MOOS-IvP Communications Software

with acoustic networking from the Goby Underwater Autonomy Project

Toby Schneider and Henrik Schmidt

Laboratory for Autonomous Marine Sensing
Department Mechanical Engineering
Massachusetts Institute of Technology, Cambridge MA

Technical Memorandum LAMSS-09-03

August 24, 2010

Abstract

This paper provides a User's Guide for the MIT `MOOS-IvP` communication modules for undersea networking of multiple autonomous underwater vehicles (AUVs). These modules provide a complete software stack for handling communication with other vehicles and the topside operators using acoustic modems. The infrastructure described in this document allows the vehicles to remain submerged under autonomous control with commands from the topside being passed exclusively through the underwater acoustic modem network.

Contents

1 Introduction

The purpose of this document is to provide a functionality overview of and User's Guide for `MOOS-IvP` modules used for the Acoustic Communication within an underwater network using the WHOI Micro-Modems.

Following this overview, this document includes a detailed guide for MOOS system developers interested in incorporating these modules into their work.

1.1 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. *Nested Autonomy* [] is a new command and control paradigm, inherently suited for such heterogeneous networks. Implemented using `MOOS-IvP`, Nested Autonomy 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 information 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), 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. 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,

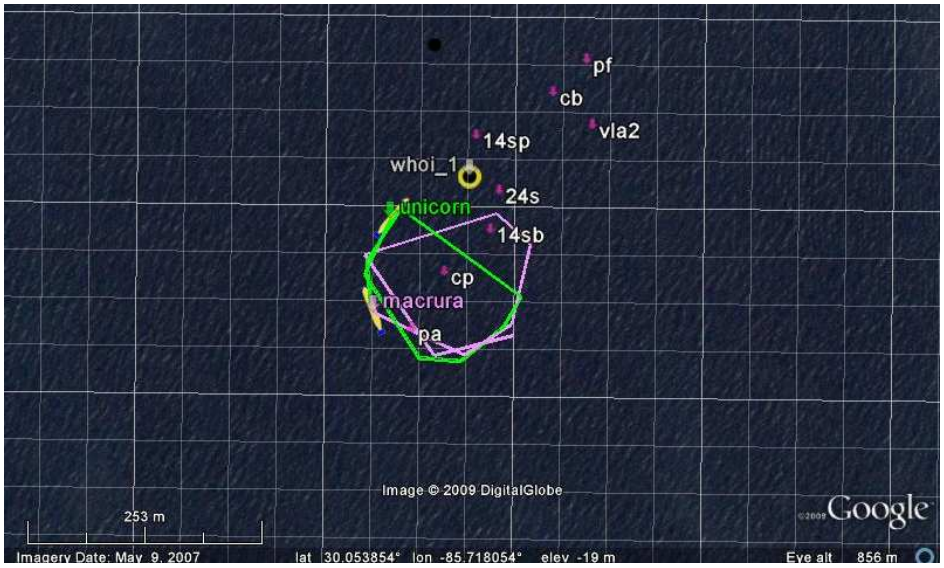


Figure 1: Collaborative autonomy demonstrated in SWAMSI09 using MIT LAMSS communication stack. The two BF21 AUVs Unicorn and Macrura perform synchronized swimming maintaining a constant bistatic angle of 60° relative to a proud cylindrical target (cp).

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 established libraries of message handling software, the open source architecture of this new MOOS communication stack 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 maximal flexibility in regard to message design, but it inherently enables arbitrary levels of encryption for LPI/LPD communication networks.

2 Overview of the LAMSS Communication Stack

MIT LAMSS 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. *Nested Autonomy*, developed over the last decade by LAMSS, is an example of such an autonomy-driven undersea sensing concept. Although

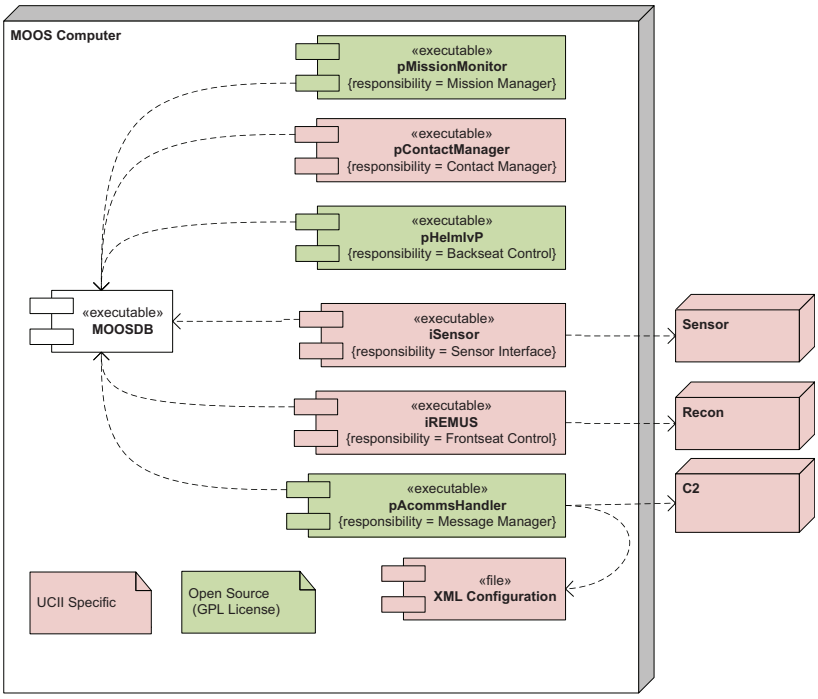


Figure 2: Incorporation of the open source LAMSS communication stack into a `MOOS-IvP` DCLT Autonomy System. The green boxes identify the open source modules, including the IvP Helm, the generic mission manager module, and the communication stack. The red modules are project specific, including the frontseat driver module, and the sensor modules. Also the message configuration specifying the message content and the coding, is project specific.

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.

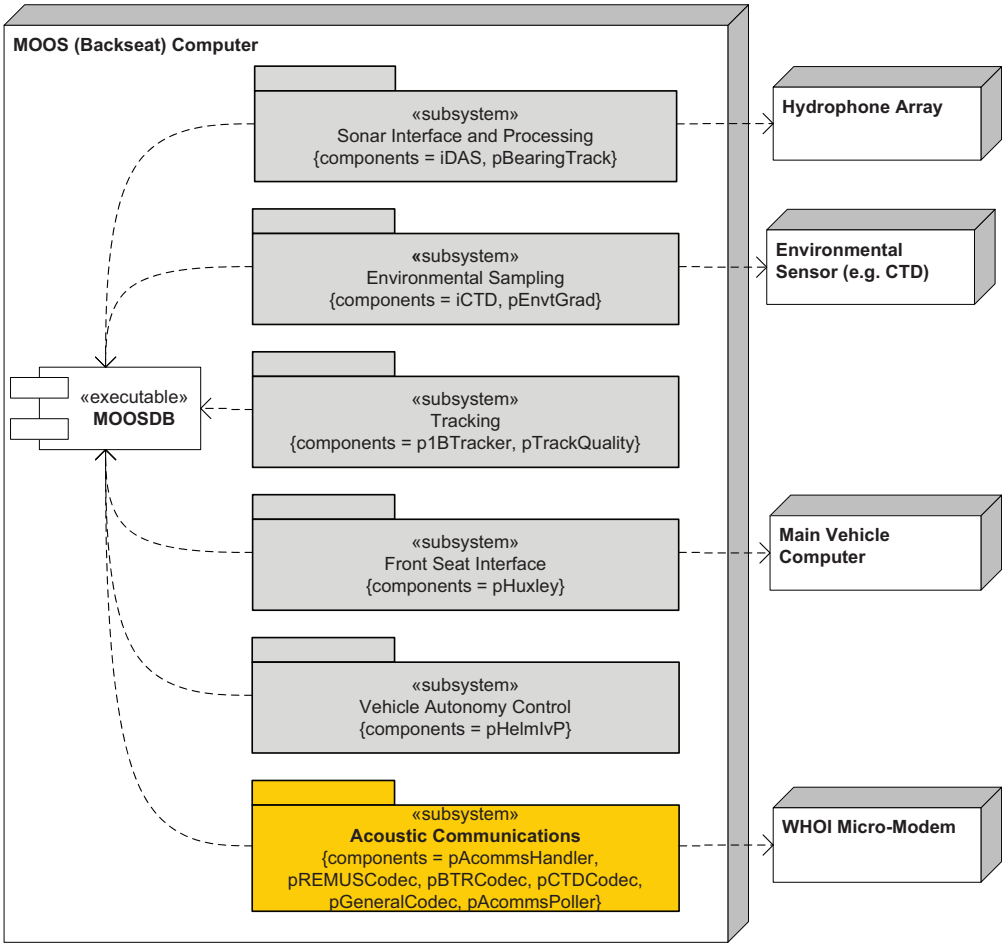


Figure 3: MOOS-IvP community for MIT sonar AUVs, with the autonomous communication, command and control modules highlighted in gold.

