Goby Underwater Autonomy Project

User Manual for Version 1.1

<https://launchpad.net/goby>

Contents

Contents 1

1 Introduction 3
- 1.1 What is Goby? 3
- 1.2 Structure of this Manual 3
- 1.3 How to get help 3

2 Goby MOOS Modules 5
- 2.1 Unified Command and Control for Subsea Autonomous Sensing Networks 5
- 2.2 Overview of the LAMSS Communication Stack 7
- 2.3 pAcommsHandler 10
- 2.4 iCommander 53
- 2.5 pREMUSCodec 59
- 2.6 iMOOS2SQL 62
- 2.7 pGeneralCodec 62
- 2.8 pBTRCodec 62
- 2.9 pCTDCodec 63
- 2.10 pAcommsPoller 63

3 What's next 64

Glossary 65

Bibliography 66

1 Introduction

1.1 What is Goby?

The Goby Underwater Autonomy Project is an autonomy architecture tailored for marine robotics with a focus on intervehicle communication.

Currently, Goby has two major parts:

- Goby-Acomms: The Goby Acoustic Communications libraries (`goby-acomms`) are provided for Version 1.1. See the Developers' documentation for details on these libraries at [1]. Users of the MOOS application `pAcommsHandler` should see Chapter 2.
- Goby-Core: An autonomy architecture that ties together various marshalling schemes (Google Protocol Buffers, MOOS, LCM, etc.) and provides a message passing middleware based on ZeroMQ (for ethernet) and Goby-DCCL (for acoustic communications). Goby-Core will be provided in release version 2.0.

1.2 Structure of this Manual

This manual covers the MOOS Applications that use Goby-Acomms release version 1.1. If you are interested in the C++ Goby-Acomms libraries directly, please read the online Developers' documentation at [1] In fact, you may want to go download and install Goby now before reading further: `https://launchpad.net/goby`.

1.3 How to get help

The Goby community is here to support you. This is an open source project so we have limited time and resources, but you will find that many are willing to contribute their help, with the hope that you will do the same as you gain experience. Please consult these resources and people, probably in this order of preference:

1. This user manual.
2. Questions and Answers on Launchpad: `https://answers.launchpad.net/goby`.
3. The developers' documentation: `http://gobysoft.com/doc`.
4. Email the listserver goby@mit.edu. Please sign up first: `http://mailman.mit.edu/mailman/listinfo/goby`.

5. Email the lead developer (T. Schneider): tes@mit.edu.

2 Goby MOOS Modules

The acoustic communications portion of Goby was developed originally for the MOOS autonomy architecture. Thus, the relevant MOOS modules `pAcommsHandler` and others are still maintained (in goby/src/moos) for the use of the `MOOS-IvP` community. `MOOS-IvP` is explained in [2] and is available at `http://moos-ivp.org`. The usage of these modules is documented here. See `http://gobysoft.org/wiki/InstallingGoby` for how to install Goby.

The beginning of this chapter motivates the design, followed by a detailed user manual for the individual MOOS processes.

2.1 Unified Command and Control for Subsea Autonomous Sensing Networks

The process of undersea observation, mapping, and monitoring is experiencing a dramatic paradigm shift away from platform-centric, human-controlled sensing, processing and interpretation. Rather, distributed sensing using networks of autonomous platforms is becoming the preferred technique. An optimal platform suite is often highly heterogeneous with large differences in mobility, maneuverability, sensing capability, and communication connectivity. The sensor systems have different constraints on platform mobility and communication capacity, and some network operations require highly coordinated maneuvering of heterogeneous platforms. Unified Command and Control [3] is a new command and control paradigm inherently suited for such heterogeneous networks. Implemented using `MOOS-IvP`, Unified C2 provides the fully integrated sensing, modeling and control that allows each platform, on its own or in collaboration with partners of opportunity, to autonomously detect, classify, localize and track (DCLT) an episodic, natural or human-created event, and subsequently report back to the operators.

A robust undersea communication infrastructure is crucial to the operation of such networks. In contrast to air and land-based equivalents, the extremely limited bandwidth, latency and intermittency of underwater acoustic communication imposes severe requirements to the selectivity of message handling. Thus, contact and track reports for high-priority event, such as a detected chemical plume from a deep ocean vent, which may indicate an imminent volcanic eruption, must be transmitted to the system operators without delay. On the other hand, reports concerning less important events and platform status reports may be delayed without significant effects. Previous message handling systems for underwater communications have only a rigid, hard-coded queuing infrastructure, and do not support such advanced priority-based selectivity, hampering the type and amount of infor-

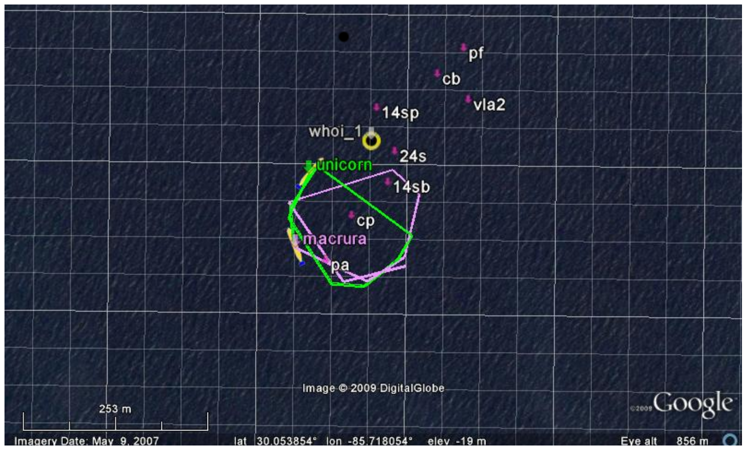

pf
cb
vla2
14sp
whoi_1
unicorn
24s
14sb
cp
macrura
pa
Image © 2009 DigitalGlobe
253 m
Imagery Date: May 9, 2007
lat 30.053854° lon -85.718054° elev -19 m
Eye alt 856 m
Google

Figure 2.1: Collaborative autonomy demonstrated in SWAMSI09 using MIT LAMSS communication stack. The two BF21 AUVs Unicorn and Macrura perform synchronized swimming maintaining a constant bistatic angle of 60° relative to a proud cylindrical target (cp).

mation that can be passed between cooperating nodes in the network. This severely limits the level of autonomy that can be supported on the network nodes.

In response to this problem, a new `MOOS-IvP` communication software stack was developed at the MIT Laboratory for Autonomous Marine Sensing Systems (LAMSS) [4], in support of autonomous sensing programs such as the ONR ASAP MURI, GOATS, and SWAMSI. This new stack has enabled the operation of a communication infrastructure which provides robust message handling for collaborative autonomous sensing by heterogeneous, undersea autonomous assets, as demonstrated in a handful of major recent field experiments. As an example, Fig. 2.1 shows the collaborative, multistatic MCM mission by the Unicorn and Macrura BF21 AUVs during SWAMSI09 in Panama City, FL. The two vehicles are circling a proud cylinder (cp) at a distance of 80 m maintaining a constant bistatic angle of 60 degrees. The collaboration was achieved fully autonomously without any intervention by the operators, with each vehicle adapting its speed based on its current position and the position of the other vehicle extrapolated from the latest status, contact or track report. Such collaborative maneuvers would not be possible using traditional communication schemes, where navigation packets must be rigidly interleaved with messages containg data and command and control sequences. In contrast, the Dynamic Compact Control Language (DCCL) used by the LAMSS communication stack allows for adequate navigation information to be packed with all other required message content.

Being based on the established open source goby-acomms libraries of message handling software, the open source architecture of this new MOOS communication stack (embodied primarily in the MOOS application pAcommsHandler lends itself directly to a wide range of military and civilian applications. It supports an arbitrary message suite and content without requirement of modifying software. All message encoding and decoding information is specified in a mission-unique configuration file written in the standard XML format. Not only does this ensure maximum flexibility in regard to message design, but it inherently enables arbitrary levels of encryption for LPI/LPD communication networks.

2.2 Overview of the LAMSS Communication Stack

MIT LAMSS [4] has over the last decade focused its research on the development of sensor-adaptive, collaborative, autonomous sensing concepts for the capture of episodic undersea events, including the mapping of coastal fronts, chemical plumes, and natural and man-made underwater acoustic sources. All these applications involve the Detection, Classification, Localization and Tracking (DCLT) of the event. To exploit the benefits of having multiple platforms involved in tracking the event, an underwater robust communication system is obviously a requirement. On the other hand, the communication capacity of such systems is many orders of magnitude below land- and air-based equivalents, requiring a much higher level of data compression and on-board processing and decision-making than is required in air-based systems. Unified C2 [3], developed over the last decade by LAMSS, is an example of such an autonomy-driven undersea sensing concept. Although this concept is based on the philosophy that the system must be able to achieve its mission objective even during periods with no or limited communication, there is obviously still a need for occasional communication, e.g. for reporting detected events of interest.

The new MOOS-IvP communication stack alleviates some of the problems and limitations of the existing software stacks in this regard. These software stacks in general were designed to sequentially transmit all messages generated by the autonomy system, with only a rigid, hard-coded priority-based message queuing infrastructure.

In undersea autonomous systems the priorities of information generated by the on-board processing are highly dynamic, depending on the tactical situation and the criticality of the generated information. Thus, for example, a contact report for a target of interest obviously must bypass queued contact reports for less significant targets. Also, in high-clutter environments, the number of contact reports may by far exceed the communication capacity and on-board priority-based filtering is required.

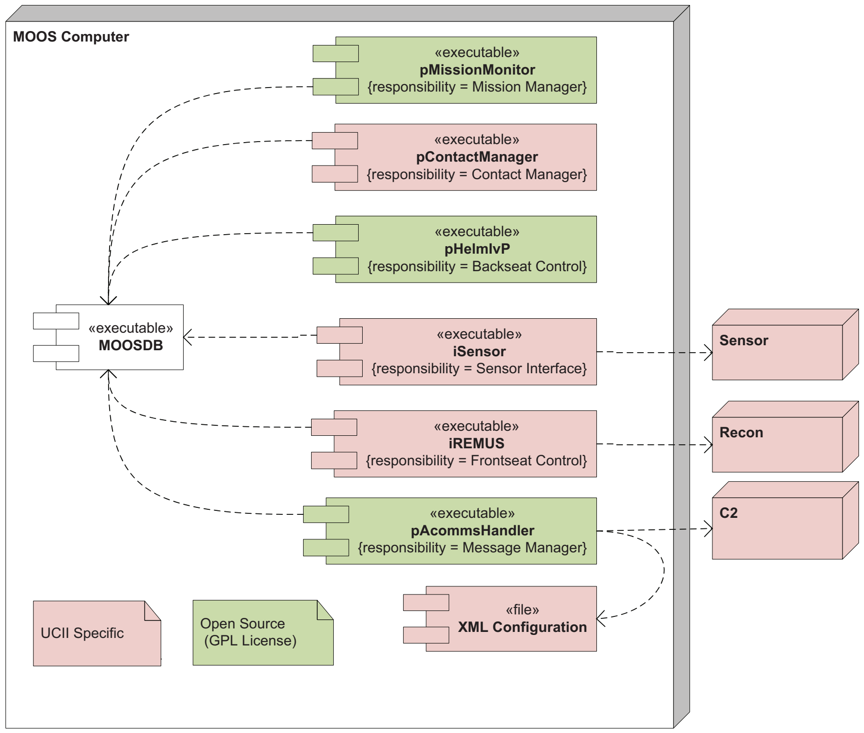


Figure 2.2: Incorporation of the open source LAMSS communication stack into a `MOOS-IvP` DCLT Autonomy System. The green boxes identify the open source modules, including the IvP Helm, the generic mission manager module, and the communication stack. The red modules are project specific, including the frontseat driver module, and the sensor modules. Also the message configuration specifying the message content and the coding, is project specific.

