

E.4.3.1 std::map< unsigned, amac::Slot >::iterator amac::MACManager::add_slot (const amac::Slot & *s*)

Returns

iterator to newly added slot

Definition at line 321 of file mac_manager.cpp.

E.4.3.2 void amac::MACManager::process_message (const modem::Message & *m*)

Call every time a message is received from vehicle to "discover" this vehicle or reset the expire timer. Only needed when the type is amac::mac_slotted_tdma.

Parameters

message the new message (used to detect vehicles)

Definition at line 234 of file mac_manager.cpp.

E.4.3.3 bool amac::MACManager::remove_slot (const amac::Slot & *s*)

removes any slots in the cycle where amac::operator==(const Slot&, const Slot&) is true.

Returns

true if one or more slots are removed

Definition at line 342 of file mac_manager.cpp.

The documentation for this class was generated from the following files:

- /var/www/root/goby-bzr/src/acomms/libamac/mac_manager.h
- /var/www/root/goby-bzr/src/acomms/libamac/mac_manager.cpp

E.5 modem::Message Class Reference

represents a message to or from the acoustic modem.

`#include <modem_message.h>`

Public Member Functions

- std::string snip () const

  short snippet summarizing the Message

Constructors/Destructor

- Message (const std::string &s="")
  Construct a message for or from the modem.

Set

- void set_data (const std::string &data)
  set the hexadecimal string data. Also computes the size and the checksum.
- void set_t (double d)
  set the time
- void set_size (double size)
  set the size (automatically set by the data size)
- void set_src (unsigned src)
  set the source id
- void set_dest (unsigned dest)
  set the destination id
- void set_rate (unsigned rate)
  set the data rate
- void set_ack (bool ack)
  set the acknowledgement value
- void set_frame (unsigned frame)
  set the frame number
- void set_t (const std::string &t)
  try to set the time with std::string
- void set_size (const std::string &size)
  try to set the size with std::string
- void set_dest (const std::string &dest)
  try to set the destination id with std::string
- void set_src (const std::string &src)
  try to set the source id with std::string
- void set_rate (const std::string &rate)
  try to set the source id with std::string
- void set_ack (const std::string &ack)
  try to set the acknowledgement value with string ("true" / "false")
- void set_frame (const std::string &frame)
  try to set the frame number with std::string

Get

- std::string data () const
  hexadecimal string
- std::string & data_ref ()
  hexadecimal string reference
- double t () const
  time in seconds since 1970-1-1 00:00:00 UTC
- unsigned src () const
  source modem id
- unsigned dest () const
  destination modem id
- unsigned rate () const
  data rate (unsigned from 0 (lowest) to 5 (highest))
- bool ack () const
  acknowledgement requested
- unsigned frame () const
  modem frame number
- double size () const
  size in bytes
- unsigned cs () const
  checksum (eight bit XOR of Message::data())

Query availability

- bool data_set () const
  is there data?
- bool t_set () const
  is there a time?
- bool size_set () const
  is there a size?
- bool src_set () const
  is there a source?
- bool dest_set () const
  is there a destination?
- bool rate_set () const
  is there a rate?
- bool ack_set () const
  is there an ack value?

- bool frame_set () const
  is there a frame number?
- bool cs_set () const
  is there a checksum?
- bool empty () const
  is the Message empty (no data)?

Serialize/Unserialize

- std::string serialize () const
  full human readable string serialization
- void unserialize (const std::string &s)
  reverse serialization

E.5.1 Detailed Description

represents a message to or from the acoustic modem. Message is intended to represent all the possible messages from the modem. Thus, depending on the specific message certain fields may not be used. Also, fields make take on different meanings depending on the message type. Where this is the case, extra documentation is provided.

Examples:

acomms/examples/chat/chat.cpp, libamac/examples/amac_simple/amac_simple.cpp, libdccl/examples/plusnet/plusnet.cpp, libmodemdriver/examples/driver_simple/driver_simple.cpp, and libqueue/examples/queue_simple/queue_simple.cpp.

Definition at line 41 of file modem_message.h.

E.5.2 Constructor & Destructor Documentation

E.5.2.1 modem::Message::Message (const std::string & *s* = "") [inline]

Construct a message for or from the modem.

Parameters

s (optionally) pass a serialized string such as src=3,dest=4,data=CA342BDF ... to initialize

Definition at line 49 of file modem_message.h.

E.5.3 Member Function Documentation

E.5.3.1 std::string modem::Message::serialize () const [inline]

full human readable string serialization

Returns

string of key=value comma delimited pairs (e.g. "src=2,dest=3,data=ABCD22345"). the order of the keys in sentence is not specified.

Examples:

libdccl/examples/plusnet/plusnet.cpp.

Definition at line 217 of file modem_message.h.

The documentation for this class was generated from the following file:

- /var/www/root/goby-bzr/src/acomms/modem_message.h

E.6 dccl::MessageVal Class Reference

defines a DCCL value

`#include <message_val.h>`

Public Member Functions

Constructors/Destructor

- MessageVal ()
  empty
- MessageVal (const std::string &s)
  construct with string value
- MessageVal (const char *s)
  construct with char value*
- MessageVal (double d, int p=MAX_DBL_PRECISION)
  construct with double value, optionally giving the precision of the double (number of decimal places) which is used if a cast to std::string is required in the future.
- MessageVal (long l)
  construct with long value
- MessageVal (int i)
  construct with int value
- MessageVal (float f)
  construct with float value
- MessageVal (bool b)
  construct with bool value
- MessageVal (const std::vector< MessageVal > &vm)
  construct with vector

Setters

- void set (std::string sval)
  set the value with a string (overwrites previous value regardless of type)
- void set (double dval, int precision=MAX_DBL_PRECISION)
  set the value with a double (overwrites previous value regardless of type)
- void set (long lval)
  set the value with a long (overwrites previous value regardless of type)
- void set (bool bval)
  set the value with a bool (overwrites previous value regardless of type)

Getters

- bool get (std::string &s) const
  extract as std::string (all reasonable casts are done)
- bool get (bool &b) const
  extract as bool (all reasonable casts are done)
- bool get (long &t) const
  extract as long (all reasonable casts are done)
- bool get (double &d) const
  extract as double (all reasonable casts are done)
- operator double () const
- operator bool () const
- operator std::string () const
- operator long () const
- operator int () const
- operator unsigned () const
- operator float () const
- **operator std::vector**< **MessageVal** > () const
- DCCLCppType type () const
  what type is the original type of this MessageVal?
- bool empty () const
  was this just constructed with MessageVal() ?
- unsigned **precision** () const

Comparison

- bool **operator==** (const MessageVal &mv) const
- bool **operator==** (const std::string &s) const
- bool **operator==** (double d) const
- bool **operator==** (long l) const
- bool **operator==** (bool b) const

E.6.1 Detailed Description

defines a DCCL value

Examples:

libdccl/examples/test/test.cpp.

Definition at line 35 of file message_val.h.

E.6.2 Member Function Documentation

E.6.2.1 bool dccl::MessageVal::get (double & *d*) const

extract as double (all reasonable casts are done)

Parameters

d double to store value in

Returns

successfully extracted (and if necessary successfully cast to this type)

Definition at line 215 of file message_val.cpp.

E.6.2.2 bool dccl::MessageVal::get (long & *t*) const

extract as long (all reasonable casts are done)

Parameters

t long to store value in

Returns

successfully extracted (and if necessary successfully cast to this type)

Definition at line 176 of file message_val.cpp.

E.6.2.3 bool dccl::MessageVal::get (bool & *b*) const

extract as bool (all reasonable casts are done)

Parameters

b bool to store value in

Returns

successfully extracted (and if necessary successfully cast to this type)

Definition at line 144 of file message_val.cpp.

E.6.2.4 bool dccl::MessageVal::get (std::string & *s*) const

extract as std::string (all reasonable casts are done)

Parameters

s std::string to store value in

Returns

successfully extracted (and if necessary successfully cast to this type)

Examples:

libdccl/examples/test/test.cpp.

Definition at line 113 of file message_val.cpp.

E.6.2.5 dccl::MessageVal::operator bool () const

allows statements of the form

```
bool b = MessageVal("1");
```

Definition at line 262 of file message_val.cpp.

E.6.2.6 dccl::MessageVal::operator double () const

allows statements of the form

```
double d = MessageVal("3.23");
```

Definition at line 254 of file message_val.cpp.

E.6.2.7 dccl::MessageVal::operator float () const

allows statements of the form

```
float f = MessageVal("3.5");
```

Definition at line 293 of file message_val.cpp.

E.6.2.8 dccl::MessageVal::operator int () const

allows statements of the form

```
int i = MessageVal(2);
```

Definition at line 283 of file message_val.cpp.

E.6.2.9 dccl::MessageVal::operator long () const

allows statements of the form

```
long l = MessageVal(5);
```

Definition at line 276 of file message_val.cpp.

E.6.2.10 dccl::MessageVal::operator std::string () const

allows statements of the form

```
std::string s = MessageVal(3);
```

Definition at line 269 of file message_val.cpp.

E.6.2.11 dccl::MessageVal::operator unsigned () const

allows statements of the form

```
unsigned u = MessageVal(2);
```

Definition at line 288 of file message_val.cpp.

E.6.2.12 void dccl::MessageVal::set (double *dval*, int *precision* = MAX_DBL_PRECISION)

set the value with a double (overwrites previous value regardless of type)

Parameters

dval values to set

precision decimal places of precision to preserve if this is cast to a string

Definition at line 105 of file message_val.cpp.