The incorporation of the MIT LAMSS communication stack into a `MOOS-IvP` DCLT Autonomy System is illustrated in Fig. 2. The green boxes identify the Open Source modules, including the

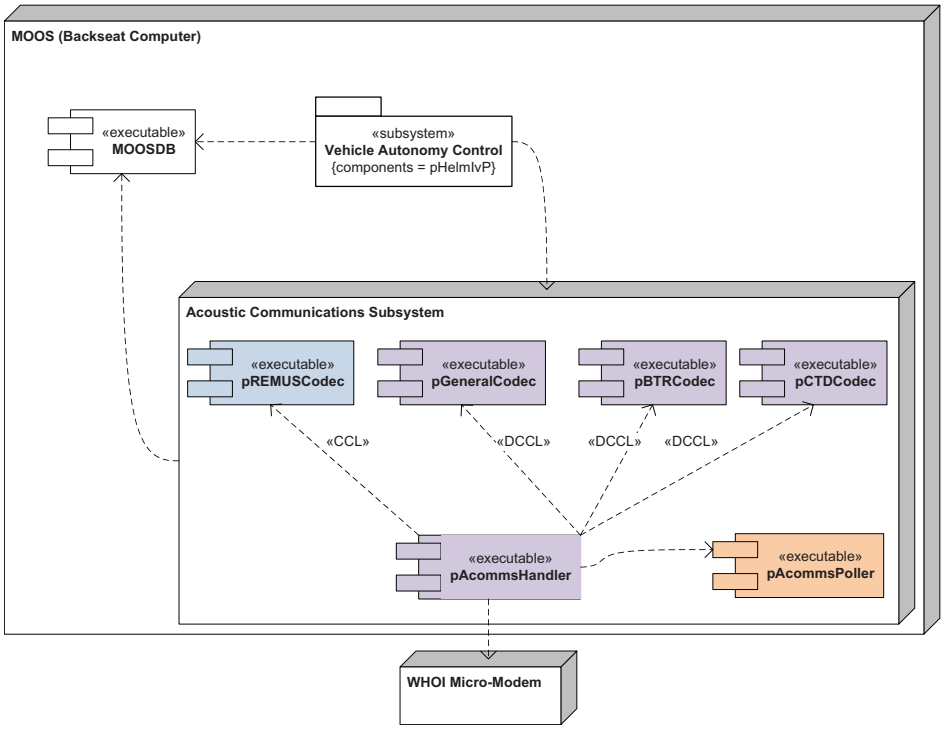


Figure 4: UML Component Model of the MIT LAMSS communication stack. The principal message handler module is `pAcommsHandler`, which communicates directly with the modem using built-in drivers, and thus not dependent on third-party MOOS modem drivers. It also manages the message stream by a dynamic, priority-based 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.

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

A list of the components of the communication stack modules is given in Table 1. A detailed description of the modules, their configuration and interaction with the `MOOSDB` is given in the appendices. Figure 5 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. 4, with the functionality of the modules being as follows:

#	Module Name	Module Description	Author	Size
1	**pAcommsHandler**	(1) Optionally handles encoding / decoding of data using DCCL. If not using this option, pGeneralCodec must be run. (2) Manages queues for transmission of acoustic messages. Different message queues can be given priorities that increase with time since the last message was sent from that queue. (3) Interfaces to the WHOI Micro-Modem firmware. (4) Handles Medium Access Control (MAC) of the acoustic network using slotted TDMA or polling.	Schneider	
2	**pREMUSCodec**	Encodes / decodes a subset of the REMUS CCL messages.	Schneider, Schmidt	
3	**iCommander**	Allows a human operator to type in the fields for a DCCL message, thus provide a way to command vehicles using DCCL.	Schneider	
4	**pCTDCodec**	Deprecated. Do not use, rather use the <max_delta> feature of pAcommsHandler which provides all the same functionality but with much more generality.	Schneider	
5	**pBTRCodec**	Deprecated. Do not use, rather use the <array_length> feature of pAcommsHandler which provides the same functionality.	Schneider	
6	**pGeneralCodec**	Handles encoding / decoding using DCCL. This can be done in pAcommsHandler now, so only run pGeneralCodec for legacy situations.	Schneider	
7	**pAcommsPoller**	Deprecated, use the MAC in pAcommsHandler.	Schneider	
Total unique lines of code Total aggregate lines of code				

Table 1: Communication software stack for MOOS-IvP onboard autonomy system.

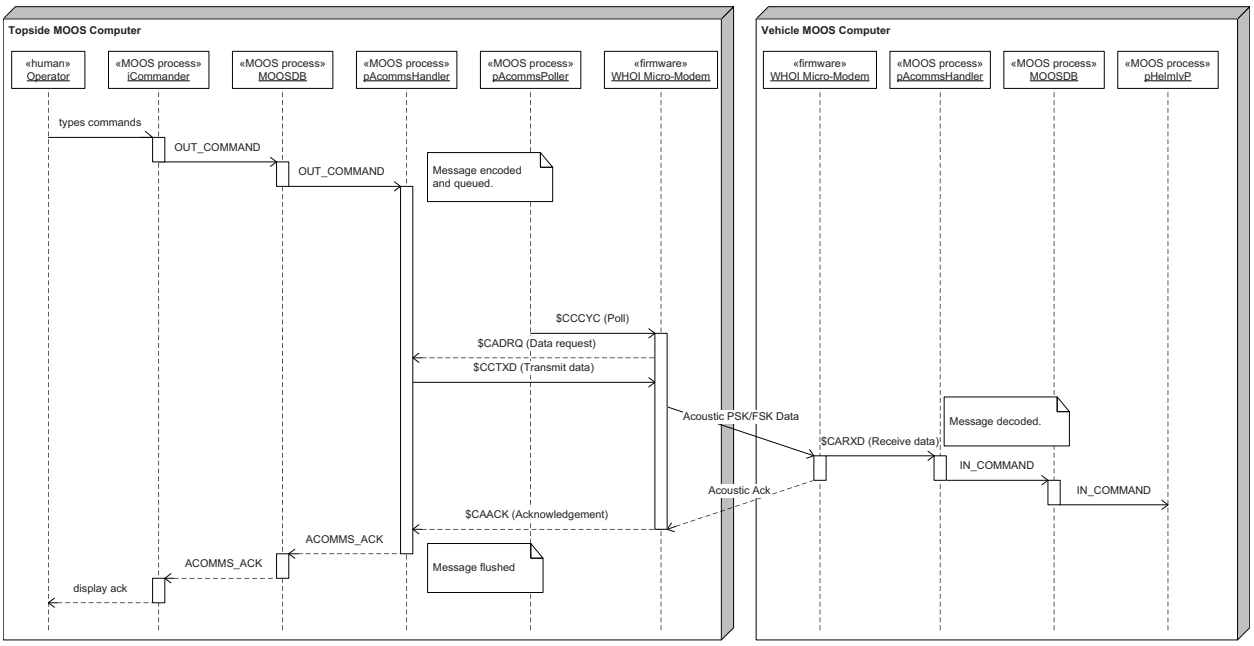


Figure 5: UML Sequence diagram for sending a command to an AUV via the LAMSS Acoustic Communications Modules.

2.1 goby-acomms

Several of the libraries used by pGeneralCodec and pAcommsHandler are now part of the Goby Underwater Autonomy Project (`https://launchpad.net/goby`). These libraries are collectively referred to as "goby-acomms". These libraries are mirrored in the moos-ivp-local subversion repository so that users of the MOOS processes presented here do not have check out `goby` code. However, the fact that these modules are all standalone from MOOS means that an interested developer can incorporate them into his or her own open source project. Please refer to the above URL if this situation applies to you.

2.2 pGeneralCodec

This process uses a custom designed compression scheme for losslessly encoding and decoding messages into arbitrary sized fields, as specified through XML configuration files. This optimizes the amount of data that will fit in a single message. In addition to the encoding/decoding specifications, the XML file for each message specifies which MOOS variable(s) should be be set upon receipt of the message, eliminating the need for dedicated interface processes between the message decoder and the autonomy system.

A number of project-specific message sets have been defined, including active sonar contact and track reports for active, multistatic acoustic applications, environmental messages for oceanographic missions, and passive sonar contact and track reports for undersea surveillance missions.

Another example is a message for acoustically poking the MOOSDB on a node. It allows the operator to modify a particular MOOS variable on a deployed node, in essence giving him full power over a deployed autonomy system. This feature has proven extremely useful for in-field debugging

of the autonomy system.

pGeneralCodec works primarily be calling the goby-acomms `dccl` library.