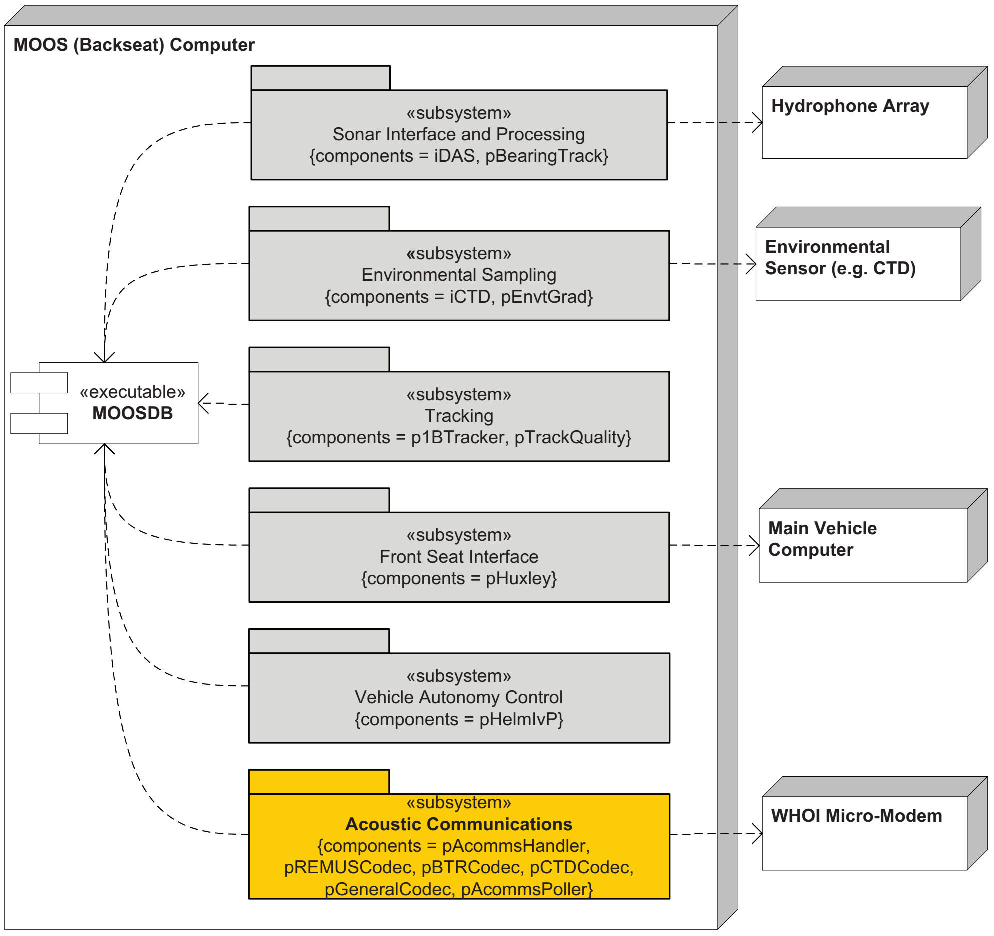


Figure 2.3: `MOOS-IvP` community for MIT sonar AUVs, with the autonomous communication, command and control modules highlighted in gold.

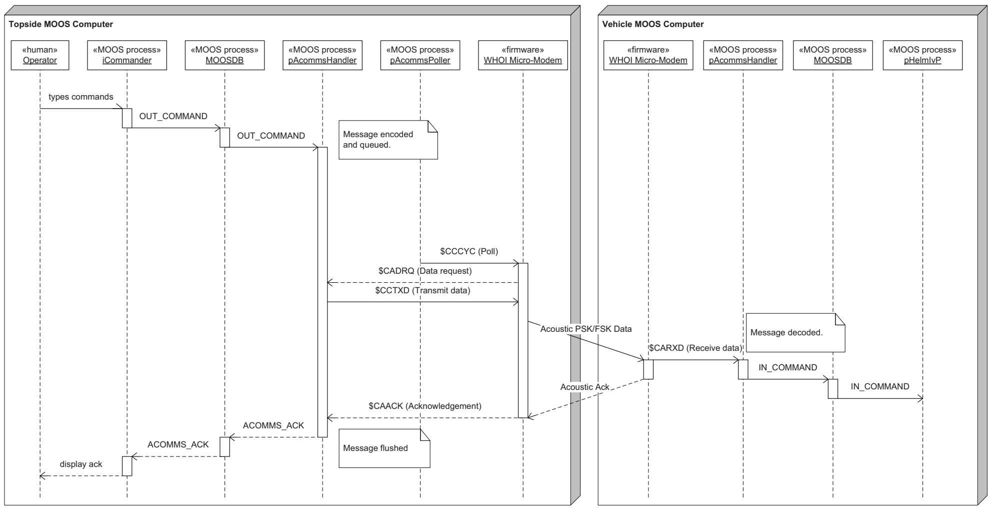


Figure 2.4: UML Sequence diagram for sending a command to an AUV via the LAMSS Acoustic Communications Modules.

The incorporation of the MIT LAMSS communication stack into a MOOS-IvP DCLT Autonomy System is illustrated in Fig. 2.2. The green boxes identify the Open Source modules, including the helm `pHelmIvP`, the generic mission manager module `pMissionMonitor`, and the communication stack. The red modules are project-specific, including the frontseat driver module `iRemus`, the sensor modules, and the contact manager process `pContactManager`. Also the message configuration files specifying the message content and the coding specifics, are project-specific and not hard-wired into the communication stack.

Figure 2.3 shows the communications subsystem as part of the whole MIT LAMSS AUV MOOS community.

Figure 2.4 shows the sequence of commands for a single operator command message sent using iCommander.

The structure of the MIT LAMSS communication stack is illustrated in Fig. 2.5.

2.3 pAcommsHandler

2.3.1 Problem

Acoustic communications are highly limited in throughput. Thus, it is unreasonable to expect "total throughput" of all communications data. Furthermore, even if total throughput is achievable over time, certain messages have a lower tolerance for delay (e.g. vehicle status) than others (e.g. CTD sample data). Refer-

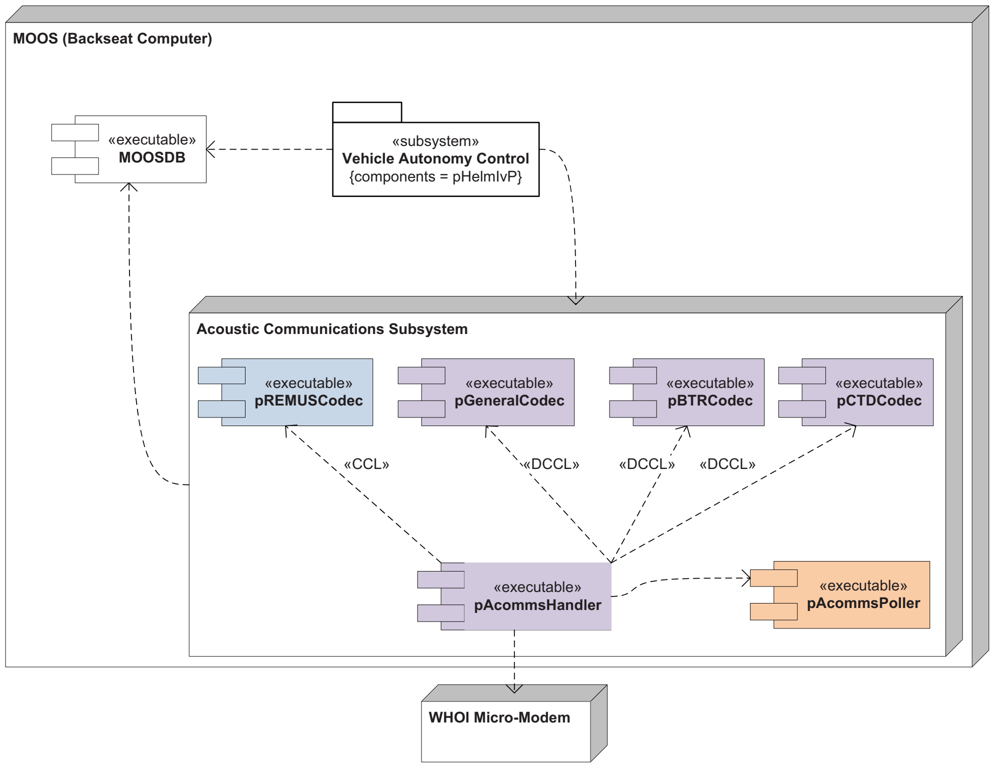

MOOS (Backseat Computer)
«executable» MOOSDB
«subsystem» Vehicle Autonomy Control {components = pHelmIvP}
Acoustic Communications Subsystem
«executable» pREMUSCodec
«executable» pGeneralCodec
«executable» pBTRCodec
«executable» pCTDCodec
«CCL»
«DCCL»
«DCCL»
«DCCL»
«executable» pAcommsHandler
«executable» pAcommsPoller
WHOI Micro-Modem

Figure 2.5: UML Component Model of the MIT LAMSS communication stack. The principal message handler module is `pAcommsHandler`, which communicates directly with the modem using built-in drivers, and thus not dependent on third-party MOOS modem drivers. It also manages the message stream by a dynamic, priority-based queuing system. The message coding and decoding is performed by `pGeneralCodec` based on the rules set out in the configuration file, and dedicated DCCL codecs for transmitting various data streams.The stack also supports standard fixed Compact Control Language (CCL) messages such as the State message used by the Remus AUV, using dedicated codecs. Dashed line indicate dependencies between components.

ence `http://acomms.whoi.edu/umodem/documentation.html` for more information on the WHOI Micro-Modem.

Also, in order to make the best use of this available bandwidth, messages need to be compacted to a minimal size before sending (effective encoding). To do this, pAcommsHandler provides an interface to the Dynamic Compact Control Language (DCCL[1].) encoder/decoder. Furthermore, DCCL has powerful parsing abilities ("algorithms") for both encoding and decoding, including the ability to perform certain geodesic conversions (e.g. latitude, longitude $\leftrightarrow$ UTM x,y) and lookups (e.g. *modem_id* $\leftrightarrow$ vehicle name) on data.

pAcommsHandler roughly performs the same functions of pFramer, pRouter, pAcommsPoller, and iMicroModem but generalized to handle any number of message queues and extended to give more control over queue parameters. The DCCL encoding is much more flexible and more compact than the CCL encoding used by these older processes.

2.3.2 Solution

pAcommsHandler provides a(n):

1. Encoder/decoder unit (codec): encodes and decodes messages using DCCL (goby-acomms `dccl` library), which reduces the data required to be sent by:
   - Predefined messages: the user must specify a message structure what specifies what fields the message contains and how large each field should be (in an intuitive fashion that DCCL turns into bits). Both the sender and receiver have preshared knowledge of the message structure. From this knowledge, no meta information about the message (beyond an identifier) needs to be sent, simply the data.
   - Custom field sizes: message fields are defined with custom tolerances (ranges and precisions) that are tighter than those given by the IEEE standards for floating point and integer numbers. For example, if a field needs to hold an integer that will never range outside [0, 1000] that field in the message will only be 10 bits long (ceil($\log_2 1001$)).
2. Priority Queuing System: maintains an arbitrary number of message queues (each tied to a different MOOS variable) for hexadecimal data strings. (goby-acomms `queue` library)

[1] the name comes from the original CCL written by Roger Stokey for the REMUS AUVs, but with the ability to dynamically reconfigure messages based on mission need. DCCL is backwards compatible with a CCL network as it uses CCL message number 32

- allows configuration of the queue priorities and dynamic growth of the priority over the time since the last sent message.
- allows management of WHOI CCL message types as well as DCCL queuing.

3. Modem Driver: handles all Micro-Modem serial communications. The driver (goby-acomms `modemdriver` library) can be used with other modems besides the WHOI Micro-Modem (see `http://gobysoft.com/doc/1.0/acomms__driver.html#acomms_writedriver` for information on writing a new driver).

4. MAC Manager: provides medium access control in the form of a simple slotted time division-multiple access (TDMA) scheme or flexible centralized polling (goby-acomms `amac` library).

2.3.3 Limitations

pAcommsHandler *does not*:

- presently provide any multi-hop routing. The sender and receiver must be directly connected acoustically.
- split user messages into packets. The user must provide data that are small enough to fit into the modem frame desired (32 - 256 bytes for the WHOI Micro-Modem).

2.3.4 Compilation

pAcommsHandler depends on the Goby and MOOS libraries. See goby/DEPENDENCIES for help resolving the dependencies on your system.