The documentation for this class was generated from the following files:

- /var/www/root/goby-bzr/src/acomms/libdccl/message_val.h
- /var/www/root/goby-bzr/src/acomms/libdccl/message_val.cpp

E.7 micromodem::MMDriver Class Reference

provides an API to the WHOI Micro-Modem driver

`#include <mm_driver.h>`

Inherits modem::DriverBase.

Public Member Functions

- MMDriver (std::ostream *out=0)
  Default constructor.
- ~MMDriver ()
  Destructor.
- void startup ()
  Starts the driver.
- void do_work ()
  Must be called regularly for the driver to perform its work.
- void initiate_transmission (const modem::Message &m)
  Initiate a transmission to the modem.
- void initiate_ranging (const modem::Message &m)
  Initiate ranging ("ping") to the modem.
- void set_gateway_prefix (bool IsGateway, int GatewayID)

E.7.1 Detailed Description

provides an API to the WHOI Micro-Modem driver

Examples:

acomms/examples/chat/chat.cpp, and libmodemdriver/examples/driver_simple/driver_simple.cpp.

Definition at line 56 of file mm_driver.h.

E.7.2 Constructor & Destructor Documentation

E.7.2.1 micromodem::MMDriver::MMDriver (std::ostream * *out* = 0)

Default constructor.

Parameters

os std::ostream object or FlexOstream to capture all humanly readable runtime and debug information (optional).

Definition at line 32 of file mm_driver.cpp.

E.7.3 Member Function Documentation

E.7.3.1 void micromodem::MMDriver::initiate_ranging (const modem::Message & *m*) [virtual]

Initiate ranging ("ping") to the modem.

Parameters

m modem::Message containing the details of the ranging request to be started. (source and destination)

Implements modem::DriverBase.

Definition at line 108 of file mm_driver.cpp.

E.7.3.2 void micromodem::MMDriver::initiate_transmission (const modem::Message & *m*) [virtual]

Initiate a transmission to the modem.

Parameters

m modem::Message containing the details of the transmission to be started. This does ∗not∗ contain data, which must be requested in a call to the datarequest callback (set by DriverBase::set_data_request_cb)

Implements modem::DriverBase.

Definition at line 95 of file mm_driver.cpp.

E.7.3.3 void micromodem::MMDriver::set_gateway_prefix (bool *IsGateway*, int *GatewayID*)

set_gateway_prefix - This function creates a prefix that is required to support the use of the Hydroid gateway buoy for acoustic communications

Parameters

IsGateway - A bool indicating whether the sytem is using a gateway buoy for acoustic communications

GatewayID - The numberical index of the buoy being used

Definition at line 492 of file mm_driver.cpp.

The documentation for this class was generated from the following files:

- /var/www/root/goby-bzr/src/acomms/libmodemdriver/mm_driver.h
- /var/www/root/goby-bzr/src/acomms/libmodemdriver/mm_driver.cpp

E.8 queue::QueueConfig Class Reference

defines parameters for configuring a message queue

`#include <queue_config.h>`

Public Member Functions

- void set_ack (bool ack)

  sets whether the queue should require an acknowledgement of all data

- void set_blackout_time (unsigned blackout_time)

  sets how long (in seconds) between sending the last message from this queue should this queue be ignored (considered "in blackout")

- void set_max_queue (unsigned max_queue)

  sets how many messages fit in this queue before discarding the oldest (newest_first = true) or the newest (newest_first = false)

- void set_newest_first (bool newest_first)

  sets whether the newest messages are sent first (FILO queue) or not (FIFO queue)

- void set_value_base (double value_base)

  sets the base value

- void set_ttl (unsigned ttl)

  sets the time to live

- void set_type (QueueType t)

  sets the type of the queue

- void set_id (unsigned id)

  sets the id of the queue. This id is the DCCL id (<id>) for queues of type queue_dccl. On the other hand, this id is the CCL id (first byte in decimal) for queues of type queue_ccl. The type and id together form a unique key (a QueueKey).

- void set_name (const std::string &name)

  sets the name of the queue. This is used informational and can be omitted if desired.

- bool ack () const
- unsigned blackout_time () const
- unsigned max_queue () const
- bool newest_first () const
- QueueType type () const
- unsigned id () const
- std::string name () const
- double value_base () const
- unsigned ttl () const

E.8.1 Detailed Description

defines parameters for configuring a message queue

Definition at line 55 of file queue_config.h.

E.8.2 Member Function Documentation

E.8.2.1 bool queue::QueueConfig::ack () const [inline]

Returns

current acknowledgement setting

Definition at line 125 of file queue_config.h.

E.8.2.2 unsigned queue::QueueConfig::blackout_time () const [inline]

Returns

current blackout time settings (seconds)

Definition at line 127 of file queue_config.h.

E.8.2.3 unsigned queue::QueueConfig::id () const [inline]

Returns

the id of the queue.

Definition at line 135 of file queue_config.h.

E.8.2.4 unsigned queue::QueueConfig::max_queue () const [inline]

Returns

current queue maximum setting (# of messages)

Definition at line 129 of file queue_config.h.

E.8.2.5 std::string queue::QueueConfig::name () const [inline]

Returns

the name of the queue.

Definition at line 137 of file queue_config.h.

E.8.2.6 bool queue::QueueConfig::newest_first () const [inline]

Returns

whether new messages are sent first (true) or not (false)

Definition at line 131 of file queue_config.h.

E.8.2.7 unsigned queue::QueueConfig::ttl () const [inline]

Returns

the time to live of messages

Definition at line 141 of file queue_config.h.

E.8.2.8 QueueType queue::QueueConfig::type () const [inline]

Returns

the type of the queue.

Definition at line 133 of file queue_config.h.

E.8.2.9 double queue::QueueConfig::value_base () const [inline]

Returns

the base value of messages in this queue

Definition at line 139 of file queue_config.h.

The documentation for this class was generated from the following file:

- /var/www/root/goby-bzr/src/acomms/libqueue/queue_config.h

E.9 queue::QueueKey Class Reference

forms a unique key for a given message queue

`#include <queue_key.h>`

Public Member Functions

- void set_type (QueueType type)
  set the key type
- void set_id (unsigned id)
  set the key id (DCCL id for type = queue_dccl, CCL id for type = queue_ccl)
- QueueType type () const
- unsigned id () const

E.9.1 Detailed Description

forms a unique key for a given message queue

Examples:

acomms/examples/chat/chat.cpp, and libqueue/examples/queue_simple/queue_simple.cpp.

Definition at line 26 of file queue_key.h.

E.9.2 Member Function Documentation

E.9.2.1 unsigned queue::QueueKey::id () const [inline]

Returns

the key id

Examples:

acomms/examples/chat/chat.cpp.

Definition at line 42 of file queue_key.h.

E.9.2.2 QueueType queue::QueueKey::type () const [inline]

Returns

the key type

Definition at line 40 of file queue_key.h.

The documentation for this class was generated from the following file:

- /var/www/root/goby-bzr/src/acomms/libqueue/queue_key.h

E.10 queue::QueueManager Class Reference

provides an API to the goby-acomms Queuing Library.

`#include <queue_manager.h>`

Public Member Functions

Constructors/Destructor

- QueueManager (std::ostream *os=0)
  Default constructor.
- QueueManager (const std::string &file, const std::string schema="", std::ostream *os=0)
  Instantiate with a single XML file.
- QueueManager (const std::set< std::string > &files, const std::string schema="", std::ostream *os=0)
  Instantiate with a set of XML files.
- QueueManager (const QueueConfig &cfg, std::ostream *os=0)
  Instantiate with a single QueueConfig object.
- QueueManager (const std::set< QueueConfig > &cfgs, std::ostream *os=0)
  Instantiate with a set of QueueConfig objects.
- ~QueueManager ()
  Destructor.

Initialization Methods

These methods are intended to be called before doing any work with the class. However, they may be called at any time as desired.

- void add_xml_queue_file (const std::string &xml_file, const std::string xml_schema="")
  Add more queues by configuration XML files (typically contained in DCCL message XML files).
- void add_queue (const QueueConfig &cfg)
  Add more Queues.
- void set_schema (const std::string schema)

Set the schema used for XML syntax checking.

- void set_modem_id (unsigned modem_id)
  Set the modem id for this vehicle.

- void set_on_demand (QueueKey key)
  Set a queue to call the data_on_demand callback every time data is request (basically forwards the modem data_request).

- void set_on_demand (unsigned id, QueueType type=queue_dccl)
  Set a queue to call the data_on_demand callback every time data is request (basically forwards the modem data_request).

Application level Push/Receive Methods

These methods are the primary higher level interface to the QueueManager. From here you can push messages and set the callbacks to use on received messages.

- void push_message (QueueKey key, modem::Message &new_message)
  Push a message using a QueueKey as a key.

- void push_message (unsigned id, modem::Message &new_message, QueueType type=queue_dccl)
  Push a message using the queue id and type together as a key.

- void set_receive_cb (MsgFunc1 func)
  Set the callback to receive incoming DCCL messages. Any messages received before this callback is set will be discarded.

- void set_receive_ccl_cb (MsgFunc1 func)
  Set the callback to receive incoming CCL messages. Any messages received before this callback is set will be discarded.

Modem Driver level Push/Receive Methods

These methods are the interface to the QueueManager from the modem driver.

- bool provide_outgoing_modem_data (const modem::Message &message_in, modem::Message &message_out)
  Finds data to send to the modem.

- void receive_incoming_modem_data (const modem::Message &message)
  Receive incoming data from the modem.

- void handle_modem_ack (const modem::Message &message)
  Receive acknowledgements from the modem.

Active methods

Call these methods when you want the QueueManager to perform time sensitive tasks

- void **do_work** ()

Additional Application level methods

Use these methods for advanced features and more fine grained control.