2.3 pAcommsHandler

pAcommsHandler is the core process in the stack. It subscribes to an arbitrary number of coded messages, specified in the configuration file, queuing them for transmission based on a set of user-configured priorities. In order to not have a low-priority message such as a Status Report becoming stale during periods of high density of other messages, such as Contact Reports the user can configure time constant for an exponential maturing of the various messages, thus establishing optimal queuing strategy for each particular CONOPS. This in contrast to `iMicroModem`, which has a hard-coded, static message priority structure, and no feature for pruning the queues, e.g. in high-clutter situations. Another feature is a message flag defining whether messages should be transmitted first-in/first out, or last-in/first-out, again a feature not available in other communication stacks.

The message scheduling can be controlled by the medium access control (MAC) built into `pAcommsHandler`. The MAC internal to `pAcommsHandler` handles either polling, slotted time division-multiple access (TDMA) with auto peer discovery, or fixed slotted TDMA.

When a node is polled for a rate 0 FSK message, only 32-byte message queues will be active. Similarly, when a node is polled for a rate 5 PSK message (8x256 byte), `pAcommsHandler` will pack the message with an optimal mix of high-priority 32-byte messages and lower priority data messages, such as those containing CTD and BTRs, based on the current priorities. This priority queuing is handled by the goby-acomms library `queue`.

`pAcommsHandler` also calls a driver that interacts directly with the WHOI Micro-Modem NMEA 0183 serial feed. To do this, it uses the goby-acomms library `modemdriver`.

Finally, `pAcommsHandler` can optionally call the same library (goby-acomms `dccl`) as `pGeneralCodec`, allowing it to be a full featured communications process without the need to run `pGeneralCodec`. `pGeneralCodec` is still provided for a user who needs a standalone encoder/decoder for DCCL messages.

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 message configuration (XML) files with `pAcommsHandler` and `pGeneralCodec` it automatically adapts to the current message set, without any need to change software code.

2.5 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. 1.

2.6 pREMUSCodec

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.

2.7 pBTRCodec

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

This is a dynamic encoder/decoder (codec) which compresses Beam-Time Records (BTRs) for passive and active sonar applications into 256 byte DCCL messages, which can be transmitted to the topside operators using high-rate Phase Shift Key (PSK) modulation on the WHOI Micro-Modem.

2.8 pCTDCodec

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

Similar to pBTRCodec, this codec compresses CTD measurements into low-rate Frequency-Shift Key (FSK), or high-rate PSK messages. pCTDCodec uses a lossless difference encoding scheme which further reduces the size of messages on highly correlated data (which CTD samples are).

2.9 pAcommsPoller

Deprecated. Use the MAC built into pAcommsHandler.

This process is typically used on the topside to schedule node communication by selective polling or broadcasting. A particularly useful feature is the possibility of interleaving polling for different rates and protocols, which in combination with the dynamic queuing by pAcommsHandler and the acknowledgement feature of the Micro-Modem allows the network to optimally adapt to the current communication capacity.

3 Communications Modules

3.1 pAcommsHandler

3.1.1 Overview

Any terms in *italics* are defined the Glossary (section A).

Problem Acoustic communications (in our case, with the WHOI Micro-Modem) 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). Reference 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

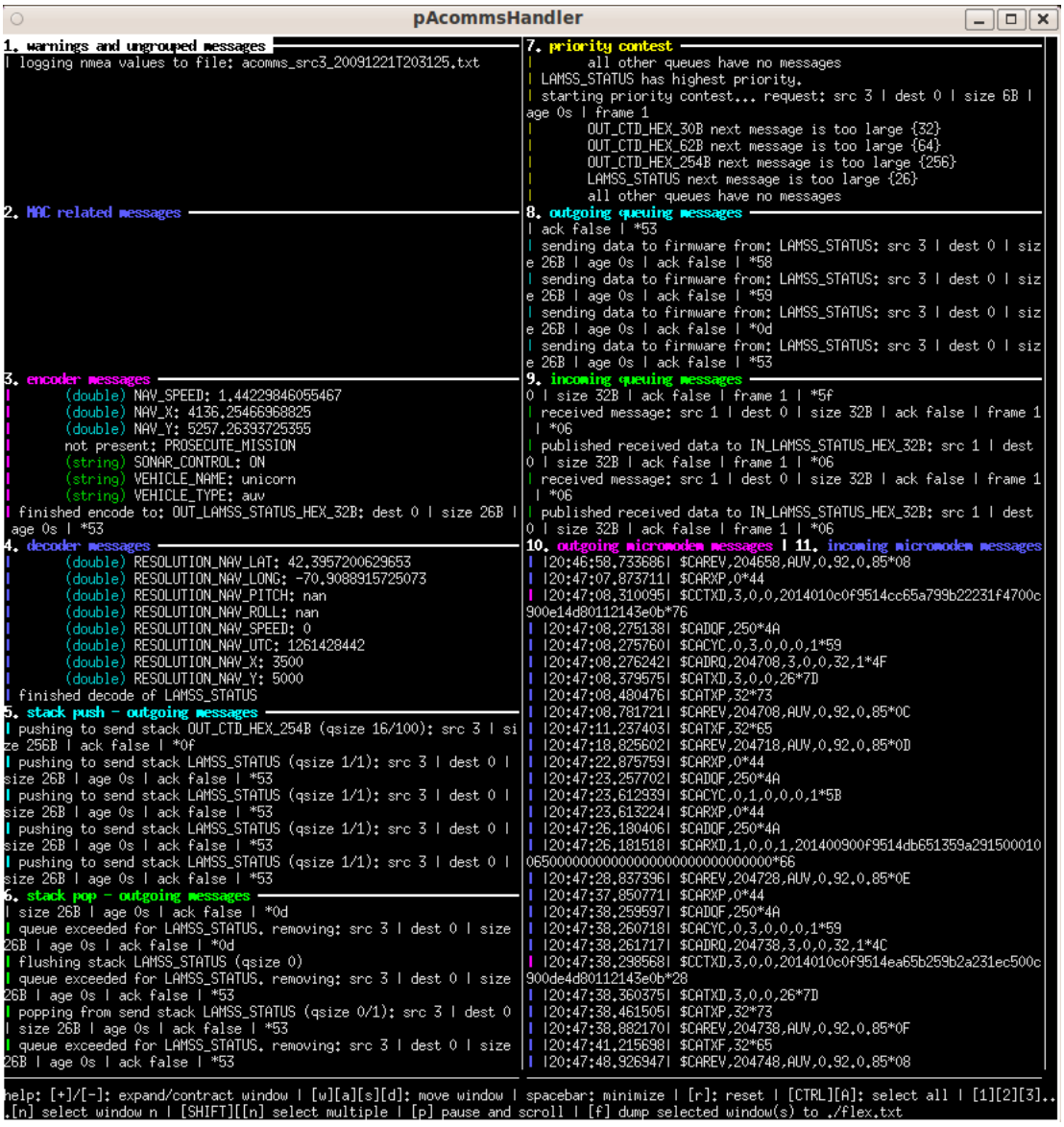


Figure 6: pAcommsHandler running with `verbosity = scope`.

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.

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)
   - 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) is intended to be extensible to other modems besides the WHOI Micro-Modem.
4. MAC Manager: provides medium access control in the form of a simple slotted time division-multiple access (TDMA) scheme or flexible centralized polling.

Limitations pAcommsHandler *does not*:

- provide any multi-hop routing. The sender and receiver must be directly in acoustic communications.
- 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).

[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

Quick Start

3.1.2 Usage

Compilation pAcommsHandler depends on the boost, boost_thread, boost_date_time, boost_regex, xerces-c, crypto++, and asio libraries in addition to the libraries included in MOOS and moos-ivp-local.

- boost: reference `http://www.boost.org/` or look for your distribution's boost developer package (libboost-dev in Debian/Ubuntu).
- boost_thread: reference `http://www.boost.org/` or look for your distribution's boost_thread developer package (libboost-thread-dev in Debian/Ubuntu).
- boost_date_time: reference `http://www.boost.org/` or look for your distribution's boost_date_time developer package (libboost-date-time-dev in Debian/Ubuntu).
- boost_regex: reference `http://www.boost.org/` or look for your distribution's boost_regex developer package (libboost-regex-dev in Debian/Ubuntu).
- asio: reference `http://sourceforge.net/projects/asio/` or look for your distribution's asio developer package (libasio-dev in Debian/Ubuntu). You do not want boost-asio, just asio. Note that you will need a fairly recent version of asio, so it may be best to simply install from source. asio is a header-only library so it is simply a matter of copying the header files to /usr/local/include or a similar directory.
- xerces-c: reference *http://xerces.apache.org/xerces-c/* or look for your distribution's xerces-c developer package (libxerces-c-dev in Debian/Ubuntu as of this writing).
- crypto++: reference *http://www.cryptopp.com* or look for your distribution's crypto++ developer package (libcrypto++-dev in Debian/Ubuntu as of this writing).
- ncurses: should be provided with most UNIX systems.