2.3.5 Parameters for the pAcommsHandler Configuration Block

Example moos file

You can always get a complete listing of MOOS file parameters with their syntax by running

```
> pAcommsHandler --example_config
```

This is a complete list of all the configuration values pAcommsHandler accepts. Most of the time you will need far fewer configuration options to use it.

```
ProcessConfig = pAcommsHandler
{
  common {  # Configuration common to all Goby MOOS applications
            # (opt)
    log: true  # Should we write a text log of the terminal
               # output? (opt) (default=true) (can also set MOOS
               # global "log=")
    log_path: "./"  # Directory path to write the text log of the
                    # terminal output (if log=true) (opt)
                    # (default="./") (can also set MOOS global
                    # "log_path=")
    community: "AUV23"  # The vehicle's name (opt) (can also set
                        # MOOS global "Community=")
    lat_origin: 42.5  # Latitude in decimal degrees of the local
                      # cartesian datum (opt) (can also set MOOS
                      # global "LatOrigin=")
    lon_origin: 10.9  # Longitude in decimal degrees of the local
                      # cartesian datum (opt) (can also set MOOS
                      # global "LongOrigin=")
    app_tick: 10  # Frequency at which to run Iterate(). (opt)
                  # (default=10)
    comm_tick: 10  # Frequency at which to call into the MOOSDB
                   # for mail. (opt) (default=10)
    verbosity: VERBOSITY_VERBOSE  # Verbosity of the terminal
                                  # window output (VERBOSITY_QUIET,
                                  # VERBOSITY_WARN,
                                  # VERBOSITY_VERBOSE,
                                  # VERBOSITY_DEBUG, VERBOSITY_GUI)
                                  # (opt)
                                  # (default=VERBOSITY_VERBOSE)
    initializer {  # Publish a constant value to the MOOSDB at
                   # startup (repeat)
      type: INI_DOUBLE  # type of MOOS variable to publish
                        # (INI_DOUBLE, INI_STRING) (req)
      moos_var: "SOME_MOOS_VAR"  # name of MOOS variable to
                                 # publish to (req)
      global_cfg_var: "LatOrigin"  # Optionally, instead of
                                   # giving `sval` or `dval`, give
                                   # a name here of a global MOOS
                                   # variable (one at the top of
                                   # the file) whose contents
                                   # should be written to
                                   # `moos_var` (opt)
      dval: 3.454  # Value to write for type==INI_DOUBLE (opt)
      sval: "a string"  # Value to write for type==INI_STRING
                        # (opt)
    }
  }
  modem_id: 1  # Unique number 1-31 to identify this node (req)
  driver_type: DRIVER_NONE  # Corresponding driver for the type
                            # of physical acoustic modem used
                            # (DRIVER_NONE, DRIVER_WHOI_MICROMODEM,
                            # DRIVER_ABC_EXAMPLE_MODEM) (opt)
                            # (default=DRIVER_NONE)
  driver_cfg {  # Configure the acoustic modem driver (opt)
    modem_id: 1  # Unique number 1-31 to identify this node (req)
    connection_type: CONNECTION_SERIAL  # Physical connection
                                        # type from this computer
                                        # (running Goby) to the
                                        # acoustic modem
                                        # (CONNECTION_SERIAL,
                                        # CONNECTION_TCP_AS_CLIENT,
                                        # CONNECTION_TCP_AS_SERVER,
                                        # CONNECTION_DUAL_UDP_BROADC
                                        # AST) (opt)
                                        # (default=CONNECTION_SERIAL
                                        # )
    line_delimiter: "\r\n"  # String used to delimit new lines
                            # for this acoustic modem (opt)
                            # (default="\r\n")
    serial_port: "/dev/ttyS0"  # Serial port for
                               # CONNECTION_SERIAL (opt)
    serial_baud: 19200  # Baud rate for CONNECTION_SERIAL (opt)
    tcp_server: "192.168.1.111"  # IP Address or domain name for
                                 # the server if
                                 # CONNECTION_TCP_AS_CLIENT (opt)
    tcp_port: 50010  # Port to serve on (for
                     # CONNECTION_TCP_AS_SERVER) or to connect to
                     # (for CONNECTION_TCP_AS_CLIENT) (opt)
  }
  mac_cfg {  # Configure the acoustic Medium Access Control (opt)
    modem_id: 1  # Unique number 1-31 to identify this node (req)
    type: MAC_NONE  # The type of TDMA MAC scheme to use
                    # (MAC_NONE, MAC_FIXED_DECENTRALIZED,
                    # MAC_AUTO_DECENTRALIZED, MAC_POLLED) (opt)
                    # (default=MAC_NONE)
    slot {  # Configure a slot in the communications cycle. Slots
            # are run in the order they are declared. Omit for
            # MAC_AUTO_DECENTRALIZED. (repeat)
      src: 1  # source modem id for this transmission (initiating
              # platform) (req)
      dest: -1  # destination modem id for this transmission; 0
```

```
                 # means broadcast, -1 means query the queuing layer
                 # for next available message (opt) (default=-1)
      rate: 0  # bit rate (integer from 0-5, 0 is slowest) (opt)
               # (default=0)
      type: SLOT_DATA  # type of message to initiate in this slot
                       # (SLOT_DATA, SLOT_PING, SLOT_REMUS_LBL)
                       # (req) (default=SLOT_DATA)
      slot_seconds: 15  # length of this slot in seconds (opt)
      last_heard_time: ""  # used internally, no need to
                           # configure manually (opt)
    }
    rate: 0  # Set rate to use for MAC_AUTO_DECENTALIZED. Use
             # `slot` for other MACTypes (opt) (default=0)
    slot_seconds: 15  # Set duration of the slot for
                      # MAC_AUTO_DECENTRALIZED. Use `slot` for
                      # other MACTypes (opt) (default=15)
    expire_cycles: 30  # Set number of quiet cycles for
                       # discarding a node from the cycle for
                       # MAC_AUTO_DECENTRALIZED. (opt) (default=30)
  }
  queue_cfg {  # Configure the Priority Queuing layer (opt)
    modem_id: 1  # Unique number 1-31 to identify this node (req)
    message_file {  # XML message file containing one or more
                    # DCCL message descriptions. Use for specifying
                    # DCCL queues. (repeat)
      path: "/home/toby/goby/src/acomms/examples/chat/chat.xml"
                                              # path to the
                                              # message XML file
                                              # (req)
      manipulator: NO_MANIP  # manipulators to modify the
                             # encoding and queuing behavior of the
                             # messages in this file (NO_MANIP,
                             # NO_ENCODE, NO_DECODE, NO_QUEUE,
                             # LOOPBACK, ON_DEMAND, TCP_SHARE_IN)
                             # (repeat)
    }
    queue {  # Use for specifying CCL queues; use message_file
             # for DCCL queues. (repeat)
      ack: true  # Require acoustic acknowledgments of messages
                 # sent from this queue (opt) (default=true)
      blackout_time: 0  # Time in seconds to ignore this queue
                        # after the last send from it. (opt)
                        # (default=0)
      max_queue: 0  # Maximum allowed messages in this queue (0
                    # means infinity). (opt) (default=0)
      newest_first: true  # true = FILO queue, false = FIFO queue
```

```
                            # (opt) (default=true)
        value_base: 1  # Base value (general importance) of the
                       # messages in this queue (opt) (default=1)
        ttl: 1800  # Time to live in seconds; messages exceeding
                   # this time are discarded. Also factors into
                   # priority equation (opt) (default=1800)
        key {  #  (opt)
          type: QUEUE_DCCL  # Type of messages in this queue
                            # (QUEUE_DCCL, QUEUE_CCL) (req)
                            # (default=QUEUE_DCCL)
          id: 14  # DCCL ID for QUEUE_DCCL, CCL Identifier (first)
                  # byte for QUEUE_CCL (req)
        }
        name: "Remus_State"  # Human readable name for this queue
                             # (req)
        in_pubsub_var: "REMUS_STATE_RAW_IN"  # Publish subscribe
                                             # architecture variable
                                             # for posting incoming
                                             # data to (opt)
        out_pubsub_var: "REMUS_STATE_RAW_OUT"  # Publish subscribe
                                               # architecture
                                               # variable for
                                               # fetching outgoing
                                               # data from (opt)
      }
    }
    dccl_cfg {  # Configure the Dynamic Compact Control Language
                # Encoding/Decoding unit (opt)
      modem_id: 1  # Unique number 1-31 to identify this node (req)
      message_file {  # XML message file containing one or more
                      # DCCL message descriptions (repeat)
        path: "/home/toby/goby/src/acomms/examples/chat/chat.xml"
                                                # path to the
                                                # message XML file
                                                # (req)
        manipulator: NO_MANIP  # manipulators to modify the
                               # encoding and queuing behavior of the
                               # messages in this file (NO_MANIP,
                               # NO_ENCODE, NO_DECODE, NO_QUEUE,
                               # LOOPBACK, ON_DEMAND, TCP_SHARE_IN)
                               # (repeat)
      }
      crypto_passphrase: "twinkletoes%24"  # If given, encrypt all
                                           # communications with this
                                           # passphrase using AES.
                                           # Omit for unencrypted
```

```
                                         # communications. (opt)
    }
    modem_id_lookup_path: ""  # Path to file containing mapping
                              # between modem_id and vehicle name &
                              # type (opt) (can also set MOOS global
                              # "modem_id_lookup_path=")
    tcp_share_enable: false  # Enable TCP Sharing (Experimental)
                             # (opt) (default=false)
    tcp_share_port: 11000  # Port to listen on for TCP Sharing
                           # (Experimental) (opt) (default=11000)
    tcp_share_to_ip: ""  # internet_address:port to share incoming
                         # messages to (Experimental). (repeat)
  }
```

Filling out the .moos file

Many of the parameters are sufficiently explained in the above list of configuration parameters. What follows is a detailed explanation of the parameters that need further explanation.

- `common`: Parameters that can be set for any of the Goby MOOS applications. Here you can control logging to a text file, terminal verbosity. You can also initialize a variable in the MOOS database at startup. Many of these parameters will automatically be set to a global MOOS variable (specified outside any ProcessConfig block) if left empty. For example, the global MOOS variable `LatOrigin` will set the pAcommsHandler variable `common::lat_origin`. This allows pAcommsHandler to conform to MOOS *de facto* conventions.
  - `verbosity`: choose `VERBOSITY_VERBOSE` for full text terminal output, `VERBOSITY_WARN` for warnings only, and `VERBOSITY_QUIET` for no terminal output. `VERBOSITY_GUI` opens an NCurses GUI helpful to debugging and visualizing the many data flows of pAcommsHandler.
  - `initializer`: since many times it is useful to have a MOOS variable including in a message that remains static for a given mission (vehicle name, etc), we give the option to publish initial MOOS variables here (for later use in messages [until overwritten, of course]). If `global_cfg_var` is set, pAcommsHandler looks for a global (i.e. specified at the top of the MOOS file or outside any `ProcessConfig` blocks) value in the .moos file with the name to the right of the colon and publishes it to a MOOS variable with the name to the left of the colon. For example:

    ```
    initializer { global_cfg_var: "LatOrigin" moos_var: "LAT_ORIGIN" }
    ```

looks for a variable in the .moos file called `LatOrigin` and publishes it to the MOOSDB as a double variable `LAT_ORIGIN` with the value given by `LatOrigin`.

  - `log_path`: folder to log all terminal output to for later debugging. Similar to system logs in /var/log.
  - `log`: boolean to indicate whether to log terminal output or not to files in the path by `log_path`.

- `modem_id`: integer that specifies the `modem_id` of this current vehicle / community. For the WHOI Micro-Modem this is the Micro-Modem "SRC" configuration parameter (as set by `\$CCCFG,SRC,#` to check). For the remainder of the document, `modem_id` refers to the value `\$CCCFG,SRC,modem_id`. This configuration parameter will be set on startup. Setting this within the main block for pAcommsHandler sets it for all the modules (`driver_cfg`, `dccl_cfg`, `queue_cfg`, `mac_cfg`)

- `modem_id_lookup_path`: path to a text file giving the mapping between `modem_id` and vehicle name and type for a given experiment. This file should look like:

```
// modem id, vehicle name (should be community name), vehicle type
0, broadcast, broadcast
1, endeavor, ship
3, unicorn, auv
4, macrura, auv
```

Encoding/Decoding (DCCL) Parameters (`dccl_cfg`)

- `modem_id`: Will be set to the same as `ProcessConfig { modem_id: }` . There is no need to set it again here.

- `message_file`: `path` to an XML file containing a message set of one or messages. If you want, you can insert one or more manipulators that change the behavior of pAcommsHandler for messages defined in that file. Allowed manipulators:

  - `NO_MANIP`: blank manipulator (behavior is not modified by this manipulator)
  - `NO_ENCODE`: do not encode this message
  - `NO_DECODE`: do not decode this message

  - `NO_QUEUE`: do not queue this message
  - `LOOPBACK`: decode this message internally immediately following encode. Note that messages addressed to the local vehicle are looped back regardless of the value of this manipulator.
  - `ON_DEMAND`: encode immediately preceding a data request command (use for time sensitive messages like STATUS). This only works if all the message variables are always assumed fresh in the MOOSDB.

- `crypto_password`: optionally provide a password here to encrypt all communications using AES. All receiving nodes must have the same password.

Queuing Parameters (`queue_cfg`) All queue configuration for DCCL messges must be configured within the XML files `<queuing />` tag and included with `message_file: {path: "message.xml"}`. Any `message_files` specified for `dccl_cfg` are copied to `queue_cfg` and vice-versa, so you don't need to specify them in two places.

CCL messages are configured using the `queue { }` object. The fields for `queue` correspond to the XML `<queuing />` tags:

- `id`: DCCL: a unique ID for this message (in the range 0-511). CCL: The decimal representation of the first byte of the CCL message to be queued.

- `ack`: 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.

- `blackout_time`: 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.

- `max_queue`: 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.

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

- `ttl`: the time (in seconds) the message is allowed to live before being discarded. This also factors into the priority calculation as messages with a lower time-to-live (ttl) grow in priority faster.

- `value_base`: 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.

- `in_pubsub_var`: name of the moos variable that is published for received messages to this queue. Not used for DCCL queuing.

- `out_pubsub_var`: name of the moos variable to subscribe to for messages to add to this queue. Not used for DCCL queuing.