- void set_ack_cb (MsgFunc1 func)
  Set the callback to receive acknowledgements of message receipt.
- void set_data_on_demand_cb (MsgFunc2 func)
  Set the callback to process queues marked `on_demand` *by QueueManager::set_on_demand.*
- void set_queue_size_change_cb (QSizeFunc func)
  Set the callback to receive messages every time a queue changes size (that is, a message is popped or pushed).
- void set_expire_cb (MsgFunc1 func)
  Set the callback to receive messages every time a message is expired due to exceeding its time to live (ttl).

Medium Access Control methods

- int request_next_destination (unsigned size=std::numeric_limits< unsigned >::max())
  Request the modem id of the next destination. Required by the MACManager to determine the next communication destination.

Informational Methods

- std::string summary () const

E.10.1 Detailed Description

provides an API to the goby-acomms Queuing Library.

Examples:

acomms/examples/chat/chat.cpp, and libqueue/examples/queue_simple/queue_simple.cpp.

Definition at line 58 of file queue_manager.h.

E.10.2 Constructor & Destructor Documentation

E.10.2.1 queue::QueueManager::QueueManager (std::ostream * *os* = 0)

Default constructor.

Parameters

os std::ostream object or FlexOstream to capture all humanly readable runtime and debug information (optional).

Definition at line 32 of file queue_manager.cpp.

E.10.2.2 queue::QueueManager::QueueManager (const std::string & *file*, const std::string *schema* = "", std::ostream * *os* = 0)

Instantiate with a single XML file.

Parameters

file path to an XML queuing configuration (i.e. contains a

```
<queuing/>
```

section) file to parse for use by the QueueManager.

schema path (absolute or relative to the XML file path) for the validating schema (message_-schema.xsd) (optional).

os std::ostream object or FlexOstream to capture all humanly readable runtime and debug information (optional).

Definition at line 40 of file queue_manager.cpp.

E.10.2.3 queue::QueueManager::QueueManager (const std::set< std::string > & *files*, const std::string *schema* = "", std::ostream * *os* = 0)

Instantiate with a set of XML files.

Parameters

files set of paths to XML queuing configuration (i.e. contains a

```
<queuing/>
```

section) files to parse for use by the QueueManager.

os std::ostream object or FlexOstream to capture all humanly readable runtime and debug information (optional).

Definition at line 51 of file queue_manager.cpp.

E.10.2.4 queue::QueueManager::QueueManager (const QueueConfig & *cfg*, std::ostream * *os* = 0)

Instantiate with a single QueueConfig object.

Parameters

cfg QueueConfig object to initialize a new queue with. Use the QueueConfig largely for non-DCCL messages. Use the XML file constructors for XML configured DCCL messages.

os std::ostream object or FlexOstream to capture all humanly readable runtime and debug information (optional).

Definition at line 63 of file queue_manager.cpp.

E.10.2.5 queue::QueueManager::QueueManager (const std::set< QueueConfig > & *cfgs*, std::ostream * *os* = 0)

Instantiate with a set of QueueConfig objects.

Parameters

cfgs Set of QueueConfig objects to initialize new queues with. Use the QueueConfig largely for non-DCCL messages. Use the XML file constructors for XML configured DCCL messages.

os std::ostream object or FlexOstream to capture all humanly readable runtime and debug information (optional).

Definition at line 73 of file queue_manager.cpp.

E.10.3 Member Function Documentation

E.10.3.1 void queue::QueueManager::add_queue (const QueueConfig & *cfg*)

Add more Queues.

Parameters

QueueConfig& cfg: configuration object for the new queue.

Definition at line 84 of file queue_manager.cpp.

E.10.3.2 void queue::QueueManager::add_xml_queue_file (const std::string & *xml_file*, const std::string *xml_schema* = "")

Add more queues by configuration XML files (typically contained in DCCL message XML files).

Parameters

xml_file path to the XML file to parse and add to this codec.

xml_schema path to the message_schema.xsd file to validate XML with. if using a relative path this must be relative to the directory of the xml_file, not the present working directory. if not provided no validation is done.

Definition at line 109 of file queue_manager.cpp.

E.10.3.3 void queue::QueueManager::handle_modem_ack (const modem::Message & *message*)

Receive acknowledgements from the modem.

If using one of the classes inheriting modem::DriverBase, this method should be bound and passed to modem::DriverBase::set_ack_cb.

Parameters

message The modem::Message corresponding to the acknowledgement (dest, src, frame#)

Definition at line 408 of file queue_manager.cpp.

E.10.3.4 bool queue::QueueManager::provide_outgoing_modem_data (const modem::Message & *message_in*, modem::Message & *message_out*)

Finds data to send to the modem.

Data from the highest priority queue(s) will be combined to form a message equal or less than the size requested in modem::Message message_in. If using one of the classes inheriting modem::DriverBase, this method should be bound and passed to modem::DriverBase::set_datarequest_cb.

Parameters

message_in The modem::Message containing the details of the request (source, destination, size, etc.)

message_out The packed modem::Message ready for sending by the modem. This will be populated by this function.

Returns

true if successful in finding data to send, false if no data is available

Definition at line 277 of file queue_manager.cpp.

E.10.3.5 void queue::QueueManager::push_message (unsigned *id*, modem::Message & *new_message*, QueueType *type* = `queue_dccl`)

Push a message using the queue id and type together as a key.

Parameters

id DCCL message id (

```
<id/>
```

) that references the queue for which to push the message to.

new_message modem::Message to push.

Definition at line 168 of file queue_manager.cpp.

E.10.3.6 void queue::QueueManager::push_message (QueueKey *key*, modem::Message & *new_message*)

Push a message using a QueueKey as a key.

Parameters

key QueueKey that references the queue to push the message to.

new_message modem::Message to push.

Definition at line 144 of file queue_manager.cpp.

E.10.3.7 void queue::QueueManager::receive_incoming_modem_data (const modem::Message & *message*)

Receive incoming data from the modem.

If using one of the classes inheriting modem::DriverBase, this method should be bound and passed to modem::DriverBase::set_receive_cb.

Parameters

message The received modem::Message.

Definition at line 463 of file queue_manager.cpp.

E.10.3.8 int queue::QueueManager::request_next_destination (unsigned *size* = std::numeric_limits<unsigned>::max())

Request the modem id of the next destination. Required by the MACManager to determine the next communication destination.

Parameters

size size (in bytes) of the next transmission. Size affects the next destination since messages that are too large are disregarded.

Returns

id of the next destination

Definition at line 569 of file queue_manager.cpp.

E.10.3.9 void queue::QueueManager::set_ack_cb (MsgFunc1 *func*) [inline]

Set the callback to receive acknowledgements of message receipt.

Parameters

func Pointer to function (or any other object boost::function accepts) matching the signature of queue::MsgFunc1. The callback (func) will be invoked with the following parameters:

key the QueueKey for this acknowledgement.

message A modem::Message containing the contents of the original message that was acknowledged.

Definition at line 229 of file queue_manager.h.

E.10.3.10 void queue::QueueManager::set_data_on_demand_cb (MsgFunc2 *func*) [inline]

Set the callback to process queues marked `on_demand` by QueueManager::set_on_demand.

This method forwards a data request from the modem to the application level for applications demanding to wait on encoding until the moment data is required (highly time sensitive data). Note that in most circumstances it is better to fill the queues asynchronously and let the QueueManager take care of this transaction.

Parameters

func Pointer to function (or other boost::function object) of the signature queue::MsgFunc2. The callback (func) will be invoked with the following parameters:

key the QueueKey for the data request.

message1 (incoming) The modem::Message containing the details of the request (source, destination, size, etc.)

message2 (outgoing) The modem::Message to be sent. This should be populated by the callback.

Definition at line 238 of file queue_manager.h.

E.10.3.11 void queue::QueueManager::set_expire_cb (MsgFunc1 *func*) [inline]

Set the callback to receive messages every time a message is expired due to exceeding its time to live (ttl).

Parameters

func Pointer to function (or any other object boost::function accepts) matching the signature of queue::MsgFunc1. The callback (func) will be invoked with the following parameters:

key the QueueKey for the queue from which the expired message originated.

message A modem::Message containing the contents of the original message that was expired.

Definition at line 253 of file queue_manager.h.

E.10.3.12 void queue::QueueManager::set_modem_id (unsigned *modem_id*) [inline]

Set the modem id for this vehicle.

Parameters

modem_id unique (within a network) number representing the modem on this vehicle.

Definition at line 130 of file queue_manager.h.

E.10.3.13 void queue::QueueManager::set_on_demand (unsigned *id*, QueueType *type* = queue_dccl)

Set a queue to call the data_on_demand callback every time data is request (basically forwards the modem data_request).

Parameters

id DCCL message id (

```
<id/>
```

) that references the queue for which to enable the `on demand` callback.

Definition at line 183 of file queue_manager.cpp.

E.10.3.14 void queue::QueueManager::set_on_demand (QueueKey *key*)

Set a queue to call the data_on_demand callback every time data is request (basically forwards the modem data_request).

Parameters

key QueueKey that references the queue for which to enable the `on demand` callback.

Definition at line 171 of file queue_manager.cpp.

E.10.3.15 void queue::QueueManager::set_queue_size_change_cb (QSizeFunc *func*) [inline]

Set the callback to receive messages every time a queue changes size (that is, a message is popped or pushed).

Parameters

func Pointer to function (or any other object boost::function accepts) matching the signature of queue::QSizeFunc. The callback (func) will be provided with the following parameters:

key the QueueKey for queue that changed size.

size the size of the queue after the size change event.

Definition at line 245 of file queue_manager.h.