3.1.3 Parameters for the pAcommsHandler Configuration Block

Example moos file

```
modem_id = 1
modem_id_lookup_path = ../../../data/acomms/modemidlookup.txt

log_path = /tmp/moos/logs  // directory to log runtime debugging output to

ProcessConfig = pAcommsHandler
{
  // available to all moos processes.
  AppTick    = 10
  CommsTick  = 10

  verbosity = verbose   // scope, verbose [default], terse, quiet
  log = true // true [default] or false

  ////////////////////////
  // DCCL Codec configuration
  ////////////////////////

  // or you could use pGeneralCodec if you set these to "false"
  // default (for legacy support) is false for both
  encode = true // true or false [default]
  decode = true // true or false [default]
```

```
  crypto_passphrase = my_precious // if omitted, messages are not encrypted

  ////////////////////
  // XML files (used for both Queuing and DCCL encoding)
  ////////////////////

  message_file = ../../../data/acomms/acoustic_moospoke.xml

  // manipulators go between "message_file = " and "somexml.xml". comma separate multiple manipulators
  message_file = on_demand = ../../../data/acomms/simple_status.xml
  message_file = loopback,no_queue = ../../../data/acomms/simple_deploy.xml

  // relative to location of xml files! (syntax checking)
  schema = ../../../moos-ivp-local/src/tes/goby/acomms/libdccl/message_schema.xsd


  ////////////////////
  // CCL Queuing configuration (for non-DCCL CCL messages)
  ////////////////////

  // send_CCL = outgoing_hex_moos_var,
                           //     id
                           //         ack
                           //             blackout_time
                           //                 max_queue
                           //                      newest_first
                           //                          value_base
                           //                               ttl
  send_CCL    = OUT_REMUS_STATUS_HEX_30B,   0e, 0, 0, 1, 1, 0.8, 1800
  // receive_CCL = incoming_hex_moos_var, id
  receive_CCL = IN_REMUS_STATUS_HEX_30B,    0e
  receive_CCL = IN_REMUS_RANGER_HEX_30B,    10
  send_CCL    = OUT_REMUS_REDIRECT_HEX_30B,   07, 1, 0, 5, 1, 5, 1800
  receive_CCL = IN_REMUS_REDIRECT_HEX_30B,    07

  ////////////////////
  // Driver configuration
  ////////////////////
  connection_type = serial // serial [default], tcp_as_client, tcp_as_server
  serial_port = /dev/ttyS0
  baud = 19200 // 9600, 19200 [default], 38400, ... must conform to BR1 setting

  // for connection_type = tcp_as_client or tcp_as_server
  // network_port = 10000
  // for connection_type = tcp_as_client
  // network_address = 192.168.1.124

  // set all CFG values to factory defaults at the start before sending our CFG values
  cfg_to_defaults = true // true [default] or false

  // modem CFG values
  cfg = snr,1
  cfg = rev,0

  ////////////////////
  // MAC
  ////////////////////
  mac = slotted // or polled, fixed_slotted, or none [default]

  // for slotted
  slot_time = 15 // seconds for the width of each slot, default 15
  rate = 0    // modem rate (0 [default], 2, 3, 5 or auto [not implemented])
  expire_cycles = 30   // how many cycles to go without hearing from a vehicle before removing them

  // for polled
  // initializer.string = ACOMMS_POLLER_UPDATE = destination=1,update_type=add,poll_type1=data,poll_from_id1=1,poll_to_id1=0,poll_rate1=0,poll_wait1=15
  // initializer.string = ACOMMS_POLLER_UPDATE = destination=1,update_type=add,poll_type1=data,poll_from_id1=2,poll_to_id1=0,poll_rate1=0,poll_wait1=15

  ////////////////////
  // Misc
  ////////////////////

  // initialize MOOS variable to value of the .moos file variable from the top of the moos file
  global_initializer.double = LAT_ORIGIN = LatOrigin
  global_initializer.double = LONG_ORIGIN = LongOrigin
  global_initializer.string = MODEM_ID_PATH = modem_id_lookup_path

  // initialize MOOS variable from value set here
  initializer.string = VEHICLE_NAME = unicorn
  initializer.string = VEHICLE_TYPE = auv
}
```

Filling out the .moos file

General Parameters

- `verbosity`: choose `verbose` for full text terminal output, `terse` for symbolic heartbeat output, and `quiet` for no terminal output. A new `scope` mode is helpful to debugging and visualizing the many data flows of pAcommsHandler. See figure 6 for a screenshot of pAcommsHandler in scope mode.

- `modem_id`: integer that specifies the modem_id of this current vehicle / community. this must match the micromodem SRC configuration parameter (send the modem `$CCCFQ,SRC` to check). For the remainder of the document, *modem_id* refers to the value `$CCCFG,SRC,modem_id`. If using the internal driver (`driver_file = internal`), this configuration parameter will be set on startup. often this is set as a global parameter to the moos file (specifed outside the ProcessConfig blocks).

- `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, other aliases
0, broadcast, broadcast
1, endeavor, ship
3, unicorn, auv
4, macrura, auv
```

- `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]).

- `global_initializer`: 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, `global_initializer.double = LAT_ORIGIN = LatOrigin` 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.

Encoding/Decoding (DCCL) Parameters

- `encode`: boolean flag indicating if pAcommsHandler should encode the messages specified in `message_file=`. Alternatively, you can run pGeneralCodec to encode these messages. [optional, default is false for legacy support].

- `decode`: boolean flag indicating if pAcommsHandler should decode the messages specified in `message_file=`. [optional, default is false for legacy support].

- `crypto_passphrase`: secret text string used to encrypt DCCL messages with. All parties to the communication must have the same passphrase and this must be kept secret for secure communications. [optional, messages will not be encrypted if this is not provided.].
- `message_file`: path to an XML file containing a message set of one or messages. You can insert manipulators in between the `message_file=` and `somefile.xml` that change the behavior of pAcommsHandler for messages defined in that file. Comma separate multiple manipulators. Allowed manipulators:
  - `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 (used to command yourself)
  - `on_demand`: encode immediately upon a data request command (use for time sensitive messages like STATUS)
- `schema`: path to the DCCL `message_schema.xsd` XML schema used for syntax checking. If using a relative path, specify the path relative to the XML file location, *not* the present working directory (pwd).

DCCL Queuing information stored in message XML files All queue configuration for DCCL messges must be configured within the XML files `<queuing />` tag and included with `message_file = message.xml`. The tags correspond to these configuration variables:

- `outgoing_hex_moos_var`: name of the moos variable to subscribe to for messages to add to this queue. Publishes here should be pure hexadecimal or a key=value string specified later in section 3.1.10.
- `id`: a user specified integer from 0-127 that is a tag for the `outgoing_hex_moos_var`. Thus, each `outgoing_hex_moos_var` must have a unique `id`. Only the `id` is sent with the message (to save space), not the full `outgoing_hex_moos_var` of the queue. Thus, this must match the `id` on the receiving vehicle's `receive` configuration in the incoming parameters section below. For example, if i have `send=SOME_OUT_HEX, 1` on the sending vehicle, the receiving vehicles must have a field `receive=SOME_IN_HEX, 1`. All messages with ID 1 will be put in `SOME_IN_HEX`. Clearly, if unique mapping is desired on the receiving end, unique `id`s must be used on the sending end.
- `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.

An example queuing block:

```
<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>
```

CCL Queue Parameters

- `send_CCL`: configure a queue for a WHOI CCL message type (do not use the moos `id`). The queues mentioned above (`send=`) use the CCL identifier byte 0x20 and then the second byte includes the `VariableID` leaving $N-2$ bytes for the user where $N = 32, 64,$ or 256 depending on the modem rate. This queue `send_CCL` does not use the second byte, thus making it useful for handling the static CCL types defined by WHOI. The send_CCL parameter is a comma delimited string of two mandatory parameters and up to six optional parameters (in order specified here). To skip an optional field and use the default value, simply use a blank ",":

All parameters in the `send_CCL` list are the same as those for the DCCL tag list with the exception of the second parameter. Here it is the `CCLIdentifierByte` in hexadecimal, not the `id`.

- `receive_CCL`: specifies a mapping between an incoming `incoming_hex_moos_var` and a moos variable to place the incoming message. The sender's modem id is stored as the community name for the received message. Rather than a `incoming_hex_moos_var` you specify a `CCLIdentifierByte` that is the first byte of the CCL message.

Micro-Modem Driver Parameters

- `connection_type`: type of connection to make to the modem (serial, tcp_as_client, tcp_as_server) [optional, default: serial]
- `serial_port`: serial port to which the MicroModem is connected [mandatory if `connection_type=serial`].
- `baud`: baud rate to use. [optional, default: 19200]
- `network_port`: networking port to use. [mandatory if `connection_type=tcp_*`]
- `network_port`: IPv4 networking address of the server to connect to. [mandatory if `connection_type=tcp_as_client`]
- `cfg`: set some modem NVRAM setting to a value. note that by default pAcommsHandler resets all NVRAM (CFG) parameters on startup (`$CCCFG,ALL,0`) and then sets the these values. thus, you must include `cfg=` lines only for parameters that differ from the defaults. Set `cfg_to_defaults=false` to turn off the automatic reset of the CFG parameters.

Medium Access Control (MAC) Parameters

- `mac`: type of Medium Access Control. Values can be `slotted`, `polled`, `fixed_slotted` or `none`.
- `slot_time`: length, in seconds, of each communication slot for the `mac=slotted` MAC option.
- `rate`: rate for the `mac=slotted` MAC option. 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)
- `initializer.string`: use the MOOS variable `ACOMMS_MAC_CYCLE_UPDATE` to adjust the TDMA schedule for the `mac=polled` or `mac=fixed_slotted` MAC option. To set a few initial values, you can use the `initializer` key to "poke" this MOOS variable on startup. The format of `ACOMMS_MAC_CYCLE_UPDATE` is a string of comma delimited `key=value` pairs, where # is an incrementing number for each slot specified in the message. See table 2 for all the keys and values.

  for example:

```
destination=1,update_type=add,slot_1_type=data,slot_1_from=1,slot_1_to=0,
slot_1_rate=0,slot_1_wait=15
```

key	value
destination	modem_id of poller to update
update_type	enum{add,replace,remove}
slot_#_type	enum{data,ping}
slot_#_from	modem_id of the sender
slot_#_to	modem_id of the receiver
slot_#_rate	rate to poll at
slot_#_wait	seconds for the slot to last before next poll

Table 2: Parameters of `mac=polled` and `mac=fixed_slotted` reconfiguration message (to `ACOMMS_MAC_CYCLE_UPDATE`)

3.1.4 MOOS variables subscribed to by pAcommsHandler

Some variables are configurable in the .moos file and/or message XML files as described in section 3.1.3. For example, if `send = OUT_CTD_HEX_30B, ...` is set in the .moos file (or `<outgoing_hex_moos_var>OUT_CTD_HEX_30B</outgoing_hex_moos_var>` in a message XML file), then `OUT_CTD_HEX_30B` is the variable actually subscribed. See section 3.1.6 and beyond for details on filling out and interpreting these XML files. See table 3 for a full list of the subscribed variables.

3.1.5 MOOS variables published by pAcommsHandler

Similarly to the subscriptions, the some publishes done by pAcommsHandler are defined in the .moos file or in message XML files. Again, instead of MOOS variables, the table below sometimes indicates the .moos file keys or XML tags for which one can define the publishes. See table 4 for a full list of the published variables.

3.1.6 DCCL Encoding/Decoding Unit: Overview

Example message XML file First, let us give a brief background on XML (e**X**tensible **M**arkup **L**anguage). 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 explicitly 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 3.1.7 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>
    <outgoing_hex_moos_var>OUT_GOTO_HEX</outgoing_hex_moos_var>
    <incoming_hex_moos_var>IN_GOTO_HEX</incoming_hex_moos_var>
    <trigger>publish</trigger>
    <trigger_moos_var mandatory_content="CommandType=GoTo">OUTGOING_COMMAND</trigger_moos_var>
    <size>32</size>
```

MOOS variable or .moos file key or XML tag specifying a MOOS variable	Type	Description	Published by
DCCL			
<destination_moos_var key="destkey"/>	$ or D	Contains the modem_id to send this message from. Can either be double (ex: 3) or string (ex: ...,destkey=3,...)	many
<trigger_moos_var/>	$ or D	A publish here triggers the creation of this message. The contents may contain message parts or not.	many
<moos_var key="somekey"/>	$ or D	Data for a given *message_var*. Can either be double (ex: 3.234) or string (ex: "bob" or "3.234") or keyed string (ex: ...,somekey=3.234,...).	many
Queue			
send= outgoing_hex_moos_var,...	$	outgoing_hex_moos_var contains hexadecimal string to queue (and eventually send).	Various Codecs (pCTDCodec, pBTRCodec, etc.)
<outgoing_hex_moos_var/>	$	outgoing_hex_moos_var contains hexadecimal string to queue (and eventually send). **Only subscribed for when DCCL encoding is disabled in pAcommsHandler.**	pGeneralCodec
Driver			
ACOMMS_NMEA_OUT	$	Raw messages to send to the modem	
MAC			
ACOMMS_MAC_CYCLE_UPDATE	$	Updates to the mac=polled or mac=fixed_slotted TDMA cycle	

Table 3: MOOS Variables Subscribed to by pAcommsHandler

MOOS variable or .moos file key or XML tag specifying a MOOS variable	Type	Description	Format
DCCL			
<publish> <moos_var type="string"/> </publish>	$	(string) Message created from decoded hexadecimal string.	Defined by <format>
<publish> <moos_var type="double"/> </publish>	D	(double) Message created from decoded hexadecimal string.	Defined by <format>
<outgoing_hex_moos_var/>	$	Encoded hexadecimal string.	
Queue			
receive= incoming_hex_moos_var,...	$	incoming_hex_moos_var contains a hexadecimal string from the micromodem to route to this queue.	see below
<incoming_hex_moos_var/>	$	incoming_hex_moos_var contains a hexadecimal string from the micromodem to route to this queue.	see below
ACOMMS_ACK	$	Notification of acknowledged messages	
ACOMMS_EXPIRE	$	Notification of expired messages (ttl exceeded)	
ACOMMS_SQSIZE_name	$	Notification of change in queue size (push or pop)	New size of queue (number of messages)
Driver			
ACOMMS_NMEA_IN	$	Incoming NMEA messages from the modem	See WHOI Micro-Modem Software Interface Guide
ACOMMS_NMEA_OUT	$	Outgoing NMEA messages to the modem	See WHOI Micro-Modem Software Interface Guide
MAC			

Table 4: MOOS Variables Published by pAcommsHandler

```
  <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>
      <moos_var>INCOMING_COMMAND</moos_var>
      <all />
    </publish>
    <publish>
      <moos_var>SPECIAL_INSTRUCTIONS</moos_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>
    <outgoing_hex_moos_var>OUT_STATUS_HEX</outgoing_hex_moos_var>
    <incoming_hex_moos_var>IN_STATUS_HEX</incoming_hex_moos_var>
    <size>32</size>
    <layout>
      <float>
        <name>nav_x</name>
        <moos_var>NAV_X</moos_var>
        <max>1000</max>
        <min>0</min>
        <precision>1</precision>
      </float>
      <float>
        <name>nav_y</name>
        <moos_var>NAV_Y</moos_var>
        <max>1000</max>
        <min>0</min>
        <precision>1</precision>
      </float>
      <enum>
        <name>health</name>
        <moos_var>VEHICLE_HEALTH</moos_var>
        <value>good</value>
        <value>low_battery</value>
        <value>abort</value>
      </enum>
    </layout>
    <on_receipt>
      <publish>
        <moos_var>STATUS_SUMMARY</moos_var>
        <all />
      </publish>
    </on_receipt>
  </message>
</message_set>
```

3.1.7 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 3.1.6 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 3.1.6. We do not need any <`moos_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 <`moos_var`>, pGeneralCodec will insert <`moos_var`>`OUTGOING_COMMAND`</`moos_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>
     <moos_var>OUTGOING_COMMAND</moos_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.
   - <`outgoing_hex_moos_var`>`OUT_GOTO_HEX`</`outgoing_hex_moos_var`>: we will publish our hex strings here for consumption by the low level modem stack processes (probably pAcommsHandler `send=OUT_GOTO_HEX...`). the publishes will look a bit like:

     ```
     OUT_GOTO_HEX: Dest=3,HexData=a9385bc098109830a9385bc0981098a9385bc098109830a9385bc0981098
     ```

   - <`incoming_hex_moos_var`>`IN_GOTO_HEX`</`incoming_hex_moos_var`>: where we expect to receive incoming messages to decode (probably from pAcommsHandler `receive=IN_GOTO_HEX...`). messages should be pure hex with the community set to the sending *modem_id*. An example for the received message on *modem_id*=3 if the sending node (say a topside command computer) is *modem_id*=1:

```
IN_GOTO_HEX: a9385bc098109830a9385bc0981098a9385bc098109830a9385bc0981098
{Community=1}
```

- <trigger>publish</trigger>: we are creating this message on a **publish** (to OUTGOING_COMMAND).
- <trigger_moos_var mandatory_content="CommandType=GoTo">OUTGOING_COMMAND</trigger_moos_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></message> is only for a GoTo message). Other messages can be created to encode/decode these other command types.
- <size>30</size>: we want this message to fit in a WHOI micromodem FSK frame (32 bytes) and thus we have 30 bytes to work with (pAcommsHandler needs 2 bytes of header).