An example queuing block (for DCCL messages):

```
<message_set>
  <message>
    <id>23</id>
    ...
    <queuing>
      <ack>false</ack>
      <blackout_time>0</blackout_time>
      <max_queue>1</max_queue>
      <newest_first>true</newest_first>
      <value_base>4</value_base>
      <ttl>1000</ttl>
    </queuing>
  </message>
  ...
</message_set>
```

Modem Driver Parameters (`driver_cfg`)

- `driver_type`: The only real driver implemented is the `DRIVER_WHOI_MICROMODEM`. `DRIVER_ABC_EXAMPLE_MODEM` is a simple test "modem". `DRIVER_NONE` disables the modem driver.
- `connection_type`: type of connection to make to the modem (`CONNECTION_SERIAL`, `CONNECTION_TCP_AS_CLIENT`, `CONNECTION_TCP_AS_SERVER`).
- `serial_port`: serial port to which the modem is connected.
- `serial_baud`: baud rate to use. Should be set to 19200 for the WHOI Micro-Modem.
- `tcp_port`: networking port to use.
- `tcp_server`: IPv4 networking address of the server to connect to.

Extensions for the WHOI Micro-Modem

- `[MicroModemConfig.nvram_cfg]`: set some modem NVRAM setting to a value. Set `[MicroModemConfig.reset_nvram]: true` to reset all NVRAM (CFG) parameters on startup (`\$CCCFG,ALL,0`). All the `[MicroModemConfig.nvram_cfg]` values are sent after this reset. You do not need to send SRC as this is set to the `modem_id`.
- `[MicroModemConfig.hydroid_gateway_id]`: Set to the HYDROID gateway id (1 or 2) *only if using a HYDROID gateway buoy*. Omit for a normal WHOI Micro-Modem.

Medium Access Control (MAC) Parameters (mac_cfg)

- `type`: type of Medium Access Control. See http://gobysoft.com/doc/1.0/acomms__mac.html#amac_schemes for an explanation of the various MAC schemes.
- `slot_seconds`: length, in seconds, of each communication slot for the `type: MAC_AUTO_DECENTRALIZED` MAC option.
- `rate`: rate for the `type: MAC_AUTO_DECENTRALIZED` MAC option. For the WHOI Micro-Modem 0 is a single 32 byte packet (FSK), 2 is three frames of 64 bytes (PSK), 3 is two frames of 256 bytes (PSK), and 5 is eight frames of 256 bytes (PSK)
- `expire_cycles`: number of consecutive cycles in which a vehicle can be silent before being removed from the cycle for the `type: MAC_AUTO_DECENTRALIZED` MAC option.

- `slot`: use this repeated field to specify a manual polling or fixed TDMA cycle for the `type: MAC_FIXED_DECENTRALIZED` and `type: MAC_POLLED`.
  - `src`: The sending `modem_id` for this slot.
  - `dest`: The receiving `modem_id` for this slot.
  - `rate`: Bit-rate code for this slot (0-5).
  - `type`: Type of transaction to occur in this slot. Can be `SLOT_DATA` (send a datagram), `SLOT_PING` (send a ranging two-way ping to another modem), `SLOT_REMUS_LBL` (ping a REMUS LBL network (WHOI Micro-Modem only)).
  - `slot_seconds`: The duration of this slot, in seconds.

2.3.6 MOOS variables subscribed to by pAcommsHandler

Except for DCCL <`src_var`>s and <`trigger_var`>s, pAcommsHandler uses the Google Protocol Buffers TextFormat class for parsing from MOOS strings. This saves significant effort in manually parsing strings. You should use these same facilities for creating and reading messages. Two helper functions are provided in goby/moos/libmoos_util/moos_protobuf_helpers will help you serialize and parse these messages. See `http://gobysoft.com/doc/1.0/acomms.html#protobuf` for a brief overview of Google Protocol Buffers as used in Goby.

- `DCCL`: Most variables subscribed to by pAcommsHandler are configured in the message XML files and are designated by the tags <`src_var`> (used to fetch data for a particular `message_var` within a DCCL message) and <`trigger_var`> (used to trigger the creatinon of a particular DCCL message and possibly provide some data for that message. See 2.3.8 for details on the XML configuration.
- `Queue`:
  - Subscribes to the variables given in `queue_cfg.queue.in_pubsub_var` for CCL queue sending. The contents of this MOOS variable should be a serialized ModemDataTransmission).
  - `ACOMMS_RANGE_COMMAND` (type: ModemRangingRequest): You write this to initiate a ranging request outside the MAC schedule. Note in general it is preferable to use the MAC cycle to coordinate data and ranging.
- `MAC`: `ACOMMS_MAC_CYCLE_UPDATE` (type: MACUpdate) You write this to update the MAC cycle for `MAC_FIXED_DECENTRALIZED` and `MAC_POLLED` modes of operation.

For example, to publish a `ACOMMS_MAC_CYCLE_UPDATE`, you would use code like this:

```
// provides serialize_for_moos
#include <goby/moos/libmoos_util/moos_protobuf_helpers.h>
// provides goby::acomms::protobuf::MACUpdate
#include <goby/protobuf/amac.pb.h>

...

MyMOOSApp::Iterate()
{
  if(do_update_mac)
  {
    using namespace goby::acomms::protobuf;
    MACUpdate mac_update;
    mac_update.set_dest(1); // update for us if modem_id == 1
    // add slot to end of existing cycle
    mac_update.set_update_type(MACUpdate::ADD);
    Slot* new_slot = mac_update.add_slot();
    new_slot->set_src(1);  // send from us
    new_slot->set_dest(3); // send to vehicle 3
    new_slot->set_rate(0);
    new_slot->set_slot_seconds(15);
    new_slot->set_type(SLOT_DATA);

    std::string serialized;
    serialize_for_moos (&serialized, mac_update);
    m_Comms.Notify("ACOMMS_MAC_CYCLE_UPDATE", serialized);
  }
}
```

2.3.7 MOOS variables published by pAcommsHandler

Except for DCCL <`publish_var`>s (which use a printf style syntax), pAcommsHandler uses the Google Protocol Buffers TextFormat class for serializing to MOOS strings.

- `DCCL`: Most variables published by pAcommsHandler are configured in the message XML files and are designated by the tags <`publish_var`> within a <`publish`> block. See 2.3.8 for details on the XML configuration.
- `Queue`:
  - `ACOMMS_INCOMING_DATA` (type: ModemDataTransmission) written for all received messages containing a data payload

  - `ACOMMS_OUTGOING_DATA` (type: ModemDataTransmission) written for all queued messages containing a data payload
  - `ACOMMS_RANGE_RESPONSE` (type: ModemRangingReply) written in response to ranging request (to another modem or LBL beacons)
  - `ACOMMS_ACK` (type: ModemDataAck) written when received data is acknowledged acoustically by a third party. Contains the original message.
  - `ACOMMS_EXPIRE` (type: ModemDataExpire) written when a message expires (time-to-live [ttl] exceeded) from the queue before being sent (ack = false) or acknowledged (ack = true)
  - `ACOMMS_QSIZE` (type: QueueSize) written when a queue changes size (pop or push) with the new size of the queue.

- `MAC`: Does not publish anything.
- `ModemDriver`:
  - `ACOMMS_NMEA_IN` (type: string), ModemMsgBase::raw() for all incoming messages ("$CA..." for WHOI Micro-Modem)
  - `ACOMMS_NMEA_OUT` (type: string), ModemMsgBase::raw() for all outgoing messages ("$CC..." for WHOI Micro-Modem)

For example, to read an `ACOMMS_RANGE_RESPONSE`, you would use code like this:

```
// provides parse_for_moos
#include <goby/moos/libmoos_util/moos_protobuf_helpers.h>
// provides goby::acomms::protobuf::ModemRangeReply
#include <goby/protobuf/modem_message.pb.h>

...

MyMOOSApp::OnNewMail()
{
  ...
  if(moos_msg.GetKey() == "ACOMMS_RANGE_RESPONSE")
  {
    using namespace goby::acomms::protobuf;
    ModemRangeReply range_response;
    parse_for_moos (serialized, &range_response);

    // now do what you want to with the nice `range_response` object
    std::cout << "one way travel time to " << range_response.base().dest()
              << " is " << range_response.one_way_travel_time(0) << std::endl;
  }
}
```

2.3.8 DCCL Encoding/Decoding Unit: Overview

Example message XML file

First, let us give a brief background on XML (eXtensible Markup Language). XML files contain tags (like <name>) that are considered "metadata" and define both the structure of the following data and the contents. Order of the tags does not matter for a given level unless otherwise specified. Text data resides both in the tags (like <name>bob</name> or as attributes of the tag (such as <name id="1245"></name>). XML files can be edited with any text editor. For more information on XML consult any number of books on the subject or browse the internet. XML is a very widely used format for storing data that can be both read by both people and computers. Also see section 2.3.9 for further examples. Let's call this file example1.xml, which we will use in two following examples:

```
<?xml version="1.0" encoding="ASCII" standalone="yes"?>
<message_set>
  <message>
    <name>GoToCommand</name>
    <id>1</id>
    <trigger>publish</trigger>
    <trigger_var mandatory_content="CommandType=GoTo">
      OUTGOING_COMMAND
    </trigger_var>
    <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>
        <moos_var>SPECIAL_INSTRUCTIONS</moos_var>
        <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>
    <on_receipt>
      <publish>
        <publish_var>INCOMING_COMMAND</publish_var>
        <all />
      </publish>
      <publish>
        <publish_var>SPECIAL_INSTRUCTIONS</publish_var>
        <format>special_instructions=%1%,lights_on=%2%</format>
        <message_var>new_instructions</message_var>
        <message_var>lights_on</message_var>
      </publish>
    </on_receipt>
  </message>
  <message>
    <name>VehicleStatus</name>
    <id>2</id>
    <trigger>time</trigger>
    <trigger_time>30</trigger_time>
    <size>32</size>
    <layout>
      <float>
        <name>nav_x</name>
        <src_var>NAV_X</src_var>
        <max>1000</max>
        <min>0</min>
        <precision>1</precision>
      </float>
      <float>
        <name>nav_y</name>
        <src_var>NAV_Y</src_var>
        <max>1000</max>
        <min>0</min>
        <precision>1</precision>
      </float>
      <enum>
        <name>health</name>
        <src_var>VEHICLE_HEALTH</src_var>
        <value>good</value>
        <value>low_battery</value>
        <value>abort</value>
      </enum>
    </layout>
    <on_receipt>
      <publish>
        <publish_var>STATUS_SUMMARY</publish_var>
        <all />
      </publish>
    </on_receipt>
  </message>
</message_set>
```

2.3.9 DCCL Encoding/Decoding Unit: Designing Messages

Designing a publish triggered message

We will look at two scenarios and detail how to design a proper message file for each scenario. We will reference the example file given in section 2.3.8 for both scenarios.

Scenario: you want to command an surface craft to move to a new location:

1. Identify the data: location (x (goto_x) and y (goto_y) on a local grid). you also want to specify a speed (goto_speed) at which it should transit, whether it should have lights (lights_on) on or not, and finally a string (special_instructions) with possible special instructions. All these data will come in to a moos variable OUTGOING_COMMAND on a string like:

```
OUTGOING_COMMAND: Destination=3,CommandType=GoTo,goto_x=351,goto_y=294,
            lights_on=true,special_instructions=make_toast,goto_speed=2.3
```

2. Type the data (i.e. is it an int, a float, a string?) and give the ranges and precisions needed:

   - `goto_x`: integer (in meters) (`int`) that will operate on a (positive valued) local grid not to exceed 10 km in either dimension.
   - `goto_y`: same as `goto_x`.
   - `goto_speed`: speed in m/s. the vehicle cannot exceed 3 m/s and does not go backwards. we would like to give precise speeds to the hundredths place. thus, we need a `float` ranging from 0 to 3 with precision 2.
   - `lights_on`: simply a flag (boolean value) whether to have our lights on or off. thus, we need a `bool` *message_var*.
   - `special_instructions`: We want a field that can hold any string of characters, but we know it will not exceed ten characters. thus, we need a `string` *message_var*.

3. Putting all this together, we can define the <`layout`> portion of the first message defined in section 2.3.8. We do not need any <`src_var`> tags within the *message_var*s since all the data are contained in the contents of the trigger variable message (`OUTGOING_COMMAND`). That is, when we leave out the <`src_var`>, pAcommsHandler will insert <`src_var`>`OUTGOING_COMMAND`</`src_var`>, which is exactly what we want. For example, taking one of the *message_var*s:

```
        <int>
          <name>goto_x</name>
          <max>10000</max>
          <min>0</min>
        </int>
```

   is exactly the same as saying

```
        <int>
          <name>goto_x</name>
          <src_var>OUTGOING_COMMAND</src_var>
          <max>10000</max>
          <min>0</min>
        </int>
```

4. Now we can fill out the rest of the tags on the <`message`> level:

- `<name>GoToCommand</name>`: just a name so we can identify this message quickly when reading through the XML.
- `<trigger>publish</trigger>`: we are creating this message on a publish (to `OUTGOING_COMMAND`).
- `<trigger_var mandatory_content="CommandType=GoTo"> OUTGOING_COMMAND </trigger_var>`: `OUTGOING_COMMAND` is the trigger variable and it must contain the substring `CommandType=GoTo`. That is, other commands might be published here (e.g. `CommandType=Loiter`, `CommandType=Track`) and we do not define the message structure of those here (this particular `<message>` is only for a GoTo message). Other messages can be created to encode/decode these other command types.
- `<size>32</size>`: we want this message to fit in a WHOI micromodem FSK frame (32 bytes).