E.10.3.16 void queue::QueueManager::set_receive_cb (MsgFunc1 *func*) [inline]

Set the callback to receive incoming DCCL messages. Any messages received before this callback is set will be discarded.

Parameters

func Pointer to function (or any other object boost::function accepts) matching the signature of queue::MsgFunc1. The callback (func) will be invoked with the following parameters:

key the QueueKey for the queue for which this received message belongs.

message A modem::Message reference containing the contents of the received DCCL message.

Definition at line 167 of file queue_manager.h.

E.10.3.17 void queue::QueueManager::set_receive_ccl_cb (MsgFunc1 *func*) [inline]

Set the callback to receive incoming CCL messages. Any messages received before this callback is set will be discarded.

Parameters

func Pointer to function (or any other object boost::function accepts) matching the signature of queue::MsgFunc1. The callback (func) will be invoked with the following parameters:

key the QueueKey for the queue for which this received message belongs.

message A modem::Message reference containing the contents of the received CCL message.

Definition at line 175 of file queue_manager.h.

E.10.3.18 void queue::QueueManager::set_schema (const std::string *schema*) [inline]

Set the schema used for XML syntax checking.

Schema location is relative to the XML file location! if you have XML files in different places you must pass the proper relative path (or just use absolute paths)

Parameters

schema location of the message_schema.xsd file.

Definition at line 125 of file queue_manager.h.

E.10.3.19 std::string queue::QueueManager::summary () const

Returns

human readable summary of all loaded queues

Definition at line 187 of file queue_manager.cpp.

The documentation for this class was generated from the following files:

- /var/www/root/goby-bzr/src/acomms/libqueue/queue_manager.h
- /var/www/root/goby-bzr/src/acomms/libqueue/queue_manager.cpp

E.11 amac::Slot Class Reference

Represents a slot of the TDMA cycle.

`#include <mac_manager.h>`

Public Types

- enum SlotType { slot_notype, slot_data, slot_ping }
  Enumeration of slot types.

Public Member Functions

Constructors/Destructor

- Slot (unsigned src=0, unsigned dest=0, int rate=0, SlotType type=slot_notype, unsigned slot_-time=15, boost::posix_time::ptime last_heard_time=boost::posix_time::not_a_date_time)
  Default constructor.

Set

- void **set_src** (unsigned src)
- void **set_dest** (int dest)
- void **set_rate** (int rate)
- void **set_type** (SlotType type)
- void **set_last_heard_time** (boost::posix_time::ptime t)
- void **set_slot_time** (unsigned t)

Get

- unsigned **src** () const
- int **dest** () const
- int **rate** () const
- SlotType **type** () const
- std::string **type_as_string** () const
- boost::posix_time::ptime **last_heard_time** () const
- unsigned **slot_time** () const

E.11.1 Detailed Description

Represents a slot of the TDMA cycle.

Definition at line 60 of file mac_manager.h.

E.11.2 Member Enumeration Documentation

E.11.2.1 enum amac::Slot::SlotType

Enumeration of slot types.

Enumerator:

slot_notype useless slot

slot_data slot to send data packet

slot_ping slot to send ping (ranging)

Definition at line 64 of file mac_manager.h.

E.11.3 Constructor & Destructor Documentation

E.11.3.1 amac::Slot::Slot (unsigned *src* = `0`, unsigned *dest* = `0`, int *rate* = `0`, SlotType *type* = `slot_notype`, unsigned *slot_time* = `15`, boost::posix_time::ptime *last_heard_time* = `boost::posix_time::not_a_date_time`) `[inline]`

Default constructor.

Parameters

src id representing sender of the data packet or ping

dest id representing destination of the data packet or ping

rate value 0-5 representing modem transmission rate. 0 is slowest, 5 is fastest

type amac::Slot::SlotType of this slot

slot_time length of time this slot should last in seconds

last_heard_time last time (in seconds since 1/1/70) the src vehicle was heard from

Definition at line 81 of file mac_manager.h.

The documentation for this class was generated from the following file:

- /var/www/root/goby-bzr/src/acomms/libamac/mac_manager.h

Appendix F

Example Documentation

F.1 acomms/examples/chat/chat.cpp

chat.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <name>Chat</name>
    <id>1</id>
    <size>32</size>
    <queuing>
      <ack>true</ack>
      <newest_first>false</newest_first>
    </queuing>
    <layout>
      <string>
        <name>message</name>
        <max_length>26</max_length>
      </string>
    </layout>
  </message>
</message_set>
```

chat.cpp

```
// copyright 2009 t. schneider tes@mit.edu
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This software is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this software.  If not, see <http://www.gnu.org/licenses/>.

// usage: connect two modems and then run
// > chat /dev/tty_modem_A 1 2 log_file_A
// > chat /dev/tty_modem_A 2 1 log_file_B
```

```
// type into a window and hit enter to send a message. messages will be queued
// and sent on a rotating cycle every 15 seconds

#include <iostream>

#include "acomms/dccl.h"
#include "acomms/queue.h"
#include "acomms/modem_driver.h"
#include "acomms/amac.h"
#include "acomms/modem_message.h"
#include "acomms/bind.h"

#include <boost/lexical_cast.hpp>

#include "chat_curses.h"

int startup_failure();
void received_data(queue::QueueKey, const modem::Message&);
void received_ack(queue::QueueKey, const modem::Message&);
std::string decode_received(unsigned id, const std::string& data);

std::ofstream fout_;
dccl::DCCLCodec dccl_;
queue::QueueManager q_manager_(&fout_);
micromodem::MMDriver mm_driver_(&fout_);
amac::MACManager mac_(&fout_);
ChatCurses curses_;


int main(int argc, char* argv[])
{
    //
    // 1. Deal with command line parameters
    //

    if(argc != 5) return startup_failure();

    std::string serial_port = argv[1];
    unsigned my_id;
    unsigned buddy_id;
    try
    {
        my_id = boost::lexical_cast<unsigned>(argv[2]);
        buddy_id = boost::lexical_cast<unsigned>(argv[3]);
    }
    catch(boost::bad_lexical_cast&)
    {
        std::cerr << "bad value for my_id: " << argv[2] << " or buddy_id: " << ar
      gv[3] << ". these must be unsigned integers." << std::endl;
        return startup_failure();
    }

    std::string log_file = argv[4];
    fout_.open(log_file.c_str());
    if(!fout_.is_open())
    {
        std::cerr << "bad value for log_file: " << log_file << std::endl;
        return startup_failure();
    }

    // bind the callbacks of these libraries
    acomms_util::bind(mm_driver_, q_manager_, mac_);

    //
    // Initiate DCCL (libdccl)
    //
    dccl_.add_xml_message_file(ACOMMS_EXAMPLES_DIR "/chat/chat.xml", "../../libdc
```

```
  cl/message_schema.xsd");

//
// Initiate queue manager (libqueue)
//
q_manager_.set_modem_id(my_id);
q_manager_.set_receive_cb(&received_data);
q_manager_.set_ack_cb(&received_ack);
q_manager_.add_xml_queue_file(ACOMMS_EXAMPLES_DIR "/chat/chat.xml", "../../li
  bdccl/message_schema.xsd");
//
// Initiate modem driver (libmodemdriver)
//
mm_driver_.set_serial_port(serial_port);
mm_driver_.set_cfg(std::vector<std::string>(1, std::string("SRC," + boost::le
  xical_cast<std::string>(my_id))));

//
// Initiate medium access control (libamac)
//
mac_.set_type(amac::mac_slotted_tdma);
mac_.set_rate(0);
mac_.set_slot_time(15);
mac_.set_expire_cycles(5);
mac_.set_modem_id(my_id);

//
// Start up everything
//
try
{
    mac_.startup();
    mm_driver_.startup();
}
catch(std::runtime_error& e)
{
    std::cerr << "exception at startup: " << e.what() << std::endl;
    return startup_failure();
}

curses_.set_modem_id(my_id);
curses_.startup();

//
// Loop until terminated (CTRL-C)
//
for(;;)
{
    std::string line;
    curses_.run_input(line);

    if(!line.empty())
    {
        std::map<std::string, dccl::MessageVal> vals;
        vals["message"] = line;

        std::string hex_out;

        unsigned message_id = 1;
        dccl_.encode(message_id, hex_out, vals);

        modem::Message message_out;
        message_out.set_data(hex_out);
        // send this message to my buddy!
        message_out.set_dest(buddy_id);

        q_manager_.push_message(message_id, message_out);
```

```
        }

        try
        {
            mm_driver_.do_work();
            mac_.do_work();
        }
        catch(std::runtime_error& e)
        {
            curses_.cleanup();
            std::cerr << "exception while running: " << e.what() << std::endl;
            return 1;
        }
    }

    return 0;
}

int startup_failure()
{
    std::cerr << "usage: chat /dev/tty_modem my_id buddy_id log_file" << std::end
      l;
    return 1;
}

void received_data(queue::QueueKey key, const modem::Message& message_in)
{
    curses_.post_message(message_in.src(), decode_received(key.id(), message_in.
      data()));
}

void received_ack(queue::QueueKey key, const modem::Message& ack_message)
{

    curses_.post_message
        (ack_message.dest(),
         std::string("received message starting with: "
                     + decode_received(key.id(), ack_message.data()).substr(0,5))
      );
}

std::string decode_received(unsigned id, const std::string& data)
{
    std::map<std::string, dccl::MessageVal> vals;
    dccl_.decode(id, data, vals);
    return vals["message"];
}
```

F.2 libamac/examples/amac_simple/amac_simple.cpp

amac_simple.cpp

```
// copyright 2009 t. schneider tes@mit.edu
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This software is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
```

```
// You should have received a copy of the GNU General Public License
// along with this software.  If not, see <http://www.gnu.org/licenses/>.