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) pGeneralCodec 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>
    <moos_var>INCOMING_COMMAND</moos_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>
    <moos_var>INCOMING_COMMAND</moos_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>
  <moos_var>SPECIAL_INSTRUCTIONS</moos_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_moos_var mandatory_content="CommandType=GoTo">OUTGOING_COMMAND</trigger_moos_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 <moos_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 moos db). 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 <moos_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, the pGeneralCodec looks for a substring of the form:

`name=value`

within the string and picks out the 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

  `(double)SOME_FLOAT_VARIABLE: 3.56`

  then 3.56 is sent.
- if instead

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

  then 3.56 is still sent.
- if instead

  `(string)SOME_FLOAT_VARIABLE: "3.56"`

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

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

  then `blah=3.56` is converted (using streams) to a float, which will probably be zero or something else undesired. In other words, this case is not what you want, whereas the above three are fine.

Further examples

- I currently store our working message files in `moos-ivp-local/data/acomms`. 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 `MY_STRING` that gets truncated at 30 chars (need two bytes for CCL and DCCL ids) and sent to broadcast):

```
<?XML version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <name>SimpleStringSender</name>
    <id>1</id>
    <trigger>publish</trigger>
    <trigger_moos_var>MY_STRING</trigger_moos_var>
    <size>32</size>
    <layout>
      <string>
        <name>my_string</name>
        <max_length>30</max_length>
      </string>
    <on_receipt>
      <publish>
```

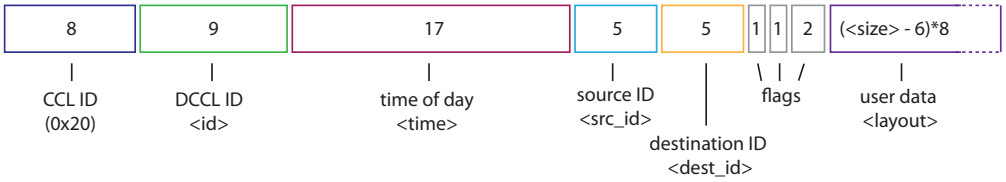


Figure 7: Layout of the DCCL header, showing the fixed size (in bits) of each header field. The user cannot modify the size of these header fields, but can access and set the data inside through the same methods used for the customizable data fields specified in <layout>. The flags are not used by DCCL, but are included for use by the lower level networking.

```
      <moos_var>INCOMING_COMMAND</moos_var>
      <all />
    </publish>
  <on_receipt>
 </message>
</message_set>
```

3.1.8 DCCL Encoding/Decoding Unit: XML Tag Reference

Message XML file reference: allowed tags Let us now give a description of all the allowed tags:

- <?xml version="1.0" encoding="UTF-8"?>: specifies the file is XML. All you need to know is that this must be the first line of every message XML file.
- <message_set>: the root element. All XML files must have a single root element. Since we are define a set of messages (one or more per file), this is a logical choice of name for the root element. [mandatory, one allowed].
- <message>: defines the start of a message. [mandatory, one or more allowed].
  - <name>: a human readable name for the message. This is not used internally at this point in time. [mandatory, one allowed]
  - <size>: the maximum allowed size of the message in bytes. There are eight bits (binary digits) to a byte. Use N here for messages passed to the micromodem where N is the desired micromodem frame size (N =32, 64, or 256 depending on the rate). If the <layout> of the message exceeds this size, pAcommsHandler will exit on startup with information about sizes, from which you can remove or reduce the size of certain *message_var*s.
  - <repeat>: make as many copies of the message structure defined in <layout> as will fit in the message <size>. No message will be sent until the message is full. For example, if the message is 32 bytes and the layout is 8 bytes, three copies of the message will be stored before sending ($32 - 6 - 3 * 8 = 0$). That is, three messages will be triggered, packed and sent as a single DCCL message. [optional, if omitted only a single copy is made]. *If* <repeat> *is specified, array_length must omitted for all message_vars.* That is, you cannot have repeated messages that contain arrays.

- <header>: the children of this tag allow the user to rename the header parts of the DCCL message. See Fig. 7 for a sketch of the DCCL header format. These names are used when passing values at encode time for the various header fields.
  * <id>: an unsigned nine bit integer (1-511) that identifies this message within a network. very similar to the CCL identifier, but for DCCL messages. The CCL identifier occupies the most significant byte (MSB) of the message followed by this id which takes the second MSB. *This must be unique within a network as this id determines the message decoding* [mandatory, one allowed]
  * <time>: seconds elapsed since 1/1/1970 ("UNIX time"). In the DCCL encoding, this reduced to seconds since the start of the day, with precision of one second. Upon decoding, assuming the message arrives within twelve hours of its creation, it is properly restored to a full UNIX time.
    · <name>: the name of this field; optional, the default is "_time". [**string**]
  * <src_id>: a unique address (<src_id> $\in [0, 31]$) of the sender of this message. For a given experiment these short unique identifiers can be mapped on to more global keys (such as vehicle name, type, ethernet MAC address, etc.).
    · <name>: default is "_src_id". [**string**]
  * <dest_id>: the eventual destination of this message (also an **unsigned integer** in the range [0,31]). If this destination exists on the same subnet as the sender, this will also be the hardware layer destination id number.
    · <name>: default is "_dest_id". [**string**]
- <layout>: defines the message structure itself (what fields [the message variables or *message_var*s] the message contains and how they are to be encoded). [mandatory, one allowed].
  * <static>: a *message_var* that is not actually sent with the message but can be used to include in received messages (*publishes*). [optional, one or more allowed].
    · <name>: the name of this *message_var*. [mandatory, one allowed].
    · <value>: the value of this static variable. [mandatory, one allowed].
  * <bool algorithm="">: a boolean (true or false) *message_var* The optional parameter **algorithm** allows you to perform certain algorithms on the data before encoding. See below. [optional, one or more allowed].
    · <name>
    · <moos_var>: the moos variable from which to pull the value of this field. [optional if <trigger>publish</trigger>: default is trigger_moos_var; mandatory if <trigger>time</trigger>, one allowed].
    · <array_length>: if larger than 1, this makes a bool array instead of a single bool. [optional, default is 1].
  * <int algorithm="">: an integer *message_var* [optional, one or more allowed].
    · <name>
    · <moos_var>
    · <max>: the maximum value this field can take. [mandatory, one allowed].
    · <min>: the minimum value this field can take. [mandatory, one allowed].

- <array_length>
- <max_delta>: if specified, delta-difference encoding is done of the <repeat>ed message or the values in the array (for <array_length> > 1). The first value is used as a key for the remaining values which are sent as a difference to this key. The number specified here is the maximum expected difference between the first value (key) and any of the remaining values in the message. [optional, if omitted, delta-difference encoding is not performed].

* <float algorithm="">: a floating point *message_var* [optional, one or more allowed].
  - <name>
  - <moos_var>
  - <max>
  - <min>
  - <precision>: an integer that specifies the number of decimal digits to preserve. Negatives are allowed. For example, <precision>2</precision> rounds 1042.1234 to 1042.12; <precision>-1</precision> rounds 1042.1234 to 1.04e3. [mandatory, one allowed].
  - <array_length>
  - <max_delta>
* <string algorithm="">: an ASCII string *message_var* [optional, one or more allowed].
  - <name>
  - <moos_var>
  - <max_length>: the length of the string value in this field. Longer strings are truncated. <max_length>4</max_length> means "ABCDEFG" is sent as "ABCD". [mandatory, one allowed].
  - <array_length>
* <enum algorithm="">: an enumeration *message_var* [optional, one or more allowed].
  - <name>
  - <moos_var>
  - <value>: a possible value (string) the enum can take. Any number of values can be specified. [mandatory, one or more allowed].
  - <array_length>
* <hex>: a message variable represented pre-encoded hexadecimal to add to the message. This field is useful if another source is encoding part or all of a DCCL message. [optional, one or more allowed].
  - <name>: the name of this *message_var*. [mandatory, one allowed].
  - <moos_var>
  - <num_bytes>: the number of bytes for this field. The string provided should be twice as many characters as <num_bytes> since each character of a hexadecimal string is one nibble (4 bits or 1/2 byte). [mandatory, one allowed].
  - <array_length>