5. Finally, we fill out the `<publish>` section which indicates where (i.e. what moos variables) and how (what format and which part(s) of the message) pAcommsHandler should publish decoded messages upon receipt of hex from other vehicles. Each `<publish>` indicates a separate action that is taken upon receipt of a message. As many `<publish>` sections as desired may be included for a given message. So, for our example message, we want to replicate the original string (a common practice):

```
INCOMING_COMMAND: CommandType=GoTo,goto_x=351,goto_y=294,
            lights_on=true,special_instructions=make_toast,goto_speed=2.3
```

to do this we fill out a publish `<all>`. This is the simplest form of the `<publish>` section:

```
      <on_receipt>
        <publish>
          <publish_var>INCOMING_COMMAND</publish_var>
          <all />
        </publish>
      </on_receipt>
```

this says to take every *message_var* and make a "key=value" comma-delimited string from it. the above `<publish>` block is a shortcut for a much longer form:

```
    <on_receipt>
      <publish>
        <publish_var>INCOMING_COMMAND</publish_var>
        <format>type=goto,goto_x=%1%,goto_y=%2%,lights_on=%3%,
        special_instructions=%4%,goto_speed=%5%</format>
        <message_var>goto_x</message_var>
        <message_var>goto_y</message_var>
        <message_var>lights_on</message_var>
        <message_var>special_instructions</message_var>
        <message_var>goto_speed</message_var>
      </publish>
    </on_receipt>
```

These two blocks are functionally identical.

We may want to also publish the `special_instructions` to another moos variable, so that:

```
SPECIAL_INSTRUCTIONS: special_instructions=make_toast,lights_on=true
```

we can do this with another publish block:

```
    <publish>
      <publish_var>SPECIAL_INSTRUCTIONS</publish_var>
      <format>special_instructions=%1%,lights_on=%2%</format>
      <message_var>new_instructions</message_var>
      <message_var>lights_on</message_var>
    </publish>
```

in this case the <`format`> block is necessary because the default would be <`format`>`new_instructions=%1%,lights_on=%2%`</`format`> not <`format`>`special_instructions=%1%,lights_on=%2%`</`format`>.

Those are the basics to designing a publish triggering message.

Designing a time triggered message Scenario: we need a status message that grabs data from various moos variables and publishes them (encoded) on a time interval. We will not go into as much detail here, but rather highlight the changes from the previous scenario.

- you will notice

```
    <trigger>time</trigger>
    <trigger_time>30</trigger_time>
```

instead of

```
    <trigger>publish</trigger>
    <trigger_var mandatory_content="CommandType=GoTo">
      OUTGOING_COMMAND
    </trigger_var>
```

this indicates that a message should be made on a time interval (given by <`trigger_time`>, which is every 30 seconds here), rather than on a publish to some MOOS variable.

- you will notice that all the *message_var*s have a <`src_var`> tag, which was omitted in the previous example since we were taking data from the trigger variable. Obviously, there is no trigger variable now so we must specify a location for the data to come from (in the MOOSDB). The newest available value will be used when the message needs to be made. This means there is no guarantee that the data is fresh. Thus, you should use MOOS variables that are often updated for a <`trigger`>`time`</`trigger`> message. If this is not the case, a <`trigger`>`publish`</`trigger`> message (see previous scenario) may be a better choice.

- the format of the value read from the <`src_var`> can have several options. First, if the *message_var* is of a numeric type (<`int`>, <`float`>, <`bool`>) and the <`moos_var`> is a double, the value of the double is used as is (with appropriate rounding and type casting). If the *message_var* is a string, two options are available. First, pAcommsHandler looks for a substring of the form:

  ```
  name=value
  ```

  within the string and picks out `value` to send for the message. If there is no such `name=` substring, the entire string is converted to the appropriate form. An example: we have a <`float`> called <`name`>`my_float`</`name`> that has a tag <`moos_var`>`SOME_FLOAT_VARIABLE`</`moos_var`>:

  - if

```
1  (double)SOME_FLOAT_VARIABLE: 3.56
```

    then 3.56 is sent.
  - if instead

```
2  (string)SOME_FLOAT_VARIABLE: "my_float=3.56"
```

    then 3.56 is still sent.
  - if instead

```
3  (string)SOME_FLOAT_VARIABLE: "3.56"
```

    again, 3.56 is sent.
  - Finally, if some other string like

```
4  (string)SOME_FLOAT_VARIABLE: "blah=3.56"
```

    then `blah=3.56` is converted to a float, which will probably be zero or something else undesired. In other words, case `4` is not what you want, whereas `1-3` are fine.

Further examples

- I currently store some example working message files in `goby/xml`. look for .xml files in this directory for further examples.
- Probably the simplest message you can make (for a single string MOOS variable published to `IN_MESSAGE` that gets truncated at 26 chars (need six bytes for the DCCL header) and sent to broadcast):

```
<?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>

      <!-- only used by pAcommsHandler (publish/subscribe)-->
      <trigger>publish</trigger> <!-- pack -->
      <trigger_var>OUT_MESSAGE</trigger_var>
      <on_receipt> <!-- unpack -->
        <publish>
          <publish_var>IN_MESSAGE</publish_var>
          <message_var>message</message_var>
        </publish>
      </on_receipt>
      <!-- end used by pAcommsHandler -->

    </message>
  </message_set>
```

2.3.10 DCCL Encoding/Decoding Unit: XML Tag Reference

The XML tag reference is now part of the Goby Developers documentation (http://gobysoft.com/doc/1.0/:

- See http://gobysoft.com/doc/1.0/acomms__dccl.html#dccl_tags for a structure of all the allowed tags.
- Visit http://gobysoft.com/doc/1.0/acomms__dccl.html#dccl_tags_details for an up-to-date reference of all the DCCL tags with a description of their usage.

Algorithms

You can perform a number of simple algorithms on data either before encoding (specified in the message_var tag (e.g. <string algorithm="">) or after receipt (specified in the <message_var> tag. You can apply more than one algorithm by separating them with commas and they are processed in the order given. The currently implemented algorithms include:

- `to_upper`: converts string, enum, or bool to uppercase
- `to_lower`: converts string, enum, or bool to lowercase
- `angle_0_360`: wraps float or int angle in degrees into the range of [0, 360)
- `angle_-180_180`: wraps float or int angle in degrees into the range of [-180, 180)
- `lon2utm_x`: converts longitude to a local utm coordinate (meters) used by LAMSS[2]. Requires `LatOrigin` and `LongOrigin` to be specified at the top of the moos file. Since a UTM conversion requires a lon/lat pair, you must specify the latitude variable here to pair with by adding a colon after this algorithm followed by the name of the latitude variable. e.g.

  ```
  <message_var algorithm="lon2utm_x:our_lat">our_lon</message_var>
  ```

  converts `our_lon` to a local x (easting) using `our_lat` as the latitude point.
- `lat2utm_y`: similar to `lon2utm_x` but for latitude. e.g.

  ```
  <message_var algorithm="lat2utm_y:our_lon">our_lat</message_var>
  ```

  converts `our_lat` to a local y (northing) using `our_lon` as the longitude point.
- `utm_x2lon`: the reverse conversion from x to longitude. similarly to the latitude, longitude to x,y conversion you must pair x and y. e.g.,

  ```
  <message_var algorithm="utm_x2lon:our_y">our_x</message_var}
  ```
- `utm_y2lat`: example:

  ```
  <message_var algorithm="utm_y2lat:our_x">our_y</message_var}
  ```
- `modem_id2name`: converts a WHOI `modem_id` to a vehicle name. requires a file (path given in the .moos as `modem_id_lookup_path: "/path/to/modemidlookup.txt"`. an example file:

[2] we define a latitude/longitude origin near our basis of operations. From this datum we calculate the UTM northings (y) and eastings (x). All further UTM calculations are the offset from this datum point. This offset is what is returned by this algorithm. Contact me if you need more information on this.

```
// modem_id, vehicle name (should be community name), vehicle type
0, broadcast, broadcast
1, endeavor, ship
3, unicorn, auv
4, macrura, auv
```

if no match is found, the modem_id is returned as a string (e.g. ”10”).

- `name2modem_id`: performs the (case insensitive) reverse lookup on the same file. if no match is found, `atoi(name.c_str())` is returned (probably zero unless you passed something like ”4” to this function).
- `modem_id2type`: similar to `modem_id2name` but returns the type of the vehicle (ship, auv, etc.)
- `power_to_dB`: takes $10 \log_{10}$ of the value.
- `dB_to_power`: takes power antilog of the value.
- `alg_TSD_to_soundspeed`: applied to temperature, with references to salinity and depth, calculates the speed of sound using the Mackenzie equation. For example:

  `<message_var algorithm="alg_TSD_to_soundspeed:sal:depth">temp</message_var>`

- `add`: adds the reference `<message_var>` to the current `<message_var>`. example: `<message_var algorithm="add:b">a</message_var>` adds b to a.
- `subtract`: subtracts the reference `<message_var>` from the current `<message_var>`.

2.3.11 DCCL Encoding/Decoding Unit: Under the Hood

See `http://gobysoft.com/doc/1.0/acomms__dccl.html#dccl_how` and [5] for details on how the DCCL encoding is done.

2.3.12 Priority Message Queuing Unit

pAcommsHandler takes all the configured queues and maintains a stack of messages for each queue. when it is prompted by data by the modem, it has a priority ”contest” between the queues. the queue with the current highest priority (as determined by the `value_base` and `ttl` fields) is selected. The next message in that queue is then provided to the MicroModem to send. For modem messages with multiple frames per packet, each frame is a separate contest. Thus a single packet may contain frames from different queues (e.g. a rate 5 PSK packet has eight 256 byte frames. frame 1 might grab a STATUS message since that has the current highest queue. then frame 2 may grab a BTR message and frames 3-8 are filled up with CTD messages (e.g. STATUS is in blackout, BTR queue is empty)). See http://gobysoft.com/doc/1.0/acomms__queue.html#queue_priority for more

For messages with `ack: true` (acknowledge requested), the last message continues to be re-sent (that is, it is not popped from the message queue) until the ACK is received from the modem (thus blocking the sending of other messages). Messages with `ack: false` are popped and discarded when they are sent (no retries).

If you do not wish for dynamic growth of the priorities, simply set the `ttl` to the special value 0. Then the priorities grow as $P = V\_base$ and messages never expire. Note that this is the same as setting `ttl` = ∞.

Messages not to us are ignored We choose modem id 0 as broadcast. thus messages with the destination field = 0 will always be read by all nodes and reported to the appropriate moos variable. Otherwise, we ignore messages unless they correspond to our modem id. so if you send a message to modem id 10, pAcommsHandler for modem ids $1 \rightarrow 9$, $11 \rightarrow N$ will ignore that. This is not the default behavior of the WHOI Micro-Modem, which always reports data, regardless of the sender's ID.

The XML tag reference is now part of the Goby Developers documentation (http://gobysoft.com/doc/1.0/:

- See http://gobysoft.com/doc/1.0/acomms__queue.html#queue_tags for a structure of all the allowed tags.
- http://gobysoft.com/doc/1.0/acomms__queue.html#queue_tags_details provides an up-to-date reference of all the Queue tags with a description of their usage.

2.3.13 Modem Driver Unit

The Modem driver unit current supports the WHOI Micro-Modem acoustic modem and is extensible to other acoustic modems. To directly monitor the modem feed, subscribe to ACOMMS_NMEA_IN and ACOMMS_NMEA_OUT. For a complete list of supported commands of the WHOI Micro-Modem, see http://gobysoft.com/doc/1.0/acomms__driver.html#acomms_mmdriver.

2.3.14 Medium Access Control (MAC) Unit

The MAC unit uses time division (TDMA) to attempt to ensure a collision-free acoustic channel.

pAcommsHandler supports two variants of the TDMA MAC scheme: centralized and decentralized. As the names suggest, Centralized TDMA (`type: MAC_POLLED`) involves control of the entire cycle from a single master node, whereas each node's respective slot is controlled by that node in Decentralized TDMA. Within decentralized TDMA, Goby supports both a fixed (preprogrammed) cycle (`type: MAC_FIXED_DECENTRALIZED`) and an autodiscovery mode (`type: MAC_AUTO_DECENTRALIZED`). To disable the pAcommsHandler MAC, use (`type: MAC_NONE`)

Centralized TDMA (Polling)

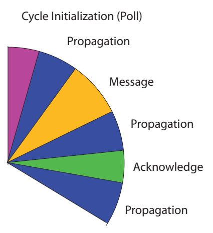
Centralized TDMA involves a master node (usually aboard the Research Vessel or on land) which initiates every transmission for the entire communcations cycle (i.e. "polls" each node for data). Thus, the other nodes are not required to maintain synchronized clocks as the timing is all performed on the master node.