#include "acomms/amac.h"
#include "acomms/modem_message.h"
#include <iostream>

int next_dest(unsigned);
void init_transmission(const modem::Message&);

int main(int argc, char* argv[])
{

    //
    // 1. Create a MACManager and feed it a std::ostream to log to
    //
    amac::MACManager mac(&std::cout);

    //
    // 2. Configure it for TDMA with basic peer discovery, rate 0, 10 second slot
      s, and expire vehicles after 2 cycles of no communications. also, we are modem id
       1.
    //
    mac.set_type(amac::mac_slotted_tdma);
    mac.set_rate(0);
    mac.set_slot_time(10);
    mac.set_expire_cycles(2);
    mac.set_modem_id(1);

    //
    // 3. Set up the callbacks
    //

    // give callback for the next destination. this is called before a cycle is i
      nitiated, and would be bound to queue::QueueManager::request_next_destination if
      using libqueue.
    mac.set_destination_cb(&next_dest);
    // give a callback to use for actually initiating the transmission. this woul
      d be bound to modem::DriverBase::initiate_transmission if using libmodemdriver.
    mac.set_initiate_transmission_cb(&init_transmission);

    //
    // 4. Let it run for a bit alone in the world
    //
    mac.startup();
    for(unsigned i = 1; i < 60; ++i)
    {
        mac.do_work();
        sleep(1);
    }

    //
    // 5. Discover some friends (modem ids 2 & 3)
    //

    mac.process_message(modem::Message("src=2"));
    mac.process_message(modem::Message("src=3"));

    //
    // 6. Run it, hearing consistently from #3, but #2 has gone silent and will b
      e expired after 2 cycles
    //

    for(;;)
    {
        mac.do_work();
        sleep(1);
```

```
        mac.process_message(modem::Message("src=3"));
    }

    return 0;
}

int next_dest(unsigned size)
{
    // let's pretend we always have a message to send to vehicle 10
    return 10;
}

void init_transmission(const modem::Message& init_message)
{
    std::cout << "starting transmission with these values: " << init_message << s
      td::endl;
}
```

F.3 libdccl/examples/dccl_simple/dccl_simple.cpp

simple.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <name>Simple</name>
    <id>1</id>
    <size>32</size>
    <layout>
      <string>
        <name>s_key</name>
        <max_length>26</max_length>
      </string>
    </layout>
  </message>
</message_set>
```

dccl_simple.cpp

```
// copyright 2009 t. schneider tes@mit.edu
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This software is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this software.  If not, see <http://www.gnu.org/licenses/>.


// encodes/decodes a string using the DCCL codec library
// assumes prior knowledge of the message format (required fields)

#include "acomms/dccl.h"
#include <iostream>

using dccl::operator<<;
```

```
int main()
{
    // initialize input contents to encoder. since
    // simple.xml only has a <string/> message var
    // we only need one map.
    std::map<std::string, dccl::MessageVal> val_map;

    //  initialize output hexadecimal
    std::string hex;

    std::cout << "loading xml file: xml/simple.xml" << std::endl;

    // instantiate the parser with a single xml file (simple.xml).
    // also pass the schema, relative to simple.xml, for XML validity
    // checking (syntax).
    dccl::DCCLCodec dccl(DCCL_EXAMPLES_DIR "/dccl_simple/simple.xml",
                         "../../message_schema.xsd");


    // read message content (in this case from the command line)
    std::string input;
    std::cout << "input a string to send: " << std::endl;
    getline(std::cin, input);

    // key is <name/> child for a given message_var
    // (i.e. child of <int/>, <string/>, etc)
    val_map["s_key"] = input;

    std::cout << "passing values to encoder:\n" << val_map;

    // we are encoding for message id 1 - given in simple.xml
    unsigned id = 1;
    dccl.encode(id, hex, val_map);

    std::cout << "received hexadecimal string: " << hex << std::endl;

    val_map.clear();

    // input contents right back to decoder
    std::cout << "passed hexadecimal string to decoder: " << hex << std::endl;

    dccl.decode(id, hex, val_map);

    std::cout << "received values:" << std::endl
              << val_map;

    // now you can use it...
    std::string output = val_map["s_key"];

    return 0;
}
```

F.4 libdccl/examples/plusnet/plusnet.cpp

nafcon_command.xml

```
<?xml version="1.0" encoding="UTF8"?>
<message_set>
  <message>
    <name>SENSOR_DEPLOY</name>
    <destination_var key="DestinationPlatformId">PLUSNET_MESSAGES</destination_var>
    <trigger>publish</trigger>
    <trigger_var mandatory_content="MessageType=SENSOR_DEPLOY">
```

```
    PLUSNET_MESSAGES
  </trigger_var>
  <size>32</size>
  <header>
    <id>10</id>
    <time>
      <name>Timestamp</name>
    </time>
    <src_id>
      <name>SourcePlatformId</name>
    </src_id>
    <dest_id>
      <name>DestinationPlatformId</name>
    </dest_id>
  </header>
  <layout>
    <static>
      <name>MessageType</name>
      <value>SENSOR_DEPLOY</value>
    </static>
    <static>
      <name>SensorCommandType</name>
      <value>0</value>
    </static>
    <float>
      <name>DeployLatitude</name>
      <precision>5</precision>
      <max>90</max>
      <min>-90</min>
    </float>
    <float>
      <name>DeployLongitude</name>
      <precision>5</precision>
      <max>180</max>
      <min>-180</min>
    </float>
    <int>
      <name>DeployDepth</name>
      <max>254</max>
      <min>0</min>
    </int>
    <int>
      <name>DeployDuration</name>
      <max>62</max>
      <min>1</min>
    </int>
    <float>
      <name>AbortLatitude</name>
      <precision>4</precision>
      <max>90</max>
      <min>-90</min>
    </float>
    <float>
      <name>AbortLongitude</name>
      <precision>4</precision>
      <max>180</max>
      <min>-180</min>
    </float>
    <int>
      <name>AbortDepth</name>
      <max>510</max>
      <min>0</min>
    </int>
    <int>
      <name>MissionType</name>
      <max>6</max>
      <min>0</min>
```

```
      </int>
      <int>
        <name>OperatingRadius</name>
        <max>1022</max>
        <min>0</min>
      </int>
      <float algorithm="angle_0_360">
        <name>DCLFOVStartHeading</name>
        <max>360</max>
        <min>0</min>
        <precision>1</precision>
      </float>
      <float algorithm="angle_0_360">
        <name>DCLFOVEndHeading</name>
        <max>360</max>
        <min>0</min>
        <precision>1</precision>
      </float>
      <float>
        <name>DCLSearchRate</name>
        <max>254</max>
        <min>0</min>
        <precision>1</precision>
      </float>
    </layout>
    <on_receipt>
      <publish>
        <publish_var>NAFCON_MESSAGES</publish_var>
        <all />
      </publish>
    </on_receipt>
  </message>
  <message>
    <name>SENSOR_PROSECUTE</name>
    <destination_var key="DestinationPlatformId">PLUSNET_MESSAGES</destination_var>
    <trigger>publish</trigger>
    <trigger_var mandatory_content="MessageType=SENSOR_PROSECUTE">
      PLUSNET_MESSAGES
    </trigger_var>
    <size>32</size>
    <header>
      <id>11</id>
      <time>
        <name>TargetTimestamp</name>
      </time>
      <src_id>
        <name>SourcePlatformId</name>
      </src_id>
      <dest_id>
        <name>DestinationPlatformId</name>
      </dest_id>
    </header>
    <layout>
      <static>
        <name>MessageType</name>
        <value>SENSOR_PROSECUTE</value>
      </static>
      <int>
        <name>SensorCommandType</name>
        <min>1</min>
        <max>4</max>
      </int>
      <int>
        <name>PlatformID</name>
        <max>30</max>
        <min>0</min>
      </int>
```

```
<int>
  <name>TrackNumber</name>
  <max>254</max>
  <min>0</min>
</int>
<float>
  <name>TargetLatitude</name>
  <precision>5</precision>
  <max>90</max>
  <min>-90</min>
</float>
<float>
  <name>TargetLongitude</name>
  <precision>5</precision>
  <max>180</max>
  <min>-180</min>
</float>
<int>
  <name>TargetDepth</name>
  <max>1022</max>
  <min>0</min>
</int>
<float algorithm="angle_0_360">
  <name>TargetHeading</name>
  <max>360</max>
  <min>0</min>
  <precision>1</precision>
</float>
<float>
  <name>TargetSpeed</name>
  <max>12.6</max>
  <min>0</min>
  <precision>1</precision>
</float>
<int>
  <name>TargetSpectralLevel1</name>
  <max>126</max>
  <min>0</min>
</int>
<int>
  <name>TargetFrequency1</name>
  <max>4094</max>
  <min>0</min>
</int>
<int>
  <name>TargetBandwidth1</name>
  <max>20</max>
  <min>0</min>
</int>
<int>
  <name>TargetSpectralLevel2</name>
  <max>126</max>
  <min>0</min>
</int>
<int>
  <name>TargetFrequency2</name>
  <max>4094</max>
  <min>0</min>
</int>
<int>
  <name>TargetBandwidth2</name>
  <max>20</max>
  <min>0</min>
</int>
<int>
  <name>ProsecuteDuration</name>
  <max>62</max>
```

```
        <min>1</min>
      </int>
      <float>
        <name>AbortLatitude</name>
        <precision>4</precision>
        <max>90</max>
        <min>-90</min>
      </float>
      <float>
        <name>AbortLongitude</name>
        <precision>4</precision>
        <max>180</max>
        <min>-180</min>
      </float>
      <int>
        <name>AbortDepth</name>
        <max>126</max>
        <min>0</min>
      </int>
    </layout>
    <on_receipt>
      <publish>
        <publish_var>NAFCON_MESSAGES</publish_var>
        <all />
      </publish>
    </on_receipt>
  </message>
</message_set>
```