- <on_receipt>: begins a set of actions to be performed when a message of this type is received from another vehicle. [mandatory, one allowed].
  - <publish>: defines a single output value upon receipt of a message. Any number of publishes containing any subset of the *message_var*s can be specified. [mandatory, one or more allowed].
    - <moos_var>: the name of the moos variable to publish to. If desired, a format string is allowed here as well (e.g. %1%_NAV_X will fill %1% with the first *message_var*). See the <format> tag description for more info. [mandatory, one allowed].
    - <format>: a string conforming to the format string syntax of the boost::format [2] library. This field will specify the format of the string published to the moos variable defined in <moos_var>. At its simplest it is a string of incrementing numbers surrounded by %%. Or, instead, you may also use a printf style string, using %d for int *message_var*, %lf for floats, and %s for strings, bools and enums. [optional: default is name1=%1%,name2=%2%,name3=%3%, where name1 is the name of the first <message_var> field to follow, name2 is the second, etc. exception: default is %1% if only a single <message_var> defined. one allowed].
    - <message_var algorithm="">: the name (<name> above) of a *message_var* contained in this message (i.e. an <int>, <bool>, etc.) the values of these fields upon receipt of a message will be used to populate the format string and the result will be published to <moos_var>. The optional parameter algorithm allows you to perform certain algorithms on the data after receipt before publishing. See below. [mandatory unless <all> used, one or more allowed].
    - <all>: equivalent to <message_var> for all the *message_var*s in the message. This is a shortcut when you want to publish all the data in a human readable string. [optional, one allowed].
- <queuing>: optional section used by pAcommsHandler to handle message queuing. See section 3.1.3 for details.
- <trigger>: how the message is created. Currently this field must take the value "publish" (meaning a message is created on a publish event to a certain moos variable) or "time" (a message is created on a certain time interval). [mandatory, one allowed]
- <trigger_var>: used if <trigger>publish</trigger>, this field gives the MOOS variable that publishes to will trigger the creation of this message [mandatory if and only if <trigger>publish</trigger>]. optional attribute mandatory_content specifies a string that must be a substring of the contents of the trigger variable in order to trigger the creation of a message. For example, if you wanted to create a certain message every time COMMAND contained the string CommandType=GoTo... but no other time, you would specify
  mandatory_content="CommandType=GoTo" within this tag.
- <trigger_time>: used if <trigger>time</trigger>, this field gives the time interval pGeneralCodec should create this message. For example, a value of

[2] see the syntax of the **format-string** at *http://www.boost.org/doc/libs/1_37_0/libs/format/doc/format.html#syntax*

<trigger_time>10</trigger_time> would mean a message was created every ten seconds. [mandatory if and only if <trigger>time</trigger>].

- <destination_var>: moos variable to find where this message should be sent. Specify attribute "key=" to specify a substring to look for within the value of this moos variable. For example, if COMMAND contained the string Destination=3 and you want this message sent to modem_id 3, then you should set key=Destination to properly parse that string. [optional: default is 0 (broadcast), one allowed].
- <incoming_hex_moos_var>: where to look for messages (hex string) to decode. [optional, one allowed. default is IN_<name/>_HEX_<size/>B].
- <outgoing_hex_moos_var>: where to publish the encoded message (as a hexadecimal string). [optional, one allowed. default is OUT_<name/>_HEX_<size/>B].

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[3]. 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:

[3]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, other aliases
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>`.

3.1.9 DCCL Encoding/Decoding Unit: Under the Hood

Bitwise Layout of the Messages We may want to know the actual layout of the binary/hex message. Let us explain it with an example; for the first example message in `example1.xml` given in section 3.1.6, if we run pGeneralCodec we get information about that message:

```
type (static):
        value: {goto}
        size [bits]: [0]
goto_x (int):
        source: {OUTGOING_COMMAND}
        [min, max] = [0,10000]
        size [bits]: [14]
goto_y (int):
        source: {OUTGOING_COMMAND}
        [min, max] = [0,10000]
        size [bits]: [14]
lights_on (bool):
        source: {OUTGOING_COMMAND}
        size [bits]: [1]
new_instructions (string):
        source: {SPECIAL_INSTRUCTIONS}
        max_length: {10}
        size [bits]: [80]
```

Table 5: Formulas for encoding the DCCL types.

DCCL Type	Size (bits)	Encode[a]
<bool>	2	$x_{enc} = \begin{cases} 2 & \text{if } x \text{ is true} \\ 1 & \text{if } x \text{ is false} \\ 0 & \text{if } x \text{ is undefined} \end{cases}$
<enum>	$\lceil \log_2(1 + \sum \epsilon_i) \rceil$	$x_{enc} = \begin{cases} i+1 & \text{if } x \in \{\epsilon_i\} \\ 0 & \text{otherwise} \end{cases}$
<string>	$length \cdot 8$	ASCII[b]
<int>	$\lceil \log_2(max - min + 2) \rceil$	$x_{enc} = \begin{cases} \text{nint}(x - min) + 1 & \text{if } x \in [min, max] \\ 0 & \text{otherwise} \end{cases}$
<float>	$\lceil \log_2((max - min) \cdot 10^{precision} + 2) \rceil$	$x_{enc} = \begin{cases} \text{nint}((x - min) \cdot 10^{precision}) + 1 & \text{if } x \in [min, max] \\ 0 & \text{otherwise} \end{cases}$
<hex>	$num\_bytes \cdot 8$	$x_{enc} = x$

· x is the original (and decoded) value; x_{enc} is the encoded value.

· min, max, $length$, $precision$, num_bytes are the contents of the <min>, <max>, <max_length>, <precision>, and <num_bytes> tags, respectively. ϵ_i is the ith <value> child of the <enum> tag (where $i = 0, 1, 2, \ldots$).

· nint(x) means round x to the nearest integer.

[a] for all types except <string> and <hex>, if data are not provided or they are out of range (e.g. $x > max$), they are encoded as zero ($x_{enc} = 0$) and decoded as not-a-number (NaN).

[b] the end of the string is padded with zeros to $length$ before encoding if necessary.

```
goto_speed (float):
        source: {OUTGOING_COMMAND}
        [min, max] = [0,3]
        precision: {2}
        size [bits]: [9]
```

the calculated sizes are used to pack the message like so (# equals size of field in bits), where left to right is the same as reading the hex string from left to right:

```
[[0 {122}][goto_x {14}][goto_y {14}][lights_on {1}][new_instructions {80}][goto_speed {9}]]
```

where [0 122] means zero fill the front of the message to the full size (30 bytes = 240 bits minus 118 for other fields = 122). Byte boundaries are dissolved and encoded as a string "ABCDEF..." where the most significant byte (MSB, or leftmost 8 bits) is 0xAB, second MSB is 0xCD, etc.

The encoding of each *message_var* is done as an unsigned integer, with the exception of strings, which are store as ASCII. The value 0 (all bits zero) always indicates "not specified" or "Not a Number" (nan). This means that the user did not specify any value for this field, specified a value causing overflow (<int> or <float> greater than <max> or less than <min>), or provided a value for an <enum> that did not match any of the enumerate's <value> options. Along with this rule, the method for encoding and decoding is given in Table 5. An example is provided in Fig. 8.

Figure 8: Example of the DCCL encoding process. The process of encoding starts with the DCCL XML file (a). Data are provided by the application (b). *libdccl* encodes these data to binary via the algorithms given in Table 5 to form the header (c) and layout (d), concatenates and zero fills the encoded layout from most significant bit to closest byte (e) to produce the full encoded message (f). Finally, this point the message is encrypted (if desired).

3.1.10 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 iMicroModem, it has a priority "contest" between the queues. the queue with the current highest priority (as determined by the `priority` and `priority_time_const` 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)).

For messages with `ack=1` (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). Perhaps i will add max retries at some point soon. Messages with `ack=0` 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` $= \infty$.

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 modem, which always reports data, regardless of the sender's ID.

Input (to pAcommsHandler) formats pAcommsHandler accepts several formats for the strings placed in the various `outgoing_hex_moos_var` moos variables (outgoing message queues). The simplest is just a hexadecimal string of the appropriate length ($N - 2$ bytes where $N = 32, 64,$ or 256 depending on the modem rate). This message will be queued to send to broadcast (modem id 0[4]). An example:

```
OUT_CTD_HEX_32B: 20086850a232ccd2be62e1f36e6b02fffa92385bce8fa2109efa8902ddf3a02
```

would queue a CTD message to be sent to broadcast (modem ID 0), given the above configuration file.

pAcommsHandler also accepts a string of key=value, comma delimited fields allowing for more flexibility. Currently, the only fields supported are `data=` (required, of course) and `dest=` for the intended destination modem_id:

```
OUT_CTD_HEX_32B: dest=3,data=20086850a232ccd2be62e1f36e6b02fffa92385bce8fa2109efa8902ddf3a02
```

would queue a CTD message to be sent to id 3.

[4]the WHOI micromodem does not treat certain modem IDs specially (such as zero). All data are reported to the control computer, regardless of whether that machine is the intended recipient. However, the communications structure defined by pAcommsHandler and other *tes* processes treat modem ID 0 as broadcast (much like xxx.xxx.xxx.255 is used for broadcast on TCP/IP networks). This means that incoming messages are read if they are to our modem ID or to modem ID 0.