This style of MAC has been widely used for small AUV operations using the WHOI Micro-Modem. Its principal advantages are that it has 1) no requirement for synchronized clocks, 2) full control over the communications cycle at runtime (assuming the master is accessible to the vehicle operators, as is usually the case); and 3) a master who can acknowledge "broadcast" messages.

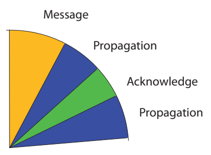
However, centralized TDMA has a number of substantial disadvantages. In order for a third-party master to initiate a transmission, an acoustic packet must be sent for this initialization. This additional "cycle initialization" packet, like any acoustic message, has a high chance of being lost (after which the data are never sent because the sending node did not receive a cycle initialization message), consumes power, and lengthens the time of the communications slot. See Fig. 2.6 for the various parts of the communication cycle with (for Centralized TDMA) and without (for Decentralized TDMA) the cycle initialization message. The additional time required for each slot of Centralized TDMA is

$$\tau_{ci} + r_{max}/c \tag{2.1}$$

where τ_{ci} is the length (in seconds) of the cycle initalization packet (about one second for the WHOI Micro-Modem), r_{max} is the maximum range of the network (typically of order 1000s of meters), and c is the compressional speed of sound (nominally 1500 m/s).



(a) Centralized TDMA



(b) Decentralized TDMA

Figure 2.6: Comparison of the time needed for a single slot for the two types of TDMA supported by pAcommsHandler. Eq. 2.1 gives the additional length of time required by the Centralized variant.

Decentralized TDMA with passive auto-discovery

Decentralized TDMA removes the cycle initialization packet and thus reduces the length of each slot and the chance of errors. However, it introduces the constraint of synchronized clocks[3] for all nodes, which can be somewhat tricky to maintain underwater.

Decentralized TDMA gives each vehicle a single slot in 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 the time of day). All slots are ordered by ascending acoustic MAC address (or "modem identification number"), which is an unsigned integer unique for each network.

During the runtime of the network, it is often desirable to add or remove nodes. Since the MAC is spread throughout the nodes, there is no easy way to change the cycle during runtime. *libamac* supports passive auto-discovery (and subsequent expiration) of nodes to provide a solution to this problem. This auto-discovery is passive because it requires no control messaging beyond the normal communications between nodes.

Vehicles are discovered by shifting a blank slot in each cycle based on their knowledge of the world and the time of day. If a new vehicle is heard from during the blank, it is added to the listening vehicle's knowledge of the world and hence their cycle. In the simplified situation (which is really a worst case scenario) discovery is defined by a single vehicle transmitting during a cycle and all the others silent (the current slot is not equal to each vehicle's acoustic MAC address).

[3]the accuracy of the clock synchronization can be low relative to other timing needs such as bistatic sonar. Generally, accuracy better than 0.1 seconds is acceptable; higher inaccuracies can be handled by increasing the guard time on both sides of each slot.

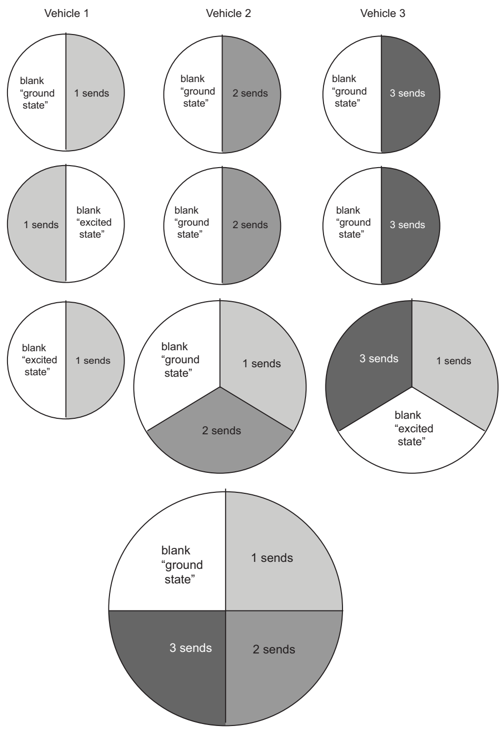


Figure 2.7: Graphical example of auto discovery for three nodes launched at the same time. Each circle represents the vehicle's cycle at each time step (represented by horizontal rows) based on the vehicle's current knowledge of the world. In the first row, all vehicles only know of themselves and put the blank slot in the last slot; thus, all communications collide and no discoveries are made. In the second row, vehicle 1's blank is moved (by pseudo-chance) to the penultimate (first) slot, so vehicles 2 and 3 discover 1. Then, in the third row vehicles 2 and 3 are discovered by the others because vehicle 3 moves its blank slot. By the fourth row all vehicles have discovered the others and continue to transmit without collision following the cycle diagrammed on this row.

time	vehicle 1	vehicle 2	result
0	send	send	collision
15	blank	blank	nothing
30	blank	send	success: 1 discovers 2
45	cycle wait	blank	nothing
60	cycle wait	send	success
75	cycle wait	blank	nothing
90	send	blank	success: 2 discovers 1
105	listen for 2	cycle wait	nothing
120	blank	cycle wait	nothing
135	send	listen for 1	success
150	listen for 2	send	success
165	blank	blank	nothing
180	send	listen for 1	success
195	blank	blank	nothing
210	listen for 2	send	success

Table 2.1: Example initialization for the Decentralized TDMA with autodiscovery. By 135 seconds, both vehicles have discovered each other and are synchronized. Thus, no more collisions will occur. This scenario assumes that both vehicles always have some data to send during their slot.

2.3.15 Simple complete example MOOS files

Example 1: Basic CCL (goby/share/cfg/MOOS/basic_ccl)

This example sends the bytes `0x020304` from node 1 (`mm1`) to node 2 (`mm2`). It shows use of all the parts of pAcommsHandler except the DCCL encoding / decoding unit. I use `iModemSim` here to simulate the WHOI Micro-Modem. This process is available in moos-ivp-local (http://oceanai.mit.edu/moos-ivp/pmwiki/pmwiki.php?n=Support.Milocal). You can also easily substitute real modems by removing iModemSim references and changing the `serial_port`.

MOOS file for Node 1: goby/share/cfg/MOOS/basic_ccl/mm1.moos

```
// t. schneider tes@mit.edu 2.16.11

// bare bones acoustic communications
// stack for topside receiver
// for CCL message

ServerHost = localhost
ServerPort = 9101
Community  = mm1

LatOrigin = 0
LongOrigin = 0

ProcessConfig = ANTLER
{
    MSBetweenLaunches = 10
    Run = MOOSDB @ NewConsole = false

    /////////////////////////////////////
    // acomms related
    /////////////////////////////////////
    // queuing
    Run = pAcommsHandler        @ NewConsole = true
    // modem simulator
    Run = iModemSim             @ NewConsole = true

    // simulate CCL data source
    Run = uTimerScript            @ NewConsole = true
}

ProcessConfig = pAcommsHandler
{
    modem_id: 1

    driver_type: DRIVER_WHOI_MICROMODEM

    driver_cfg
    {
        serial_port: "/tmp/ttyLOOPA2"
    # doesn't work with iModemSim, set to true for real ops
        [MicroModemConfig.reset_nvram]: false
    }

    mac_cfg
    {
        type: MAC_FIXED_DECENTRALIZED
        slot
        {
            src: 1
            dest: 2
            rate: 0
            type: SLOT_DATA
            slot_seconds: 10
        }
    }

    queue_cfg
    {
        queue
        {
            key {
                type: QUEUE_CCL
                id: 2 # decimal CCL id (first byte)
            }
            in_pubsub_var: "IN_TEST_32B"
            out_pubsub_var: "OUT_TEST_32B"
            name: "TEST"
        }
    }
}
// must set serial_loopbacks to use
// as root run the shell script (in moos-ivp-local/scripts)
// > loopbacks
ProcessConfig = iModemSim
{
        AppTick   = 4
CommsTick = 4

Port = /tmp/ttyLOOPA1
Speed = 19200

IPPort = 49234
BroadcastAddr = 127.0.0.1

        InputLocType = constant_local
        ConstantPosX = 0
        ConstantPosY = 0
        ConstantDepth = 0
}



ProcessConfig = uTimerScript
{
    // data is 2 2 3 4 in octal
    EVENT  = var=OUT_TEST_32B, val="data: "\002\002\003\004"", time = 10
    RESET_TIME = end
}
```

MOOS file for Node 2: goby/share/cfg/MOOS/basic_ccl/mm2.moos

```
// t. schneider tes@mit.edu 4.28.10

// bare bones acoustic communications
// stack for auv
// for CCL message

ServerHost = localhost
ServerPort = 9102
Community  = mm2

LatOrigin = 0
LongOrigin = 0

ProcessConfig = ANTLER
{
    MSBetweenLaunches = 10
    Run = MOOSDB @ NewConsole = false

    /////////////////////////////////////
    // acomms related
    /////////////////////////////////////
    // queuing
    Run = pAcommsHandler        @ NewConsole = true

    Run = iModemSim             @ NewConsole = true
}

ProcessConfig = pAcommsHandler
{
    modem_id: 2

    driver_type: DRIVER_WHOI_MICROMODEM

    driver_cfg
    {
        serial_port: "/tmp/ttyLOOPB2"
    # doesn't work with iModemSim, set to true for real ops
        [MicroModemConfig.reset_nvram]: false
    }

    mac_cfg
    {
        type: MAC_FIXED_DECENTRALIZED
        slot
        {
            src: 1
            dest: 2
            rate: 0
            type: SLOT_DATA
            slot_seconds: 10
        }
    }

    queue_cfg
    {
        queue
        {
            key {
                type: QUEUE_CCL
                id: 2  # decimal CCL id (first byte)
            }
            in_pubsub_var: "IN_TEST_32B"
            out_pubsub_var: "OUT_TEST_32B"
            name: "TEST"
        }
    }
}


// must set serial_loopbacks to use
// as root run the shell script (in moos-ivp-local/src/bin)
// > loopbacks
ProcessConfig = iModemSim
{
AppTick   = 4
CommsTick = 4

Port = /tmp/ttyLOOPB1
Speed = 19200

IPPort = 49234
BroadcastAddr = 127.0.0.1

        InputLocType = constant_local
        ConstantPosX = 0
        ConstantPosY = 0
        ConstantDepth = 0
}
```

Example 2: DCCL and CCL (goby/share/cfg/MOOS/ccl_and_dccl)

This example sends the DCCL "Simple Status" messsage from node 1 (`mm1`) to node 2 (`mm2`). `mm2` sends the REMUS CCL State message to `mm1`. It thus uses all the components of pAcommsHandler. As in the previous example, you can use real modems by removing iModemSim and changing the `serial_port` to the proper real serial port.

MOOS file for Node 1: goby/share/cfg/MOOS/ccl_and_dccl/mm1.moos

```
// t. schneider tes@mit.edu 3.2.11

// bare bones acoustic communications
// stack for topside receiver

ServerHost = localhost
ServerPort = 9101
Community  = mm1

LatOrigin =  42.35
LongOrigin = -70.95

NoNetwork = true
modem_id_lookup_path = modemidlookup.txt


ProcessConfig = ANTLER
{
    MSBetweenLaunches = 10
    Run = MOOSDB @ NewConsole = false

    Run = pREMUSCodec           @ NewConsole = true, XConfig=1
    Run = pAcommsHandler        @ NewConsole = true, XConfig=2
    Run = iModemSim             @ NewConsole = true, XConfig=3

    1 = -geometry,80x15+0+0
    2 = -geometry,80x100+0+230
    3 = -geometry,80x15+0+570
}
```

```

ProcessConfig = pREMUSCodec
{
  mdat_state_var: "IN_REMUS_STATUS"
  mdat_state_out: "OUT_REMUS_STATUS"
  create_status: false
}


ProcessConfig = pAcommsHandler
{
    common
    {
      verbosity: VERBOSITY_GUI
        initializer { type: INI_DOUBLE  global_cfg_var: "LatOrigin"  moos_var: "LAT_ORIGIN" }
        initializer { type: INI_DOUBLE  global_cfg_var: "LongOrigin" moos_var: "LONG_ORIGIN" }
        initializer { type: INI_STRING  moos_var: "VEHICLE_TYPE" sval: "topside" }
        initializer { type: INI_STRING  moos_var: "VEHICLE_NAME" sval: "mm1" }
        initializer { type: INI_DOUBLE  moos_var: "NAV_X"  dval: 100 }
        initializer { type: INI_DOUBLE  moos_var: "NAV_Y"  dval: 300 }
        initializer { type: INI_DOUBLE  moos_var: "NAV_HEADING" dval: 150 }
        initializer { type: INI_DOUBLE  moos_var: "NAV_SPEED" dval: 0 }
        initializer { type: INI_DOUBLE  moos_var: "NAV_DEPTH" dval: 0 }
    }

    modem_id: 1

    driver_type: DRIVER_WHOI_MICROMODEM
    driver_cfg
    {
      serial_port: "/tmp/ttyLOOPA2"
# doesn't work with iModemSim, set to true for real ops
      [MicroModemConfig.reset_nvram]: false
    }

    mac_cfg
    {
        type: MAC_FIXED_DECENTRALIZED
        slot { src: 1  dest: 2  rate: 0  type: SLOT_DATA  slot_seconds: 10 } # downlink
        slot { src: 2  dest: 1  rate: 0  type: SLOT_DATA  slot_seconds: 10 } # uplink
    }

    queue_cfg
    {
        queue
        {
            key {
                type: QUEUE_CCL
                id: 14 # decimal CCL id (first byte)
            }
            in_pubsub_var: "IN_REMUS_STATUS"
            out_pubsub_var: "OUT_REMUS_STATUS"
            name: "Remus_State"
        }
    }

    dccl_cfg
    {
        message_file { path: "../../../xml/simple_status.xml" }
    }
}

// must set serial_loopbacks to use
// as root run the shell script (in moos-ivp-local/src/bin)
// > loopbacks
ProcessConfig = iModemSim
{
AppTick   = 4
CommsTick = 4

Port = /tmp/ttyLOOPA1
Speed = 19200

IPPort = 49234
BroadcastAddr = 127.0.0.1

        InputLocType = constant_local
        ConstantPosX = 0
        ConstantPosY = 0
        ConstantDepth = 0
}