nafcon_report.xml

```
<?xml version="1.0" encoding="UTF8" ?>
<message_set>
  <message>
    <name>SENSOR_STATUS</name>
    <trigger>publish</trigger> <!-- package upon publish to some moos var -->
    <trigger_moos_var mandatory_content="MessageType=SENSOR_STATUS">
      PLUSNET_MESSAGES
    </trigger_moos_var>
    <size>32</size>
    <header>
      <id>12</id>
      <time>
        <name>Timestamp</name>
      </time>
      <src_id>
        <name>SourcePlatformId</name>
      </src_id>
      <dest_id>
        <name>DestinationPlatformId</name>
      </dest_id>
    </header>
    <layout>
      <static>
        <name>MessageType</name>
        <value>SENSOR_STATUS</value>
      </static>
      <static>
        <name>SensorReportType</name>
        <value>0</value>
      </static>
      <float>
        <name>NodeLatitude</name>
        <precision>5</precision>
        <max>90</max>
        <min>-90</min>
      </float>
```

```
<float>
  <name>NodeLongitude</name>
  <precision>5</precision>
  <max>180</max>
  <min>-180</min>
</float>
<int>
  <name>NodeDepth</name>
  <max>1022</max>
  <min>0</min>
</int>
<int>
  <name>NodeCEP</name>
  <max>1022</max>
  <min>0</min>
</int>
<float algorithm="angle_0_360">
  <name>NodeHeading</name>
  <max>360</max>
  <min>0</min>
  <precision>1</precision>
</float>
<float>
  <name>NodeSpeed</name>
  <max>12.6</max>
  <min>0</min>
  <precision>1</precision>
</float>
<int>
  <name>MissionState</name>
  <max>6</max>
  <min>0</min>
</int>
<int>
  <name>MissionType</name>
  <max>6</max>
  <min>0</min>
</int>
<int>
  <name>LastGPSTimestamp</name>
  <max>2000000000</max>
  <min>1000000000</min>
</int>
<int>
  <name>PowerLife</name>
  <max>1022</max>
  <min>1</min>
</int>
<int>
  <name>SensorHealth</name>
  <max>14</max>
  <min>0</min>
</int>
<int>
  <name>RecorderState</name>
  <max>1</max>
  <min>0</min>
</int>
<int>
  <name>RecorderLife</name>
  <max>30</max>
  <min>0</min>
</int>
<int>
  <name>NodeSpecificInfo0</name>
  <max>254</max>
  <min>0</min>
```

```
      </int>
      <int>
        <name>NodeSpecificInfo1</name>
        <max>254</max>
        <min>0</min>
      </int>
      <int>
        <name>NodeSpecificInfo2</name>
        <max>254</max>
        <min>0</min>
      </int>
      <int>
        <name>NodeSpecificInfo3</name>
        <max>254</max>
        <min>0</min>
      </int>
      <int>
        <name>NodeSpecificInfo4</name>
        <max>254</max>
        <min>0</min>
      </int>
      <int>
        <name>NodeSpecificInfo5</name>
        <max>254</max>
        <min>0</min>
      </int>
    </layout>
    <on_receipt>
      <publish>
        <moos_var>NAFCON_MESSAGES</moos_var>
        <all />
      </publish>
    </on_receipt>
  </message>
  <message>
    <name>SENSOR_CONTACT</name>
    <trigger>publish</trigger>
    <trigger_moos_var mandatory_content="MessageType=SENSOR_CONTACT">
      PLUSNET_MESSAGES
    </trigger_moos_var>
    <size>32</size>
    <header>
      <id>13</id>
      <time>
        <name>ContactTimestamp</name>
      </time>
      <src_id>
        <name>SourcePlatformId</name>
      </src_id>
      <dest_id>
        <name>DestinationPlatformId</name>
      </dest_id>
    </header>
    <layout>
      <static>
        <name>MessageType</name>
        <value>SENSOR_CONTACT</value>
      </static>
      <static>
        <name>SensorReportType</name>
        <value>1</value>
      </static>
      <float algorithm="angle_0_360">
        <name>SensorHeading</name>
        <max>360</max>
        <min>0</min>
        <precision>1</precision>
```

```
</float>
<float>
  <name>SensorPitch</name>
  <max>90</max>
  <min>-90</min>
  <precision>0</precision>
</float>
<float>
  <name>SensorRoll</name>
  <max>180</max>
  <min>-180</min>
  <precision>0</precision>
</float>
<float>
  <name>SensorLatitude</name>
  <precision>5</precision>
  <max>90</max>
  <min>-90</min>
</float>
<float>
  <name>SensorLongitude</name>
  <precision>5</precision>
  <max>180</max>
  <min>-180</min>
</float>
<int>
  <name>SensorDepth</name>
  <max>1022</max>
  <min>0</min>
</int>
<int>
  <name>SensorCEP</name>
  <max>1022</max>
  <min>0</min>
</int>
<float>
  <name>ContactBearing</name>
  <max>360</max>
  <min>0</min>
  <precision>1</precision>
</float>
<float>
  <name>ContactBearingUncertainty</name>
  <max>10</max>
  <min>0</min>
  <precision>1</precision>
</float>
<float>
  <name>ContactBearingRate</name>
  <max>10</max>
  <min>-10</min>
  <precision>1</precision>
</float>
<float>
  <name>ContactBearingRateUncertainty</name>
  <max>3.0</max>
  <min>0</min>
  <precision>1</precision>
</float>
<float>
  <name>ContactElevation</name>
  <max>90</max>
  <min>-90</min>
  <precision>1</precision>
</float>
<float>
  <name>ContactElevationUncertainty</name>
```

```
      <max>10</max>
      <min>0</min>
      <precision>1</precision>
    </float>
    <int>
      <name>ContactSpectralLevel1</name>
      <max>126</max>
      <min>0</min>
    </int>
    <int>
      <name>ContactFrequency1</name>
      <max>4094</max>
      <min>0</min>
    </int>
    <int>
      <name>ContactBandwidth1</name>
      <max>20</max>
      <min>0</min>
    </int>
    <int>
      <name>ContactSpectralLevel2</name>
      <max>126</max>
      <min>0</min>
    </int>
    <int>
      <name>ContactFrequency2</name>
      <max>4094</max>
      <min>0</min>
    </int>
    <int>
      <name>ContactBandwidth2</name>
      <max>20</max>
      <min>0</min>
    </int>
  </layout>
  <on_receipt>
    <publish>
      <moos_var>NAFCON_MESSAGES</moos_var>
      <all />
    </publish>
  </on_receipt>
</message>
<message>
  <name>SENSOR_TRACK</name>
  <trigger>publish</trigger>
  <trigger_moos_var mandatory_content="MessageType=SENSOR_TRACK">
    PLUSNET_MESSAGES
  </trigger_moos_var>
  <size>32</size>

  <header>
    <id>14</id>
    <time>
      <name>TrackTimestamp</name>
    </time>
    <src_id>
      <name>SourcePlatformId</name>
    </src_id>
    <dest_id>
      <name>DestinationPlatformId</name>
    </dest_id>
  </header>

  <layout>
    <static>
      <name>MessageType</name>
      <value>SENSOR_TRACK</value>
```

```
</static>
<static>
  <name>SensorReportType</name>
  <value>2</value>
</static>
<int>
  <name>PlatformID</name>
  <max>30</max>
  <min>0</min>
</int>
<int>
  <name>TrackNumber</name>
  <max>254</max>
  <min>0</min>
</int>
<float>
  <name>TrackLatitude</name>
  <precision>5</precision>
  <max>90</max>
  <min>-90</min>
</float>
<float>
  <name>TrackLongitude</name>
  <precision>5</precision>
  <max>180</max>
  <min>-180</min>
</float>
<int>
  <name>TrackCEP</name>
  <max>1022</max>
  <min>0</min>
</int>
<int>
  <name>TrackDepth</name>
  <max>1022</max>
  <min>0</min>
</int>
<int>
  <name>TrackDepthUncertainty</name>
  <max>62</max>
  <min>0</min>
</int>
<float algorithm="angle_0_360">
  <name>TrackHeading</name>
  <max>360</max>
  <min>0</min>
  <precision>1</precision>
</float>
<float>
  <name>TrackHeadingUncertainty</name>
  <max>10</max>
  <min>0</min>
  <precision>1</precision>
</float>
<float>
  <name>TrackSpeed</name>
  <max>12.6</max>
  <min>0</min>
  <precision>1</precision>
</float>
<float>
  <name>TrackSpeedUncertainty</name>
  <max>3</max>
  <min>0</min>
  <precision>1</precision>
</float>
<int>
```

```
        <name>DepthClassification</name>
        <max>6</max>
        <min>0</min>
      </int>
      <int>
        <name>TrackClassification</name>
        <max>6</max>
        <min>0</min>
      </int>
      <int>
        <name>TrackSpectralLevel1</name>
        <max>126</max>
        <min>0</min>
      </int>
      <int>
        <name>TrackFrequency1</name>
        <max>4094</max>
        <min>0</min>
      </int>
      <int>
        <name>TrackBandwidth1</name>
        <max>20</max>
        <min>0</min>
      </int>
      <int>
        <name>TrackSpectralLevel2</name>
        <max>126</max>
        <min>0</min>
      </int>
      <int>
        <name>TrackFrequency2</name>
        <max>4094</max>
        <min>0</min>
      </int>
      <int>
        <name>TrackBandwidth2</name>
        <max>20</max>
        <min>0</min>
      </int>
    </layout>
    <on_receipt>
      <publish>
        <moos_var>NAFCON_MESSAGES</moos_var>
        <all />
      </publish>
    </on_receipt>
  </message>
</message_set>
```