Output (from pAcommsHandler) formats Upon receipt of a message from the Micro-Modem, the first byte of the message is checked against the moos CCL type (0x20) and any of the additional `receive_CCL CCLIdentifierByte` given in the .moos file. if none of these match, the message is disregarded. If the message matches the moos CCL type (0x20), the second byte is examined for the `id`. If the `id` matches one of the `receive` fields in the .moos file, the hexadecimal string is stripped of its first two bytes and placed in the correspoding `incoming_hex_moos_var` moos variable. The community name is set to the sender's variable ID[5]. For example, the modem reports:[6]

```
$CARXD,3,0,0,1,20016850a232ccd2be62e1f36e6b02fffa92385bce8fa2109efa8902ddf3a02
```

pAcommsHandler places the message in the appropriate moos variable, which for the example .moos file here is `IN_CTD_HEX_32B`:

```
IN_CTD_HEX_32B: 20016850a232ccd2be62e1f36e6b02fffa92385bce8fa2109efa8902ddf3a02
{Community=3}
```

from here either pGeneralCodec or pAcommsHandler's DCCL unit will decode the message (or pREMUSCodec for CCL messages).

3.1.11 Modem Driver Unit

The Modem driver unit current supports the WHOI Micro-Modem acoustic modem. It is tested to work with revision 0.93.0.30 of the Micro-Modem firmware, but is known to work with older firmware (at least 0.92.0.85). The following commands of the WHOI Micro-Modem are implemented:

Modem to Control Computer ($CA):

- $CAACK - acknowledgement of sent message.
- $CADRQ - data request.
- $CARXD - received hexadecimal data.
- $CAREV - revision number and heartbeat. Used to check for correct clock time and modem reboots.
- $CAERR - error message.

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

- $CCTXD - transmit data.
- $CCCYC - initiate a cycle.
- $CCCLK - set the clock. The clock is set on startup until a suitably value within 1 second of the computer time is reported back. If the modem reboots ($CAREV,...,INIT), the clock is set again.

[5] use CMOOSMsg::GetCommunity() to extract the community

[6] see Micro-Modem Mainboard Software Interface Guide at `http://acomms.whoi.edu/umodem/documentation.html` for details on these messages

- $CCCFG - configure NVRAM value. All values passed to `cfg=` will be passed to $CCCFG at startup.
- $CCCFQ - query configuration values. $CCCFQ,ALL is sent after all the $CCCFG lines to log the NVRAM parameters.

To directly monitor the modem feed, subscribe to ACOMMS_NMEA_IN and ACOMMS_NMEA_OUT. If you wish to control the modem directly, write a valid $CC NMEA string (without newline or carriage return) to ACOMMS_NMEA_OUT.

3.1.12 Medium Access Control (MAC) Unit

...documentation coming soon...

3.2 pGeneralCodec

pGeneralCodec is a MOOS process that provides a standalone interface to the DCCL library. It can be used in the same way as pAcommsHandler's DCCL unit (see all subsections under section 3.1 that refer to DCCL). If using pAcommsHandler with pGeneralCodec, set `encode=false` and `decode=false` in the MOOS ProcessConfig block for pAcommsHandler so you don't get duplicate encoding and decoding. Most new applications should just use the DCCL encoding and decoding built into pAcommsHandler. pGeneralCodec is still supplied to maintain backwards compatibility for legacy applications.

3.3 pCTDCodec

pCTDCodec is a deprecated MOOS module that is superceded by the delta-difference encoding available in pAcommsHandler via the <max_delta> tag combined with either <repeat> or <array_length>.

3.4 iCommander

3.4.1 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 3.1.6 for the configuration of these files):

```
LatOrigin = 42.5
LongOrigin = 10.08333

// where to find the file specifying modem lookups
modem_id_lookup_path = ../../../data/acomms/modemidlookup.txt
Community = topside

//---------------------------------------------------------------
// iCommander configuration  block
ProcessConfig = iCommander
{
  // available to all moos processes.
```

```
  AppTick    = 4
  CommsTick  = 4

  // available to all tes moos processes

  //verbose, terse, quiet
  verbosity = verbose

  // for checking xml structure correctness
  // highly recommended to use this
  // requires path relative to xml file location (or full path)
  schema = ../../acomms/libdccl/message_schema.xsd

  message_file = ../../../data/acomms/nafcon_command.xml
  message_file = ../../../data/acomms/nafcon_report.xml

  load = iCommander_autosave.txt
}
```

Filling out the .moos file

- `verbosity`: choose verbose for full text terminal output, terse for symbolic heartbeat output, and quiet for no terminal output.
- `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.

3.4.2 Reference Sheet

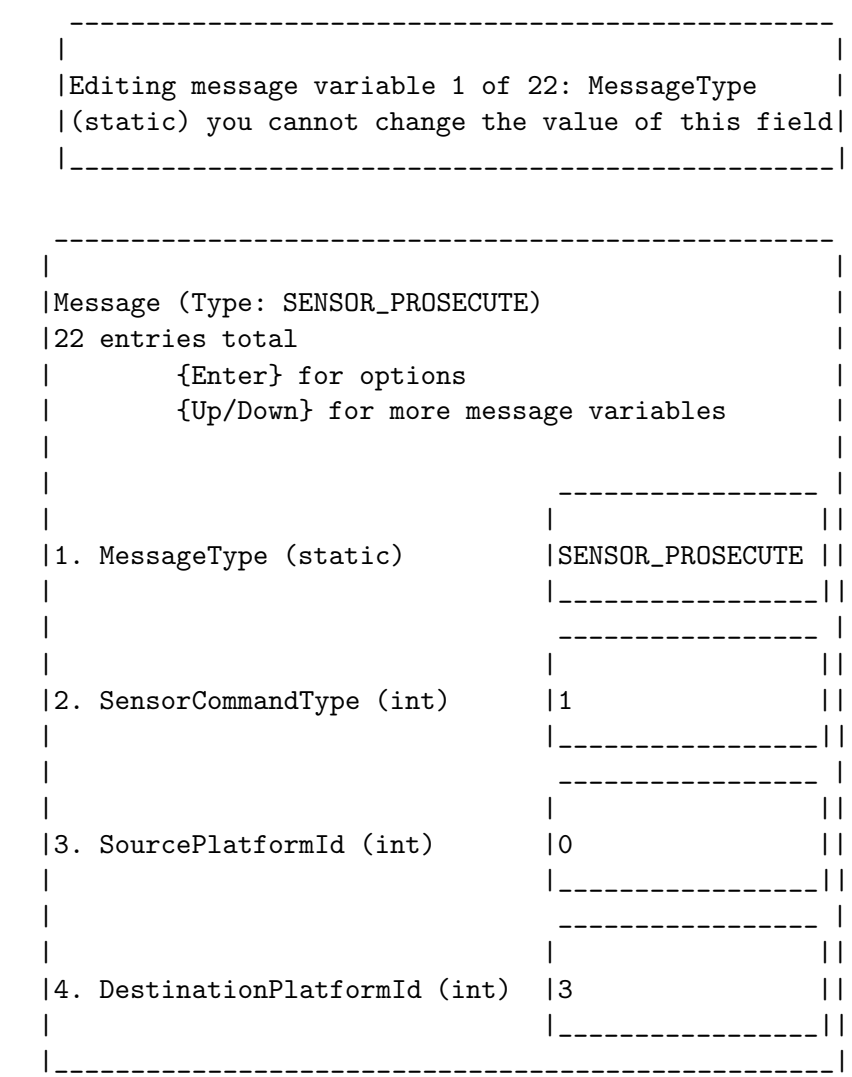
Main Menu

```
 ------------------------------------------------------------
|            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



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
- 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 pGeneralCodec's algorithms, and pAcommsPoller.
- 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 PROSECUTE message would look like this:

```
 ------------------------------------------------------------------------------
|                                                                              |
| Message acknowledged from queue: OUT_PROSECUTE_HEX_30B at time: 2009-Mar-03 03:53:41 |
|______________________________________________________________________________|
```

References

A Glossary

- *message_var*: the term for a field within a message. a *message_var* can be of several types: int, bool, double, float, string, or enum. the *message_var*s are defined within the `<layout />` part of the message.
- *modem_id*: the number given to each whoi micromodem (this is like a variable MAC address) that defines senders and receivers. modem_ids must be unique for each network and are configured using `$CCCFG,SRC,#`, where `#` is the modem_id (integer from 0 to 127?).
- *publish*: the term for a given action to be performed upon receipt of a message. this term is used since this action will involve a moos publish to some variable (after some string parsing/formatting).
- *XML*: extensible markup language: a specification for defining a custom markup language, which is a set of annotations given to text to indicate structure. here we use XML to indicate the structure of a "message" and its subsequent breakup ("publishes", during decoding)
  - *tag*: the name given to the "metadata" in the XML file. for example: the tag `<message />` indicates the start and end of a message (this is "metadata"), but nothing about what it contains (which is"data")
  - *attributes*: data contained within a tag. for example, in `<int algorithm="to_upper"></int>`, **algorithm** is an attribute of the tag **int**.
  - *CDATA*: **C**haracter **DATA**, or the data contained within a tag or an attribute. for example, in `<name>nav_x</name>`, **nav_x** is CDATA.

Index