```

MOOS file for Node 2: goby/share/cfg/MOOS/ccl_and_dccl/mm2.moos

```
// t. schneider tes@mit.edu 3.2.11

// bare bones acoustic communications
// stack for auv

```

```
ServerHost = localhost
ServerPort = 9102
Community  = mm2

LatOrigin =  42.35
LongOrigin = -70.95

modem_id_lookup_path = modemidlookup.txt
modem_id = 2

NoNetwork = true

ProcessConfig = ANTLER
{
    MSBetweenLaunches = 10

    Run = MOOSDB @ NewConsole = false

    Run = pREMUSCodec           @ NewConsole = true, XConfig=1
    Run = pAcommsHandler        @ NewConsole = true, XConfig=2
    Run = iModemSim             @ NewConsole = true, XConfig=3

    1 = -geometry,80x15-0+0
    2 = -geometry,80x100-0+230
    3 = -geometry,80x15-0+570
}

ProcessConfig = pREMUSCodec
{
  create_status: true

  mdat_state_var: "IN_REMUS_STATUS"
  mdat_state_out: "OUT_REMUS_STATUS"
  modem_id_lookup_path: "modemidlookup.txt"
}

ProcessConfig = pAcommsHandler
{
    common
    {
      verbosity: VERBOSITY_GUI
        initializer { type: INI_DOUBLE  global_cfg_var: "LatOrigin"  moos_var: "LAT_ORIGIN" }
        initializer { type: INI_DOUBLE  global_cfg_var: "LongOrigin" moos_var: "LONG_ORIGIN" }
        initializer { type: INI_STRING  moos_var: "VEHICLE_TYPE"  sval: "auv" }
        initializer { type: INI_STRING  moos_var: "VEHICLE_NAME"  sval: "mm2" }
        initializer { type: INI_DOUBLE  moos_var: "NAV_X"  dval: 123 }
        initializer { type: INI_DOUBLE  moos_var: "NAV_Y"  dval: 321 }
        initializer { type: INI_DOUBLE  moos_var: "NAV_HEADING" dval: 45 }
        initializer { type: INI_DOUBLE  moos_var: "NAV_SPEED" dval: 1.2 }
        initializer { type: INI_DOUBLE  moos_var: "NAV_DEPTH" dval: 111 }
    }

    modem_id: 2
    modem_id_lookup_path: "modemidlookup.txt"

    driver_type: DRIVER_WHOI_MICROMODEM
    driver_cfg
    {
        serial_port: "/tmp/ttyLOOPB2"
    # doesn't work with iModemSim, set to true for real ops
        [MicroModemConfig.reset_nvram]: false
    }

    mac_cfg
    {
        type: MAC_FIXED_DECENTRALIZED
        slot { src: 1  dest: 2  rate: 0  type: SLOT_DATA  slot_seconds: 10 } # downlink
        slot { src: 2  dest: 1  rate: 0  type: SLOT_DATA  slot_seconds: 10 } # uplink
    }

    queue_cfg
    {
        queue
        {
            key { type: QUEUE_CCL id: 14 }
            in_pubsub_var: "IN_REMUS_STATUS"
            out_pubsub_var: "OUT_REMUS_STATUS"
            name: "Remus_State"
        }
    }

    dccl_cfg
    {
        message_file { path: "../../../xml/simple_status.xml"
                       manipulator: NO_ENCODE }
    }
}

// must set serial_loopbacks to use
// as root run the shell script (in moos-ivp-local/src/bin)
// > loopbacks
ProcessConfig = iModemSim
```

```
{
AppTick   = 4
CommsTick = 4

Port = /tmp/ttyLOOPB1
Speed = 19200

IPPort = 49234
BroadcastAddr = 127.0.0.1

        InputLocType = constant_local
        ConstantPosX = 0
        ConstantPosY = 0
        ConstantDepth = 0
}
```

XML definition of Simple Status: goby/xml/simple_status.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <name>SIMPLE_STATUS</name>
    <trigger>time</trigger>
    <trigger_time>5</trigger_time>
    <size>32</size>
    <header>
      <id>20</id>
      <time>
        <name>Timestamp</name>
      </time>
      <src_id algorithm="to_lower,name2modem_id">
        <name>Node</name>
        <moos_var>VEHICLE_NAME</moos_var>
      </src_id>
    </header>
    <layout>
      <static>
        <name>MessageType</name>
        <value>LAMSS_STATUS</value>
      </static>
      <float>
        <name>nav_x</name>
        <moos_var>NAV_X</moos_var>
        <max>100000</max>
        <min>-100000</min>
        <precision>0</precision>
      </float>
      <float>
        <name>nav_y</name>
        <moos_var>NAV_Y</moos_var>
        <max>100000</max>
        <min>-100000</min>
        <precision>0</precision>
      </float>
      <float>
        <name>Speed</name>
        <moos_var>NAV_SPEED</moos_var>
        <max>20</max>
        <min>-2</min>
        <precision>1</precision>
      </float>
      <float algorithm="angle_0_360">
        <name>Heading</name>
        <moos_var>NAV_HEADING</moos_var>
        <max>360</max>
        <min>0</min>
        <precision>2</precision>
      </float>
      <float>
        <name>Depth</name>
        <moos_var>NAV_DEPTH</moos_var>
        <max>6400</max>
        <min>0</min>
        <precision>1</precision>
      </float>
    </layout>

    <!-- decoding -->
    <on_receipt>
      <publish>
        <moos_var>STATUS_REPORT_IN</moos_var>
        <all />
      </publish>
      <publish>
        <moos_var>NODE_REPORT</moos_var>
        <format>NAME=%1%,TYPE=%2%,UTC_TIME=%3$.0lf,X=%4%,Y=%5%,LAT=%6$lf,LON=%7$lf,SPD=%8%,HDG=%9%,DEP
        <message_var algorithm="modem_id2name">Node</message_var>
<message_var algorithm="modem_id2type">Node</message_var>
        <message_var>Timestamp</message_var>
        <message_var>nav_x</message_var>
```

```
          <message_var>nav_y</message_var>
          <message_var algorithm="utm_y2lat:nav_x">nav_y</message_var>
          <message_var algorithm="utm_x2lon:nav_y">nav_x</message_var>
          <message_var>Speed</message_var>
          <message_var>Heading</message_var>
          <message_var>Depth</message_var>
        </publish>
      </on_receipt>
      <queuing>
        <ack>false</ack>
        <blackout_time>10</blackout_time>
        <ttl>300</ttl>
        <value_base>1.5</value_base>
      </queuing>
    </message>
  </message_set>

```

Modem Lookup Table: goby/share/cfg/MOOS/ccl_and_dccl/modemidlookup.txt

```
1,mm1,topside
2,mm2,auv
```

2.4 iCommander

`iCommander` is a topside command and control (C2) tool which provides a simple console for issuing commands through the acoustic network. By sharing DCCL message configuration (XML) files with `pAcommsHandler` it automatically adapts to the current message set, without any need to change code.

Parameters for the iCommander Configuration Block

Example .moos file The moos file is simple since the bulk of the configuration is stored in separate XML files (see section 2.3.8 for the configuration of these files):

```
ProcessConfig = iCommander
{
  common {  # Configuration common to all Goby MOOS applications
            # (opt)
    log: true  # Should we write a text log of the terminal
```

```
                # output? (opt) (default=true) (can also set MOOS
                # global "log=")
    log_path: "./"  # Directory path to write the text log of the
                    # terminal output (if log=true) (opt)
                    # (default="./") (can also set MOOS global
                    # "log_path=")
    community: "AUV23"  # The vehicle's name (opt) (can also set
                        # MOOS global "Community=")
    lat_origin: 42.5  # Latitude in decimal degrees of the local
                      # cartesian datum (opt) (can also set MOOS
                      # global "LatOrigin=")
    lon_origin: 10.9  # Longitude in decimal degrees of the local
                      # cartesian datum (opt) (can also set MOOS
                      # global "LongOrigin=")
    app_tick: 10  # Frequency at which to run Iterate(). (opt)
                  # (default=10)
    comm_tick: 10  # Frequency at which to call into the MOOSDB
                   # for mail. (opt) (default=10)
    verbosity: VERBOSITY_VERBOSE  # Verbosity of the terminal
                                  # window output (VERBOSITY_QUIET,
                                  # VERBOSITY_WARN,
                                  # VERBOSITY_VERBOSE,
                                  # VERBOSITY_DEBUG, VERBOSITY_GUI)
                                  # (opt)
                                  # (default=VERBOSITY_VERBOSE)
    initializer {  # Publish a constant value to the MOOSDB at
                   # startup (repeat)
      type: INI_DOUBLE  # type of MOOS variable to publish
                        # (INI_DOUBLE, INI_STRING) (req)
      moos_var: "SOME_MOOS_VAR"  # name of MOOS variable to
                                 # publish to (req)
      global_cfg_var: "LatOrigin"  # Optionally, instead of
                                   # giving `sval` or `dval`, give
                                   # a name here of a global MOOS
                                   # variable (one at the top of
                                   # the file) whose contents
                                   # should be written to
                                   # `moos_var` (opt)
      dval: 3.454  # Value to write for type==INI_DOUBLE (opt)
      sval: "a string"  # Value to write for type==INI_STRING
                        # (opt)
    }
  }
  dccl_cfg {  # Configure the DCCL Encoder (opt)
    modem_id: 1  # Unique number 1-31 to identify this node (req)
    message_file {  # XML message file containing one or more
```

```
                     # DCCL message descriptions (repeat)
      path: "/home/toby/goby/src/acomms/examples/chat/chat.xml"
                                                # path to the
                                                # message XML file
                                                # (req)
      manipulator: NO_MANIP  # manipulators to modify the
                             # encoding and queuing behavior of the
                             # messages in this file (NO_MANIP,
                             # NO_ENCODE, NO_DECODE, NO_QUEUE,
                             # LOOPBACK, ON_DEMAND, TCP_SHARE_IN)
                             # (repeat)
    }
    crypto_passphrase: "twinkletoes%24"  # If given, encrypt all
                                         # communications with this
                                         # passphrase using AES.
                                         # Omit for unencrypted
                                         # communications. (opt)
  }
  modem_id_lookup_path: ""  # Path to file containing mapping
                            # between modem_id and vehicle name &
                            # type (opt) (can also set MOOS global
                            # "modem_id_lookup_path=")
  load: ""  # Path to iCommander save file to load automatically
            # on startup (repeat)
  show_variable: ""  # MOOS Variable to scope on the GUI (repeat)
  force_xy_only: false  # Set true to set all Latitude/Longitude
                        # fields to use x/y values instead (opt)
                        # (default=false)
}
```

As with pAcommsHandler, the above configuration file can be generated at any time with the command:

```
iCommander --example_config
```

Filling out the .moos file Some of the DCCL configuration (`dccl_cfg`) parameters are not used, such as the `crypto_passphrase`.

- `common`: See section 2.3.5.
- `dccl_cfg.message_file`: path to an XML file containing a message set of one or messages. These are the DCCL messages. You can also load messages XML files through the Main Menu in the program.

- `load`: path to a file of iCommander saved message(s) to load automatically on startup. You can also load messages through the Main Menu in the program.

Reference Sheet

Main Menu

```
 -------------------------------------------------------------
|            iCommander: Vehicle Command Message Sender        |
|                        2 messages loaded.                    |
|    Main Menu:                                                |
|    > Return to active message                                |
|    > Select Message                                          |
|    > Load                                                    |
|    > Save                                                    |
|    > Import Message File                                     |
|    > Exit                                                    |
|______________________________________________________________|
```

- *Return to active message* - only available if you have actively edited a message this session. Choose to return to the editing screen of the last message you were editing.
- *Select Message* - pick a message type to edit. All messages are read from DCCL (dynamic compact control language) XML message files.
- *Load* - load a saved message parameters file. This allows you to save values for message fields from session to session.
- *Save* - saves all open messages to a single file for later use. These files are plain text for easy use outside iCommander.
- *Import Message File* - import another DCCL XML file for use.
- *Exit* - quit cleanly.

Editing screen