plusnet.cpp

```
// t. schneider tes@mit.edu 3.31.09
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This software is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this software.  If not, see <http://www.gnu.org/licenses/>.
```

```
// encodes/decodes the message given in the pGeneralCodec documentation
// also includes the simple.xml file to show example of DCCLCodec instantiation
// with multiple files

// this is an example showing some of the "MOOS" related features of libdccl that
       can be used (if desired) in the absence of MOOS

#include "acomms/dccl.h"

#include <exception>
#include <iostream>

using dccl::operator<<;

int main()
{
    std::cout << "loading nafcon xml files" << std::endl;

    dccl::DCCLCodec dccl;
    dccl.add_xml_message_file(DCCL_EXAMPLES_DIR "/plusnet/nafcon_command.xml", ".
      ./../message_schema.xsd");
    dccl.add_xml_message_file(DCCL_EXAMPLES_DIR "/plusnet/nafcon_report.xml", "..
      /../message_schema.xsd");

    std::cout << std::string(30, '#') << std::endl
              << "detailed message summary:" << std::endl
              << std::string(30, '#') << std::endl
              << dccl;

    std::cout << std::string(30, '#') << std::endl
              << "ENCODE / DECODE example:" << std::endl
              << std::string(30, '#') << std::endl;

    // initialize input contents to encoder
    std::map<std::string, dccl::MessageVal> in_vals;

    // initialize output message
    modem::Message msg;

    in_vals["PLUSNET_MESSAGES"] = "MessageType=SENSOR_STATUS,SensorReportType=0,S
      ourcePlatformId=1,DestinationPlatformId=3,Timestamp=1191947446.91117,NodeLatitude
      =47.7448,NodeLongitude=-122.845,NodeDepth=0.26,NodeCEP=0,NodeHeading=169.06,NodeS
      peed=0,MissionState=2,MissionType=2,LastGPSTimestamp=1191947440,PowerLife=6,Senso
      rHealth=0,RecorderState=1,RecorderLife=0,NodeSpecificInfo0=0,NodeSpecificInfo1=0,
      NodeSpecificInfo2=23,NodeSpecificInfo3=0,NodeSpecificInfo4=3,NodeSpecificInfo5=0"
      ;

    std::cout << "passing values to encoder:" << std::endl
              << in_vals;

    dccl.pubsub_encode("SENSOR_STATUS", msg, in_vals);

    std::cout << "received dccl::Message: "
              << msg.serialize()
              << std::endl;


    std::multimap<std::string, dccl::MessageVal> out_vals;


    std::cout << "passed dccl::Message to decoder: "
              << msg.serialize()
              << std::endl;

    dccl.pubsub_decode("SENSOR_STATUS", msg, out_vals);

    std::cout << "received values:" << std::endl
```

```
              << out_vals;

    return 0;
}
```

F.5 libdccl/examples/test/test.cpp

test.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <name>TEST</name>
    <size>256</size>
    <id>4</id>
    <repeat>5</repeat>
    <layout>
      <static>
        <name>Stat</name>
        <value>hi</value>
      </static>

      <float algorithm="sum:F:I">
        <name>SUM</name>
        <max>200</max>
        <min>-100</min>
        <precision>2</precision>

      </float>

      <float algorithm="*2">
        <name>F</name>
        <max>100</max>
        <min>-50</min>
        <precision>2</precision>
      </float>

      <bool algorithm="invert">
        <name>B</name>

      </bool>

      <enum>
        <name>E</name>
        <value>cat</value>
        <value>dog</value>
        <value>mouse</value>
        <value>emu</value>

      </enum>

      <string algorithm="prepend_fat">
        <name>S</name>
        <max_length>15</max_length>

      </string>

      <float>
        <name>my_NAN</name>
        <max>100</max>
        <min>-50</min>
        <precision>0</precision>
```

```
      </float>

      <int algorithm="+1">
        <name>I</name>
        <max>100</max>
        <min>-50</min>
        <max_delta>10</max_delta>
      </int>

<!--      <hex>
        <name>H</name>
        <num_bytes>4</num_bytes>
      </hex> -->

    </layout>
  </message>
</message_set>
```

test.cpp

```
// t. schneider tes@mit.edu 11.20.09
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This software is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this software.  If not, see <http://www.gnu.org/licenses/>.


#include "acomms/dccl.h"
#include <iostream>
#include <cassert>

using dccl::operator<<;

void plus1(dccl::MessageVal& mv)
{
    long l = mv;
    ++l;
    mv = l;
}

void times2(dccl::MessageVal& mv)
{
    double d = mv;
    d *= 2;
    mv = d;
}

void prepend_fat(dccl::MessageVal& mv)
{
    std::string s = mv;
    s = "fat_" + s;
    mv = s;
}

void invert(dccl::MessageVal& mv)
{
    bool b = mv;
    b ^= 1;
```

```
    mv = b;
}

void algsum(dccl::MessageVal& mv, const std::vector<dccl::MessageVal>& ref_vals)
{
    double d = 0;
    // index 0 is the name ("sum"), so start at 1
    for(size_t i = 0, n = ref_vals.size(); i < n; ++i)
    {
        d += double(ref_vals[i]);
    }
    mv = d;
}


int main()
{
    std::cout << "loading xml file: test.xml" << std::endl;

    // instantiate the parser with a single xml file
    dccl::DCCLCodec dccl(DCCL_EXAMPLES_DIR "/test/test.xml", "../../message_schem
      a.xsd");

    std::cout << dccl << std::endl;

    // load up the algorithms
    dccl.add_algorithm("prepend_fat", &prepend_fat);
    dccl.add_algorithm("+1", &plus1);
    dccl.add_algorithm("*2", &times2);
    dccl.add_algorithm("invert", &invert);
    dccl.add_adv_algorithm("sum", &algsum);

    // must be kept secret!
    dccl.set_crypto_passphrase("my_passphrase!");

    std::map<std::string, std::vector<dccl::MessageVal> > in;

    bool b = true;
    std::vector<dccl::MessageVal> e;
    e.push_back("dog");
    e.push_back("cat");
    e.push_back("emu");

    std::string s = "raccoon";
    std::vector<dccl::MessageVal> i;
    i.push_back(30);
    i.push_back(40);
    std::vector<dccl::MessageVal> f;
    f.push_back(-12.5);
    f.push_back(1);

    std::string h = "abcd1234";
    std::vector<dccl::MessageVal> sum(2,0);


    in["B"] = std::vector<dccl::MessageVal>(1,b);
    in["E"] = e;
    in["S"] = std::vector<dccl::MessageVal>(1,s);
    in["I"] = i;
    in["F"] = f;
    in["H"] = std::vector<dccl::MessageVal>(1,h);
    in["SUM"] = sum;

    std::string hex;
    std::cout << "sent values:" << std::endl
              << in;
```

```
    dccl.encode(4, hex, in);

    std::cout << "hex out: " << hex << std::endl;
    hex.resize(hex.length() + 20,'0');
    std::cout << "hex in: " << hex << std::endl;

    std::map<std::string, std::vector<dccl::MessageVal> > out;

    dccl.decode(4, hex, out);

    std::cout << "received values:" << std::endl
              << out;

    sum[0] = double(i[0]) + double(f[0]);
    sum[1] = double(i[1]) + double(f[1]);
    i[0] = int(i[0]) + 1;
    i[1] = int(i[1]) + 1;

    dccl::MessageVal tmp = b;
    invert(tmp);
    b = tmp;

    tmp = s;
    prepend_fat(tmp);
    tmp.get(s);

    tmp = f[0];
    times2(tmp);
    f[0] = tmp;

    tmp = f[1];
    times2(tmp);
    f[1] = tmp;

    assert(out["B"][0] == b);
    assert(out["E"][0] == e[0]);
    assert(out["E"][1] == e[1]);
    assert(out["E"][2] == e[2]);
    assert(out["S"][0] == s);
    assert(out["F"][0] == f[0]);
    assert(out["F"][1] == f[1]);
    assert(out["SUM"][0] == sum[0]);
    assert(out["SUM"][1] == sum[1]);
    assert(out["I"][0] == i[0]);
    assert(out["I"][1] == i[1]);
    //assert(out["H"][0] == h);

    std::cout << "all tests passed" << std::endl;
}
```

F.6 libdccl/examples/two_message/two_message.cpp

two_message.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <name>GoToCommand</name>
    <id>2</id>
    <size>32</size>
    <header>
      <dest_id>
        <name>destination</name>
      </dest_id>
```

```
    </header>
    <layout>
      <static>
        <name>type</name>
        <value>goto</value>
      </static>
      <int>
        <name>goto_x</name>
        <max>10000</max>
        <min>0</min>
      </int>
      <int>
        <name>goto_y</name>
        <max>10000</max>
        <min>0</min>
      </int>
      <bool>
        <name>lights_on</name>
      </bool>
      <string>
        <name>new_instructions</name>
        <max_length>10</max_length>
      </string>
      <float>
        <name>goto_speed</name>
        <max>3</max>
        <min>0</min>
        <precision>2</precision>
      </float>
    </layout>
  </message>
  <message>
    <name>VehicleStatus</name>
    <id>3</id>
    <size>32</size>
    <layout>
      <float>
        <name>nav_x</name>
        <max>10000</max>
        <min>0</min>
        <precision>1</precision>
      </float>
      <float>
        <name>nav_y</name>
        <max>10000</max>
        <min>0</min>
        <precision>1</precision>
      </float>
      <enum>
        <name>health</name>
        <value>good</value>
        <value>low_battery</value>
        <value>abort</value>
      </enum>
    </layout>
  </message>
</message_set>
```

two_message.cpp

```
// t. schneider tes@mit.edu 3.31.09
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
```

```
// This software is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this software.  If not, see <http://www.gnu.org/licenses/>.


// encodes/decodes the message given in the pGeneralCodec documentation
// also includes the simple.xml file to show example of DCCLCodec instantiation
// with multiple files
#include "acomms/dccl.h"
#include <exception>
#include <iostream>

using dccl::operator<<;


int main()
{
    std::cout << "loading xml files: xml/simple.xml, xml/two_message.xml"
              << std::endl;

    std::set<std::string> xml_files;
    xml_files.insert(DCCL_EXAMPLES_DIR "/dccl_simple/simple.xml");
    xml_files.insert(DCCL_EXAMPLES_DIR "/two_message/two_message.xml");

    dccl::DCCLCodec dccl(xml_files, "../../message_schema.xsd");

    // show some useful information about all the loaded messages
    std::cout << std::string(30, '#') << std::endl
              << "number of messages loaded: " << dccl.message_count() << std::en
      dl
              << std::string(30, '#') << std::endl;

    std::cout << std::string(30, '#') << std::endl
              << "detailed message summary:" << std::endl
              << std::string(30, '#') << std::endl
              << dccl;

    std::cout << std::string(30, '#') << std::endl
              << "brief message summary:" << std::endl
              << std::string(30, '#') << std::endl
              << dccl.brief_summary();

    std::cout << std::string(30, '#') << std::endl
              << "all loaded ids:" << std::endl
              << std::string(30, '#') << std::endl
              << dccl.all_message_ids();

    std::cout << std::string(30, '#') << std::endl
              << "all loaded names:" << std::endl
              << std::string(30, '#') << std::endl
              << dccl.all_message_names();


    std::cout << std::string(30, '#') << std::endl
              << "all required message var names:" << std::endl
              << std::string(30, '#') << std::endl;


    std::set<unsigned> s = dccl.all_message_ids();
    for (std::set<unsigned>::const_iterator it = s.begin(), n = s.end(); it != n;
       ++it)
        std::cout << (*it) << ": " << dccl.message_var_names(*it) << std::endl;
```

```
    std::cout << std::string(30, '#') << std::endl
              << "ENCODE / DECODE example:" << std::endl
              << std::string(30, '#') << std::endl;

    // initialize input contents to encoder
    std::map<std::string, dccl::MessageVal> vals;

    // initialize output hexadecimal
    std::string hex2, hex3;

    // id = 2, name = GoToCommand
    vals["destination"] = 2;
    vals["goto_x"] = 423;
    vals["goto_y"] = 523;
    vals["lights_on"] = true;
    vals["new_instructions"] = "make_toast";
    vals["goto_speed"] = 2.3456;

    // id = 3, name = VehicleStatus
    vals["nav_x"] = 234.5;
    vals["nav_y"] = 451.3;
    vals["health"] = "abort";

    std::cout << "passing values to encoder:" << std::endl
              << vals;

    dccl.encode(2, hex2, vals);
    dccl.encode(3, hex3, vals);

    std::cout << "received hexadecimal string for message 2 (GoToCommand): "
              << hex2
              << std::endl;

    std::cout << "received hexadecimal string for message 3 (VehicleStatus): "
              << hex3
              << std::endl;

    vals.clear();

    std::cout << "passed hexadecimal string for message 2 to decoder: "
              << hex2
              << std::endl;

    std::cout << "passed hexadecimal string for message 3 to decoder: "
              << hex3
              << std::endl;


    dccl.decode(2, hex2, vals);
    dccl.decode(3, hex3, vals);

    std::cout << "received values:" << std::endl
              << vals;

    return 0;
}
```

F.7 libmodemdriver/examples/driver_simple/driver_simple.cpp

driver_simple.cpp

```
// copyright 2009 t. schneider tes@mit.edu
//
```

```
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This software is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this software.  If not, see <http://www.gnu.org/licenses/>.

//
// Usage: run
// > driver_simple /dev/tty_of_modem_A 1
//
// wait a few seconds
//
// > driver_simple /dev/tty_of_modem_B 2
//
// be careful of collisions if you start them at the same time


#include "acomms/modem_driver.h"
#include <iostream>

bool data_request(const modem::Message&, modem::Message&);
void data_receive(const modem::Message&);

int main(int argc, char* argv[])
{
    if(argc != 3)
    {
        std::cout << "usage: driver_simple /dev/tty_of_modem modem_id" << std::en
      dl;
        return 1;
    }

    //
    // 1. Create and initialize the driver we want (currently WHOI Micro-Modem)
    //

    micromodem::MMDriver mm_driver(&std::cout);

    // set the serial port given on the command line
    mm_driver.set_serial_port(argv[1]);

    // set the source id of this modem
    std::string our_id = argv[2];
    std::vector<std::string> cfg(1, std::string("SRC," + our_id));

    mm_driver.set_cfg(cfg);

    // for handling $CADRQ
    mm_driver.set_datarequest_cb(&data_request);
    mm_driver.set_receive_cb(&data_receive);

    //
    // 2. Startup the driver
    //

    mm_driver.startup();

    //
    // 3. Initiate a transmission cycle
    //
```

```
    modem::Message transmit_init_message;
    transmit_init_message.set_src(our_id);
    transmit_init_message.set_dest(acomms::BROADCAST_ID);
    // one frame @ 32 bytes
    transmit_init_message.set_rate(0);

    mm_driver.initiate_transmission(transmit_init_message);

    //
    // 4. Run the driver
    //

    // 10 hz is good
    while(1)
    {
        mm_driver.do_work();

        // in here you can initiate more transmissions as you want
        usleep(100);
    }
    return 0;
}

//
// 5. Handle the data request ($CADRQ)
//

bool data_request(const modem::Message& request_message, modem::Message& message_
      out)
{
    message_out.set_src(request_message.src());
    message_out.set_dest(request_message.dest());
    message_out.set_data("aa1100bbccddef0987654321");

    // we have data
    return true;
}

//
// 6. Post the received data
//

void data_receive(const modem::Message& message_in)
{
    std::cout << "got a message: " << message_in << std::endl;
    std::cout << "\t" << "data: " << message_in.data() << std::endl;
}
```

F.8 libqueue/examples/queue_simple/queue_simple.cpp

simple.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <name>Simple</name>
    <id>1</id>
    <size>32</size>
    <queuing>
      <ack>false</ack>
      <value_base>1</value_base>
      <ttl>1800</ttl>
    </queuing>
```

```
    <!-- used by libdccl alone and not needed for libqueue, but left for reference-->
    <layout>
      <string>
        <name>s_key</name>
        <max_length>24</max_length>
      </string>
    </layout>
    <!-- end used by libdccl -->
  </message>
</message_set>
```

queue_simple.cpp

```
// copyright 2009 t. schneider tes@mit.edu
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This software is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this software.  If not, see <http://www.gnu.org/licenses/>.


// queues a single message from the DCCL library

#include "acomms/queue.h"
#include <iostream>

using queue::operator<<;

void received_data(queue::QueueKey, const modem::Message&);

int main()
{
    //
    //  1. Initialize the QueueManager
    //

    // create a QueueManager for all our queues
    // and at the same time add our message as a DCCL queue
    queue::QueueManager q_manager(QUEUE_EXAMPLES_DIR "/queue_simple/simple.xml",
                                  "../../../libdccl/message_schema.xsd");

    // our modem id (arbitrary)
    unsigned our_id = 1;
    q_manager.set_modem_id(our_id);

    // set up the callback to handle received DCCL messages
    q_manager.set_receive_cb(&received_data);

    // see what our QueueManager contains
    std::cout << q_manager << std::endl;

    //
    //  2. Push a message to a queue
    //

    // let's make a message to store in the queue
    modem::Message app_layer_message_out;
```

```
    // we're making a loopback in this simple example, so make this message's des
      tination our id
    unsigned dest = 1;

    app_layer_message_out.set_dest(dest);
    // typically grab these data from DCCLCodec::encode, but here we'll just ente
      r an example
    // hexadecimal string
    app_layer_message_out.set_data("2000802500006162636431323334");

    // push to queue 1 (which is the Simple message <id/>)
    q_manager.push_message(1, app_layer_message_out);
    std::cout << "pushing message to queue 1: " << app_layer_message_out << std::
      endl;
    std::cout << "\t" << "data: " <<  app_layer_message_out.data() << std::endl;

    //
    //  3. Create a loopback to simulate the Link Layer (libmodemdriver & modem f
      irmware)
    //

    std::cout << "executing loopback (simulating sending a message to ourselves o
      ver the modem link)" << std::endl;

    // pretend the modem is requesting data of up to 32 bytes
    modem::Message data_request_message;
    data_request_message.set_size(32);

    modem::Message link_layer_message_out;
    q_manager.provide_outgoing_modem_data(data_request_message, link_layer_messag
      e_out);

    // we set the incoming message equal to the outgoing message to create the lo
      opback.
    modem::Message link_layer_message_in = link_layer_message_out;


    //
    //  4. Pass the received message to the QueueManager
    //

    q_manager.receive_incoming_modem_data(link_layer_message_in);

    return 0;
}

//
//  5. Do something with the received message
//
void received_data(queue::QueueKey key, const modem::Message& app_layer_message_i
      n)
{
    std::cout << "received message (key is " << key << "): " << app_layer_message
      _in << std::endl;
    std::cout << "\t" << "data: " <<  app_layer_message_in.data() << std::endl;
}
```

Index