```

 --------------------------------------------------
|                                                  |
|Editing message variable 1 of 22: MessageType     |
     |(static) you cannot change the value of this field|
     |__________________________________________________|


    __________________________________________________
   |                                                  |
   |Message (Type: SENSOR_PROSECUTE)                  |
   |22 entries total                                  |
   |        {Enter} for options                       |
   |        {Up/Down} for more message variables      |
   |                                                  |
   |                                 ________________ |
   |                                |                ||
   |1. MessageType (static)         |SENSOR_PROSECUTE ||
   |                                |________________||
   |                                 ________________ |
   |                                |                ||
   |2. SensorCommandType (int)      |1               ||
   |                                |________________||
   |                                 ________________ |
   |                                |                ||
   |3. SourcePlatformId (int)       |0               ||
   |                                |________________||
   |                                 ________________ |
   |                                |                ||
   |4. DestinationPlatformId (int)  |3               ||
   |                                |________________||
   |__________________________________________________|
```

Scroll to select the box to edit. Note that you will need to scroll up or down off the screen to see all the fields at once. The information box at the top will tell you how large the field can be based on the DCCL settings. You cannot enter a value outside these ranges. Hit enter to get the editing menu.

Editing menu

```
         __________________________________________________________
        |                                                          |
        |                     Choose an action                     |
        |> Return to message                                       |
        |> Send                                                    |
        |> Preview                                                 |
        |> Quick switch to another open message                    |
        |> Insert special: current time                            |
        |> Insert special: local X,Y to longitude,latitude         |
        |> Insert special: community                             |
        |> Insert special: modem id                              |
        |> Clear message                                         |
        |> Main Menu                                             |
        |                                                        |
        |                                                        |
        |________________________________________________________|
```

- *Return to message*
- *Send* - publish the variables for use by pAcommsHandler
- *Preview* - preview the message to be sent in exact syntactical form
- *Quick switch to another open message* - switch to another message with information (either edited this session or loaded)
- *Insert special: current time* - insert a placeholder ("_time") that will be replaced with the current UNIX time when message is sent (e.g. 1236053988). Shortcut: type 't' directly into the field and bypass this menu.
- *Insert special: local X,Y to longitude,latitude* - insert a placeholder designator to do a UTM local grid to latitude / longitude conversion. first the latitude (Y or northings) is entered ("y(lat)1:"), then you choose where to put the longitude (X or eastings) ("x(lon)1:"). after the colon enter the desired value in meters that will be converted to latitude/longitude based in the LatOrigin/LongOrigin set in the top of the MOOS file. Note that you may have more than one pair of x/y. This is the reason for the number following "y(lat)"/"x(lon)". "y(lat)1" is paired with "x(lon)1", "y(lat)2" is paired with "x(lon)2", etc. Shortcut: type 'y' or 'x' respectively directly into the fields and bypass this menu.
- *Insert special: community* - insert the name of this MOOS community.
- *Insert special: modem id* - choose a modem id from a list of names. This is based off the modem id lookup table used by pAcommsHandler.
- Clear message
- Main Menu

Acknowledgments If you are using pAcommsHandler with the ACK field set to 1 (true), all acoustic message acknowledgments are displayed at the top of the screen. For example, the ack of a `LAMSS_DEPLOY` message would look like this:

```
 ---------------------------------------------
|                                             |
|Message acknowledged from queue: LAMSS_DEPLOY|
| for destination: 5                          |
| at time: 2011-Mar-03 22:38:12               |
|_____________________________________________|
```

Similarly, expired messages (messages that exceed their *ttl* without being sent) are shown as well:

```
 ---------------------------------------------
|                                             |
|Message expired from queue: LAMSS_DEPLOY     |
| for destination: 5                          |
| at time: 2011-Mar-03 22:38:12               |
|_____________________________________________|
```

2.5 pREMUSCodec

Example .moos file

```
ProcessConfig = pREMUSCodec
{
  common {  # Configuration common to all Goby MOOS applications
            # (opt)
    log: true  # Should we write a text log of the terminal
               # output? (opt) (default=true) (can also set MOOS
               # global "log=")
    log_path: "./"  # Directory path to write the text log of the
                    # terminal output (if log=true) (opt)
                    # (default="./") (can also set MOOS global
                    # "log_path=")
    community: "AUV23"  # The vehicle's name (opt) (can also set
                        # MOOS global "Community=")
    lat_origin: 42.5  # Latitude in decimal degrees of the local
                      # cartesian datum (opt) (can also set MOOS
                      # global "LatOrigin=")
    lon_origin: 10.9  # Longitude in decimal degrees of the local
                      # cartesian datum (opt) (can also set MOOS
                      # global "LongOrigin=")
    app_tick: 10  # Frequency at which to run Iterate(). (opt)
                  # (default=10)
    comm_tick: 10  # Frequency at which to call into the MOOSDB
                   # for mail. (opt) (default=10)
    verbosity: VERBOSITY_VERBOSE  # Verbosity of the terminal
                                  # window output (VERBOSITY_QUIET,
                                  # VERBOSITY_WARN,
                                  # VERBOSITY_VERBOSE,
                                  # VERBOSITY_DEBUG, VERBOSITY_GUI)
                                  # (opt)
                                  # (default=VERBOSITY_VERBOSE)
    initializer {  # Publish a constant value to the MOOSDB at
                   # startup (repeat)
      type: INI_DOUBLE  # type of MOOS variable to publish
                        # (INI_DOUBLE, INI_STRING) (req)
      moos_var: "SOME_MOOS_VAR"  # name of MOOS variable to
                                 # publish to (req)
      global_cfg_var: "LatOrigin"  # Optionally, instead of
                                   # giving `sval` or `dval`, give
                                   # a name here of a global MOOS
                                   # variable (one at the top of
                                   # the file) whose contents
                                   # should be written to
                                   # `moos_var` (opt)
      dval: 3.454  # Value to write for type==INI_DOUBLE (opt)
      sval: "a string"  # Value to write for type==INI_STRING
                        # (opt)
    }
  }
  create_status: false  # Will generate REMUS State message if
                        # true (opt) (default=false)
  mdat_state_var: "IN_REMUS_STATUS_HEX_30B"  # MOOS variable for
                                             # incoming REMUS state
                                             # messages (raw) (opt)
                                             # (default="IN_REMUS_ST
                                             # ATUS_HEX_30B")
  mdat_state_out: "OUT_REMUS_STATUS_HEX_30B"  # MOOS variable for
                                              # outgoing REMUS
                                              # state messages
                                              # (raw) (opt)
                                              # (default="OUT_REMUS_
                                              # STATUS_HEX_30B")
  mdat_ranger_var: "IN_REMUS_RANGER_HEX_30B"  # MOOS variable for
```

```
                                                # incoming REMUS
                                                # ranger messages
                                                # (raw) (opt)
                                                # (default="IN_REMUS_R
                                                # ANGER_HEX_30B")
    mdat_ranger_out: "OUT_REMUS_RANGER_HEX_30B"
                                                # MOOS variable for
                                                # outgoing REMUS
                                                # ranger messages
                                                # (raw) (opt)
                                                # (default="OUT_REMUS_
                                                # RANGER_HEX_30B")
    mdat_redirect_var: "IN_REMUS_REDIRECT_HEX_30B"
                                                # MOOS variable for
                                                # incoming REMUS
                                                # redirect messages
                                                # (raw) (opt)
                                                # (default="IN_REMUS_R
                                                # EDIRECT_HEX_30B")
    mdat_redirect_out: "OUT_REMUS_REDIRECT_HEX_30B"
                                                # MOOS variable for
                                                # outgoing REMUS
                                                # redirect messages
                                                # (raw) (opt)
                                                # (default="OUT_REMUS_
                                                # REDIRECT_HEX_30B")
    mdat_alert_var: "IN_REMUS_ALERT_HEX_30B"  # MOOS variable for
                                              # incoming REMUS alert
                                              # messages (raw) (opt)
                                              # (default="IN_REMUS_ALE
                                              # RT_HEX_30B")
    mdat_alert_out: "OUT_REMUS_ALERT_HEX_30B"  # MOOS variable for
                                               # outgoing REMUS alert
                                               # messages (raw) (opt)
                                               # (default="OUT_REMUS_A
                                               # LERT_HEX_30B")
    mdat_alert2_var: "IN_REMUS_ALERT2_HEX_30B"  # MOOS variable for
                                                # incoming REMUS
                                                # alert2 messages
                                                # (raw) (opt)
                                                # (default="IN_REMUS_A
                                                # LERT2_HEX_30B")
    mdat_alert2_out: "OUT_REMUS_ALERT2_HEX_30B"
                                                # MOOS variable for
                                                # outgoing REMUS
                                                # alert2 messages
```

```
                                                 # (raw) (opt)
                                                 # (default="OUT_REMUS_
                                                 # ALERT2_HEX_30B")
  modem_id_lookup_path: ""  # Path to file containing mapping
                            # between modem_id and vehicle name &
                            # type (opt) (can also set MOOS global
                            # "modem_id_lookup_path=")
}
```

As with pAcommsHandler, the above configuration file can be generated at any time with the command:

```
pREMUSCodec --example_config
```

This codec handles several of the standard REMUS CCL messages. It can be configured to generate CCL State messages at regular intervals, and it will translate incoming CCL State messages into the standard `NODE_REPORT` format used internally in the LAMSS autonomy systems. This codec allows a MOOS vehicle to perform collaborative behaviors, such as collision avoidance, with a non-MOOS, standard CCL vehicle. See section 2.3.15 for an example of using pREMUSCodec.

2.6 iMOOS2SQL

This is a transponder process, which translates Status, Contact, and Track Reports into a format for interfacing the MOOS C2 with the generic Google Earth-based (geov) topside display, e.g. as shown in Fig. 2.1. This module is available in moos-ivp-local (`http://oceanai.mit.edu/moos-ivp/pmwiki/pmwiki.php?n=Support.Milocal`).

2.7 pGeneralCodec

Deprecated. Do not use, rather use pAcommsHandler with no driver, no MAC, and no queueing if only encoding/decoding is desired.

2.8 pBTRCodec

Deprecated. Do not use, rather use the <`array_length`> feature of pAcommsHandler which provides the same functionality.

2.9 pCTDCodec

Deprecated. Do not use, rather use the <`max_delta`> feature of pAcommsHandler which provides all the same functionality but with much more generality.

2.10 pAcommsPoller

Deprecated. Use the MAC built into pAcommsHandler.

3 What's next

That's all for `goby-core` in Release 1.1. There's still a lot to do so keep tuned. If you want the bleeding edge, you can check out the Goby 2.0 branch with `bzr checkout lp:goby/2.0`.

Here's what's on the horizon:

- support for seamless inter-platform communications via acoustics (acomms), serial, wifi, and ethernet. Maybe even two cans and a string.
- a `Wt` [6] based configuration, launch, and runtime manager.

Stay tuned at `https://launchpad.net/goby`. Thanks.

Glossary

autonomy architecture loosely defined, a collection of software applications and libraries that facilitate communications, decision making, timing, and other utilties needed for making robots function. Another common term for this is autonomy “middleware”. 2

LAMSS A multidiscplinary research group at the Center for Ocean Engineering (Dept. of Mechanical Engineering) at Massachusetts Institute of Technology. LAMSS focuses on collaborative marine robotics for a variety of acoustic and non acoustic sensing tasks. See `http://lamss.mit.edu`.. 5, 6, 9

Bibliography

[1] Goby Developers, “Goby underwater autonomy project documentation.” [Online]. Available: http://gobysoft.com/doc

[2] M. R. Benjamin, H. Schmidt, P. M. Newman, and J. J. Leonard, “Nested autonomy for unmanned marine vehicles with MOOS-IvP,” *Journal of Field Robotics*, vol. 27, no. 6, pp. 834–875, 2010. [Online]. Available: http://dx.doi.org/10.1002/rob.20370

[3] T. Schneider and H. Schmidt, “Unified command and control for heterogeneous marine sensing networks,” *Journal of Field Robotics*, vol. 27, no. 6, pp. 876–889, 2010. [Online]. Available: http://dx.doi.org/10.1002/rob.20346

[4] “The laboratory for autonomous marine sensing systems (LAMSS).” [Online]. Available: http://lamss.mit.edu/

[5] T. Schneider and H. Schmidt, “The Dynamic Compact Control Language: A compact marshalling scheme for acoustic communications,” in *Proceedings of the IEEE Oceans Conference 2010*, Sydney, Australia, 2010.

[6] Emweb, “Wt, a C++ web toolkit.” [Online]. Available: http://www.webtoolkit.eu/wt